University of Kentucky
UKnowledge

University of Kentucky Master's Theses | Graduate School

2004

FAULT LINKS: IDENTIFYING MODULE AND FAULT TYPES AND THEIR RELATIONSHIP

Inies Raphael Chemmannoor Michael
University of Kentucky, irchem2@uky.edu

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation
Michael, Inies Raphael Chemmannoor, "FAULT LINKS: IDENTIFYING MODULE AND FAULT TYPES AND THEIR RELATIONSHIP" (2004). *University of Kentucky Master's Theses*. 226.
https://uknowledge.uky.edu/gradschool_theses/226

This Thesis is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Master's Theses by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF THESIS

FAULT LINKS: IDENTIFYING MODULE AND FAULT TYPES AND THEIR RELATIONSHIP

The presented research resulted in a generic component taxonomy, a generic code-fault taxonomy, and an approach to tailoring the generic taxonomies into domain-specific as well as project-specific taxonomies. Also, a means to identify fault links was developed. Fault links represent relationships between the types of code-faults and the types of components being developed or modified. For example, a fault link has been found to exist between Controller modules (that forms a backbone for any software via. its decision making characteristics) and Control/Logic faults (such as unreachable code). The existence of such fault links can be used to guide code reviews, walkthroughs, testing of new code development, as well as code maintenance. It can also be used to direct fault seeding. The results of these methods have been validated. Finally, we also verified the usefulness of the obtained fault links through an experiment conducted using graduate students. The results were encouraging.

KEYWORDS: fault based analysis, fault links, fault chains, component, taxonomy, validation and static analysis

Inies Raphael C.M.

09/21/2004

Date

Copyright © Inies Raphael Chemmannoor Michael 2004

FAULT LINKS: IDENTIFYING MODULE AND FAULT TYPES AND THEIR RELATIONSHIP

By

Inies Raphael Chemmannoor Michael

Dr. Jane Hayes
(Director of Thesis)

Dr. Greg W. Wasilkowski
(Director of Graduate Studies)

09/21/2004
(Date)

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgements.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name	Date

THESIS

Inies Raphael Chemmannoor Michael

The Graduate School

University of Kentucky

2004

FAULT LINKS: IDENTIFYING MODULE AND FAULT TYPES AND THEIR RELATIONSHIP

THESIS

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science in the College of
Engineering at the University of Kentucky

By

Inies Raphael Chemmannoor Michael

Lexington, Kentucky

Director: Dr. Jane Hayes, Professor of Computer Science

Lexington, Kentucky

2004

MASTER’S THESIS RELEASE

I authorize the University of Kentucky Libraries
to reproduce this thesis in whole or in part
for purposes of research.

Signed: Inies Raphael C. M.

Date: 09/21/2004

To my parents, Joyce and Sebin

ACKNOWLEDGMENTS

In the course of this thesis, several people were instrumental in leading to its timely completion. First, my Thesis Chair and mentor Dr. Jane Hayes provided excellent scholarly support. This work would not have been possible without her incessant guidance. She extended her constant encouragement and expertise to shape me into the researcher that I am today. At moments when I would falter, her words “You can do it Inies” indeed were a morale booster and inspired me to achieve my goal. In addition, I would like to thank my Thesis Committee: Dr. Mukesh Singhal and Dr. Alexander Dekhtyar, who provided valuable technical suggestions, thereby helping me achieve better standards in my thesis.

I would also like to take this opportunity to thank other members who, despite not having extended technical input, nevertheless were highly instrumental in helping me complete this endeavor successfully. I want to express my gratitude towards my friends, here at UK and in India, who were an immense moral support during my Master’s. Especially, I would like to mention my friend Miss. Kalyani Bharadwaj for her constant encouragement and guidance throughout my Master’s program. Last but not the least; I would do complete injustice if I failed to mention my parents, without whom I would not stand tall on this day. From the very beginning, my parents instilled in me the ability to embrace healthy competition in all walks of life and the desire to achieve my goals with undying perseverance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter One

Introduction

Background

Recent issues such as severe virus attacks, software incompatibilities, system and software hacking, and competitions between software firms have increased the demand for high quality and reliable software. The Blaster worm, for instance, created chaos by crashing numerous vulnerable Windows machines across the Net. The worm has revolutionized the rules on malicious code attacks, causing Microsoft to release numerous patches to guard against the problem. Even the latest release of Windows Service Pack 2 has been reported to have incompatibility problems that impact more than 10 percent of the Windows XP PCs. Some of the main reasons for the failure of software companies to produce quality products are lack of resources (time, money, CASE tools, etc) to ensure software quality, lack of knowledge regarding the timely usage of apt resources, and inability to organize the developed products or those under development. Hayes et. al., [27] state that "software developers are struggling to develop high quality, reliable software systems while staying on schedule and on budget, and users are still struggling to use the resulting software in the most effective ways". Fault Based Analysis (FBA) and Fault Based Testing (FBT) are related technologies that seek to address this problem. These technologies provide software firms with the necessary and sufficient knowledge to enhance their development process, thereby ensuring software quality with available resources.

Fault-based testing (FBT) and Fault-based analysis (FBA) are two different techniques that when implemented together compliment each other to produce valuable results. In general, fault-based testing is intended to generate test data that can demonstrate the absence of any pre-specified faults. On the other hand, fault-based analysis introduced by Hayes [27], is used to determine static techniques (such as traceability analysis). More than just static techniques, FBA can go a step deeper to even determine specific activities within those techniques (e.g., perform back-tracing to identify unintended functions). These activities should be performed to ensure that a set of pre-specified faults do not exist. Using historical data, FBA can be used to identify the type of faults that are most likely to occur. For example, developers of version 10 of a software system could use information on the number and type of faults from versions 8 and 9 to guide their code walkthroughs [27]. Fault-based analysis can also be used to perform risk analysis to identify faults that can have devastating effects on the project outcome if ignored. Static techniques identified using FBA are applied as part of verification and validation (V&V) effort. In addition to fault-based testing, fault-based analysis can be used to improve the efficiency of the V&V for any software development effort. Fault-based analysis is most commonly applied to development of Critical Catastrophic High Risk (CCHR) systems.

Based on our work on a semantic model of faults [54], Offutt's work on testing coupling [52], our work on traceability [28], and requirement faults [27], we developed 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 as a "fault link" which is part of a larger relationship or sequence called a "fault chain." Fault chain refers to a relationship that exists between faults that have occurred in different stages of the software development life cycle. That is, a fault that appeared in the final stages of the life cycle may be traced back to its root that exists in an earlier stage of the life cycle and vice versa. For example, the control/logic fault type of unnecessary processing caught during software testing may be traced back to ambiguous requirement fault that exist in the requirement specification.

Our present research concentrates only on fault links. A fault link is a relationship between the type of module being developed or changed and the fault type. For example, we posit that if a developer is writing a Computational-centric module, it is more likely that a computational fault will be introduced than any other type of fault. Though this may seem intuitive or "not surprising," currently there are no empirical results to confirm it. The need for experiments in software engineering has been acknowledged for many years [61]. New technologies or changes in the process should be tested before they are implemented, in order to determine their impact in the specific context [47].

In our research, we present methods that will provide this knowledge about fault links to software companies to overcome their limitations and contribute to software quality. We strongly believe that similar type of projects under the same domain will contain the same type of components and same types of faults. Therefore, we will apply the methods developed in our research to a set of projects under a domain and use the results obtained to support the quality assurance of similar projects in the future. We view each and every software product as belonging to exactly one domain. For example, Apache web server software belongs to open source web based software; VIsual editior iMproved (VIM) belongs to text editor software, etc. In our current research, we chose a domain called "online course management software" and applied our methods to projects under this domain.

Motivation

Factors such as, increasing demand for high quality software, lack of sufficient resources, limited knowledge of resource application, and the presence of stiff competition among software products have resulted in software firms funding researchers to provide them with a solution to solve their existing problems. The software companies are finding it difficult to use all their resources within the period of time allotted to produce quality software. It is a known fact that efficient usage of all possible resources will result in high quality software, but not all resources can be used within the allotted time frame. Therefore, knowledge about correct and timely usage of resources will help save money and time to obtain quality software.

In general, software can be grouped into different domains based on implementation environment and application. Software that belongs to different domains may need different types of resources. For example, a Distributed Data Management System might need techniques that have the capacity to detect or prevent control/logic and data faults. Thus, it is necessary that the software engineer be able to identify the types of resources that are to be used for a particular domain or even a particular type of project within the domain. Therefore, software engineers without the necessary knowledge spend most of the available time and money in trying to find the apt technique to establish software quality. If they know what kinds of faults will occur in their domain and what kind of techniques can find them, then they can more optimally allocate their resources.

In this paper, we present a methodology that can be used by software engineers to answer the question “Are these resources useful to induce quality into this project under this domain?” The taxonomies presented in this paper can be used to group the components that form software based on their type. This kind of grouping will help software engineers to keep the software organized and well maintained.

Objective

The overall goal of this research is that current software development practices can be improved with the knowledge of fault links; especially the manner in which resources are used to ensure that the quality of the software can be improved. If we can demonstrate that fault links exist and if we can codify them, we can improve the development, testing, and maintenance of complex computer systems in several ways. We can offer preventive items for walkthrough checklists for newly developed code. We can recommend that exit criteria be added to walkthrough checklists for maintained code. For example, if for a particular project, we know that a fault link exist between the computational-centric component and computational fault then for any computational-centric module is being examined for the project, do not exit the walkthrough until an extra check has been made to ensure that no computational errors exist. We can offer a list of fault-based tests that should be conducted based on the fault links. We can guide the allocation of verification and validation resources to best reduce risk. Finally, we can offer guidance to testing and reliability researchers who rely on fault seeding as a mechanism for evaluating their techniques. As pointed out in [54], we tend to seed syntactically small faults. Through our fault link research, we can gain deeper insight into true distributions of fault types and understand what types of faults we should be seeding based on module type.

This paper presents a means to identify the relationship that exists between the type of component and the types of faults (i.e., fault links). The generic taxonomies established and the approach to categorize code fault and components presented in this paper can be used in any project that belongs to a specific domain. The research also provides indirect insights to a new concept of project-domain architecture that enables efficient project maintenance. By project-domain architecture we mean the project-to-domain mapping based on the project implementation and project-to-type mapping based on the project goals and priorities. Using these mapping schemes, any software firm that deal with numerous projects under various domains can get their projects organized for better maintenance. Projects can be organized based on their type or based on the domain to which they belong.

Scope

Although the research provides an efficient methodology to ensure the enhancement of software quality and reliability, it has its own limitations and constraints. The generic taxonomies and the processes that shall be discussed were intended to be applicable to any project or domain. But the results obtained from these processes are domain-specific, i.e. the results are unique to the domain for which the processes were applied and therefore, are not applicable to other domains. Moreover, the research does not provide any concrete guidelines on project and domains classification. This may result in a project being classified differently by different researchers, based on their own interpretation of the project description.

The research is entirely dependent on the availability of sufficient project data (i.e., code bug reports, project descriptions, component descriptions, etc.) that are accurate, which unfortunately is almost close to impossible. The correctness of the results obtained depends on the correctness of these project data. The methodology also requires that the software product be composed of various unique purpose components with accurate description for each of them. Currently, the software development process followed by the software firms is of inadequate quality. Consequently, the above-mentioned limitations tend to persist for all types of research.

Utility

We will explain the utility of our research with a practical scenario. Consider a software firm that uses the concept of modularization to develop a software product. The product is now ready to undergo testing in order to ensure quality and reliability. The firm decides on performing a code walkthrough as a part of the testing phase. There are numerous components that contribute to the efficient functioning of the software product. The firm is faced with limited amount of time and money to deliver the product, limited knowledge about the type of components and faults, etc. With these limitations it is not possible to perform code inspection on all the components and also to check for all faults. The generic taxonomies presented in this paper will help the engineers to categorize the components of the software and will also present them with the knowledge of all the possible faults. The domain-specific and the project-specific taxonomies obtained by executing our methods on similar projects, will limit the number of components that need to be inspected. It will also limit the type of faults that need to be prevented to ensure quality. The fault links information obtained from our method on similar projects will save time when inspecting various types of component. In other words, when a component of a particular type (categorized based on the generic component taxonomy) is undergoing inspection, the fault link information will enable the code inspectors to check for the relevant faults, instead of expending time in checking for other non-related fault types.

Our methodology, when applied, results in increased efficiency of software processes, thereby increasing software product quality with limited resources. The paper is organized as follows: Generic component and fault taxonomies are presented in Chapter two. Chapter three describes the processes to tailor generic taxonomies and the process to identify fault links. Validation of the work is presented in Chapter four, along with the results obtained by applying the process to the online course management domain. Related work is presented in Chapter five. Chapter six is devoted to conclusions and future work.

Copyright © Inies Raphael Chemmannoor Michael 2004

Chapter Two

Generic Taxonomy

<u>**Definitions**</u>

Fault: It is an incorrect internal state that is the manifestation of some software error [4].

Component: A component can be a single statement or a single function or procedure that contributes to the purpose of the program [4, 33].

Taxonomy: According to the Webster's dictionary, taxonomy is a classification of faults or components based on some similarities or relationships.

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.

In our research, we deal with two types of generic taxonomies: *generic component taxonomy* and *generic code-fault taxonomy*. The generic taxonomies have been obtained through an exhaustive literature survey and also by applying the process of categorization on two projects. The projects used are Apache web server (version 1.23.x) and Mozilla web browser (version 3.23.x). Both the generic component taxonomy and the generic fault taxonomy are shown in Appendix A and in Appendix B, respectively. These taxonomies are used as inputs to our processes. These processes have also performed updates that have resulted in several changes to the both our taxonomies. The following subsections discuss both updated taxonomies in detail.

Component Taxonomy

Any simple or complex program can be viewed as a component or a combination of components. Each component serves a unique purpose for the program. Figure 1 presents a pictorial representation of our component taxonomy. In our research we have identified two methods for classifying the components. The methods are discussed below.

- Classify by purpose: The components of a program are classified based on their main purpose. This method is easy to comprehend and apply and is also faster than method two. However, it does not easily lend itself to automation.
- Classify of LOC: The components are classified based on the percentage of lines of code that perform specific functions, such as computation, data manipulations, etc. We count the number of lines that belong to a particular category in a component, select the category with the highest Lines Of Code, and assign the component to that category. For example, "If (salary > 1000)" is a controller statement. This method is advantageous in that it provides information about the statements used in the program and can be easily automated with some standard guidelines. Unfortunately, this method is plagued with certain drawbacks including: (i) not easy to perform categorization, (ii) time consuming, (iii) tedious when performed manually, and (iv) not easy to understand.

In this particular research we make use of classify by purpose for classifying components. Each module category is described below.

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

- <u>Computational-centric:</u> At the module level, modules whose main purpose is to calculate or compute results belong in this category. At the statement level, any statement that changes any variable or state of the program falls under this category.
- <u>Controller:</u> Any module whose main purpose is to control the sequence of program execution falls under this category. The collaborative modules or statements form a backbone to software because they decide on the instructions to be executed and the number of times they are to be executed.

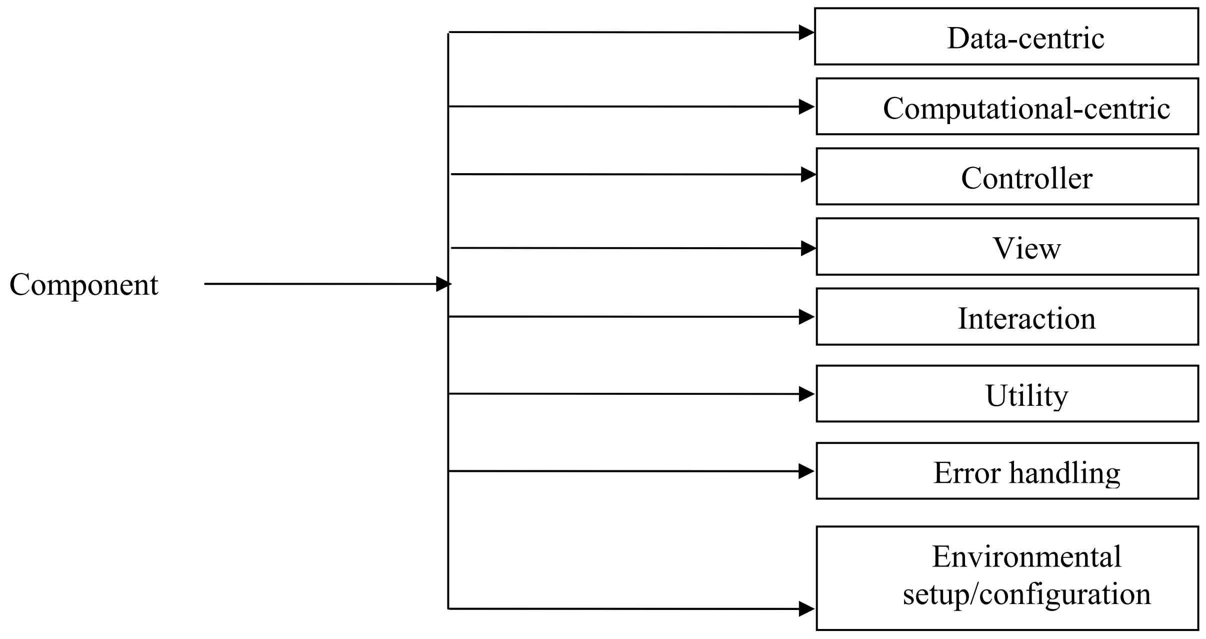


Figure 1. Generic component taxonomy

- <u>View:</u> Any module that designs or handles graphical user-interface controls or manipulates the attributes of the controls is part of this category. Also, the statements used for displaying information belong to this category.
- <u>Interaction:</u> Any module or statement that performs a function call or passes parameters to other modules or tries to access the data structures outside the module falls under this category.

- Utility: The main purposes of the modules that fall under this category are to provide additional services for the enhancement of the entire software and to support other modules to carry out their functionality efficiently.
- Error Handling: The main purpose of the modules that come under this category is to handle exceptions or errors that are likely to occur, when the software is either dormant or active.
- Environmental setup/configuration: The main purpose of the modules is to set up an appropriate environment for the software to function efficiently.

Code-fault Taxonomy

Our fault taxonomy does not include errors that can be caught by the compiler at compile time. Moreover compile time faults are easily detected during development by the developer. We attempted to make the module and fault taxonomies generic enough to be language independent and method independent (i.e. the method used to implement the project that you are working with can be procedural or object-oriented or component-based, etc). This section provides us with a detailed description of all the fault types that make up our generic code-fault taxonomy 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 will only be applicable for languages such as C or assembly languages.

The fault taxonomy also takes practical realities into account. Specifically, the taxonomy only relies on bug reports or problem reports and does not assume that (up to date) specifications or design are available for analysis. The following fault types are significant and have been included because they have been shown to be important fault categories in the past [4, 18, 21, 25, 40, 43, 62, 64].

Basic Concepts and Definitions:

Before reading ahead it is necessary to understand some of fundamental concepts and definitions that form the basis for our research.

Software error: is defined as a static defect in the software [4, 5]

Software fault: It is an incorrect internal state that is the manifestation of some error [4].

Software failure: is an incorrect external behavior with respect to the requirements or other description of the expected behavior.

Consider a situation in which a patient visits the doctor's office with a list of failures (that is, symptoms). The doctor then must diagnose and discover the error, or root cause of the symptom. To aid in the diagnosis, the doctor may then conduct some diagnostic tests that will help identify anomalous internal conditions, such as high blood pressure or high cholesterol. The abnormal internal conditions correspond to faults in our terminology. This analogy not only helps us to clearly understand the definitions of some of the above terms (errors, faults, and failures) but also aids to distinguish between them. The definitions of fault and failure also allow us to distinguish testing from debugging.

Apart from the basic definitions and concepts mentioned above, there are also other terms and concepts defined in the literature that can be relevant to our research. The following are the list of additional terms and concepts.

Testing: The process of evaluating the correctness of the software is called testing. It is generally carried out right after the implementation but prior to handing the software to the user.

Debugging: The process of trying to locate the error that leads to failure is called debugging.

Faults by omission: Faults generated due to omission of functions rather than due to improper functioning are generally termed as faults by omission [41].

Faults by commission: Faults generated due to incorrect or improper execution of functions are defined as faults by commission [41].

<u>Code-fault Taxonomy Definitions:</u>

1. 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.

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 divide 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 result in faults that fall under this category.

Example: “int x;” instead of “float x;”

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

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

1.3. Data accessing: The process of accessing data from data structures that are defined accurately is called data accessing. The accessed data structures are presumed to be correctly defined. The following is a list of subcategories of fault types that are grouped under data accessing:

Examples:

i. Incorrect data type for processing or incorrect storing and retrieving of data or incorrect data referenced
ii. Data flow anomaly: involves the sequence of accesses to an object: e.g., reading an object before it is created or creating and then not using an object.

Points to remember under this category:

a. The faults can be fault by omission and faults by commission.
b. Editing or updating Database (DB) with some computation does NOT belong to this category.
c. DB access without some computation belongs to this category.

d. We assume that the programming language under consideration assigns default values implicitly if the programmer failed to initialize data.

2. Computation:

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. Errors during computation may manifest themselves as faults that belong to this category. The faults are due to commission and not to omission.

2.1. Incorrect equation: Errors in the equation used for computation may result in incorrect results. Note that the term "may" instead have "will" in the definition. This is because there are situations in which an incorrect equation can sometimes give correct results.

Example:

i. A= B + C instead of A = B / C

ii. A = A * (2 + B) / C instead of A = A * 2 + (B / C)

2.3. Wrong manipulation: These faults arise from incorrect execution of computational operations.

Example: Append instead of precede

Points to remember under computation-related faults:

a. Incorrect editing, deleting or updating of data structure values belongs in this category.

b. Incorrect editing or updating of table fields through computation belongs in this category.

3. Interface:

Modularized software is made up of a number of modules, each with a unique purpose and functionality. A module may or may not interact with other modules. However, if there is an interaction it will be through exchange of data. Interface between modules are established by their interaction. Interface is also established between a module and an external data structure when the module makes use of the data structure in its local environment. Errors during establishing such interfaces may result in faults that fall within the interface-related faults category. This fault type is further sub-divided into the following categories.

3.1. Incorrect module interaction: For a module to interact with another module it has to invoke or call the other module with the relevant parameters. If the module invokes a wrong module it may result in faults that belong to this category.

Example: Invoking add() instead of avg()

3.2. Incorrect module-external data structure interaction: In order for a module to interact with an external data structure, it has to use the name of the data structure to access it. Wrong data structure invocation may result in faults that fall under this category.

Example: Accessing array A [] instead of array B []

3.3. Incorrect input parameters: As stated earlier, for a module to invoke another module, it not only requires the name of the module, but also needs to pass the necessary parameters required by the invoked module.

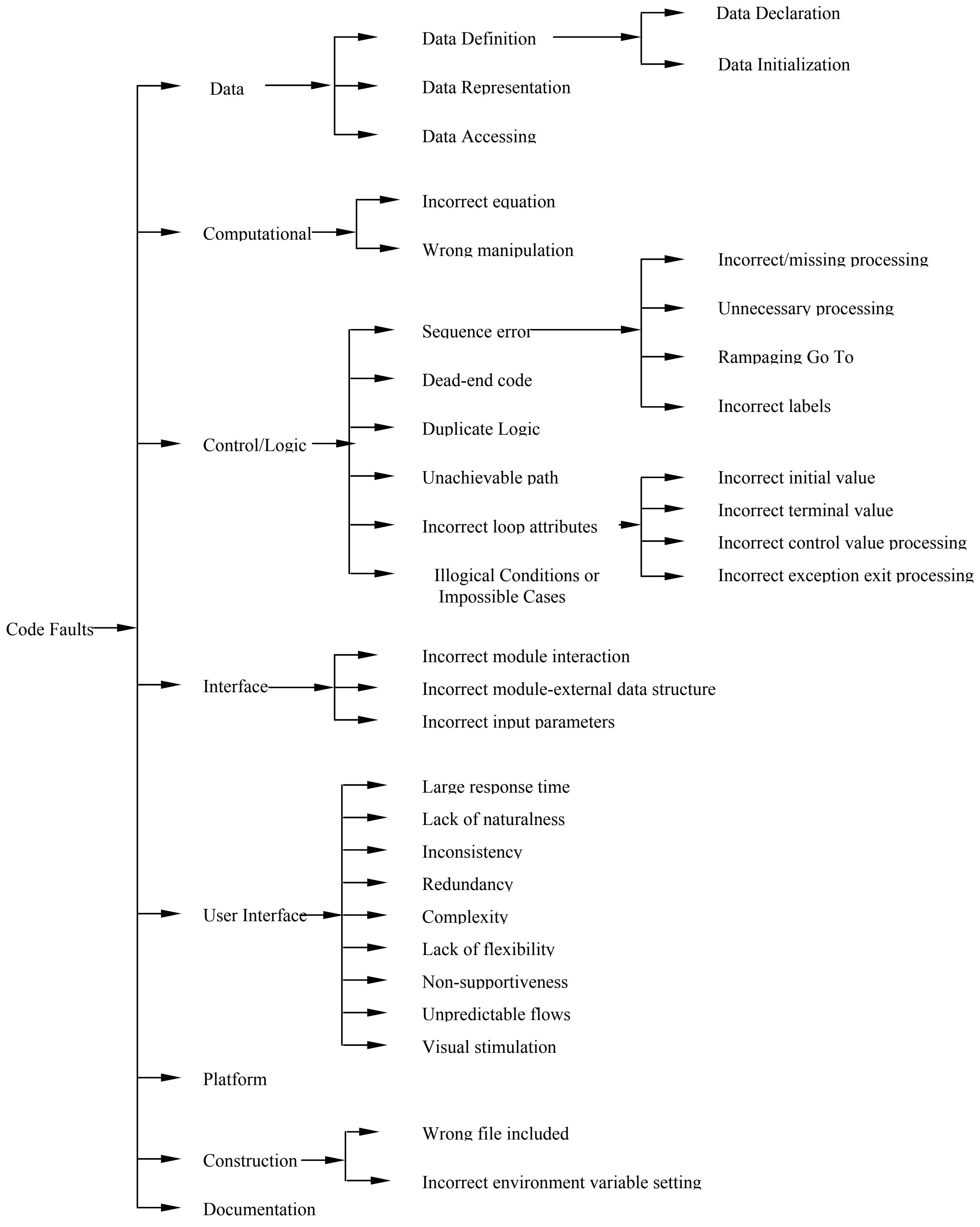


Figure 2. Generic Code Fault Taxonomy

Wrong parameters being passed may result in wrong output from the module or may even end up invoking a wrong module. Faults manifested due to these type of errors fall under this category.

Example: add(x, y) instead of add(x, z)

Note: y and z belong to the same data type

Points to remember under interface-related faults category:

a. We assume that the data structures defined are external to the local environment of the components that are using it.
b. The external data structure that we are referring to does not include DB.

4. Control / Logic:

The control and logic statements form the backbone of any software 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. Faults manifested due to errors in these statements fall under this category.

4.1. Unachievable path or unreachable code [5]: Despite a specific code segment being part of a functionally meaningful path in the code, errors in control logic statements can cause the path or code to be unreachable.

4.2. Dead-end code [5]: Although a code segment requires an exit, errors can result in statements that only allow entry but not exit. Such code is called dead-end code.

Example: infinite loop

4.3. Duplicated logic: Control logic statements that need to be executed only for a specified number of times, can occasionally be executed beyond the requirement. These statements or logic sequences that are the result of duplication are not necessary and may serve as sources of errors.

4.4. Sequence error: Faults manifested due to incorrect order or sequence of execution of the control/logic statements belongs to this category. The subcategories in this category are listed below.

> **4.4.a.** Incorrect/missing processing: The improper program code execution may lead to incorrect or sometimes even missing functionality.
>
> **4.4.b.** Unnecessary processing: Incorrect sequencing of program code may lead to unintended processing, thereby resulting in software's large response time or even in wrong functionality.
>
> **4.4.c.** Rampaging Go Tos: Go to statements causing unnecessary and incorrect processing due to their frenzied behavior are termed as rampaging.
>
> **4.4.d**. Incorrect labels: Some programming languages use the concept of labeling statements in order to help efficient control flow transfers. Therefore, wrong labeling of statements causes incorrect sequence of execution.

4.5. Incorrect loop attributes: The loop statement is one of the control statements. The loop statement consists of a control variable that controls the loop. The control variable has an initial and terminal value that undergoes some processing within the loop. Faults that are caused due to incorrect initialization and processing of control variables fall under this category. Hence, based on these attributes, this fault category can be further sub-divided as:

4.5.a. Incorrect initial value: The starting value of the control flag is wrong.

4.5.b. Incorrect terminal value: The ending value of the control flag is wrong.

4.5.c. Incorrect control value processing: The processing carried out on the control flag during the loop execution is wrong.

4.5.d. Incorrect exception exit processing: The condition imposed on the control flag for the loop to stop execution and exit is wrong.

4.6. Illogical Conditions or Impossible Cases (ICOIC): This category is illustrated using the following examples:

- " if (a == a) " - this is an illogical condition
- " if (a != a) " - this is an impossible case and is also an illogical condition
- "constant A =10; constant B =20; if (A > B) " - this is an impossible case.

Points to remember under the control/logic related faults category:

a. All undefined functions fall under this category
b. Missing processing or condition checks fall under this category.

c. Incorrect Logic fall under this category. For example, does not handle some type of input.

5. User Interface (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. Faults so formed belong to the UI fault type.

5.1. Large response time [43, 62]: Response time is the time the user has to wait for a response from the interface after performing some action. Large response time can frustrate the user.

5.2. Lack of naturalness [40] : Lack of naturalness in the user interface causes the user to alter his or her approach significantly, which may be undesirable from the user's perception. Some of the main issues include: improper ordering of interface, use of language (jargon) not understood by the user, using phrases that are not self-explanatory, etc.

Examples:

i. A user interface designer might refer to a task as "updating a file." However, if the user comprehends it as "posting" then the UI designer must also employ the same dialogue.

ii. Use of “mv,” “cp” (UNIX) are examples of non-self explanatory phrases.

5.3. Inconsistency [25, 40, 62]: Whenever a user works with one part of the system, the user builds up an expectation regarding the meaning and layout of the controls on the screen. The user expects the meaning and layout of controls to be consistent throughout the system. Any inconsistency found will frustrate the user. Therefore, it is important to maintain a consistent interface.

Example: From PCs to cash dispensers, people have become accustomed to confirming a command by pressing Return or Enter. Diversion from norms may cause confusion.

5.4. Redundancy [40]: Non-redundancy of a user interface requires minimal inputs and outputs to the users. For example, the user should never be allowed to enter information that can be automatically generated by the system. Also, the system should not provide too much information that is detrimental to the user.

5.5. Complexity [62]: A complex interface is not very easy to work with for any type of user. Some of the issues regarding complexity of the user interface include: lack of ease of use, lack of ease to learn, and lack of ease to navigate. For example, a complex UI is never for a novice user easy to understand and learn.

5.6. Lack of flexibility [25, 40]: Flexibility of the user interface refers to how well it can cater to or tolerate different levels of user familiarity. For example, different types of dialogue may be used in different situations. Initially hierarchical menu structure can be provided to first time users, and once the user gets familiarized with the GUI, he can use command and parameters.

5.7. Non-supportiveness [40]: Supportiveness of a user interface in the running system refers to the assistance provided to the user by the interface. There are three main issues regarding supportiveness, viz., quality and quantity of instructions provided, nature of the error message, and confirmation of what the system is doing. For example, a display of an “hour glass” to indicate some background operation being carried out by the system should be present.

5.8. Unpredictable flows: is when the flow of control in the user-interface gets beyond the control of the user. An example of unpredictable flow is when the user tries to perform a spell check on her document and the software also performs a thesaurus function, despite not being invoked by the user.

5.9. Visual stimulation [40, 62]: refers to faults dealing with the improper use of color, fonts, graphics, control layout, etc.

Example:

i. Label of button incorrect.

ii. Incorrect positioning of checkbox.

iii. Incorrect size of the frame.

Points to remember under UI-related faults category:

a. To debug some of the faults, it is necessary to have a well-written requirement specification, in addition to the source code of the system.

6. Construction:

Some software requires that a proper environment be setup even before the software starts its execution. The environment setup or configuration actions may be implemented as part of the software itself. Errors occurring during establishing an apt environment for the software may result in faults that fall under this category.

6.1. Wrong file included: Some programming languages in which software can be implemented require that certain files be included for their accurate execution. For example, languages like Java and C require that certain files are imported and included respectively for their execution. Wrong files included may result in faults that fall under this category.

Note: These files that are included contain definitions of function and commands that are used within the source code. Two or more files may contain different definitions for the same command. So it is necessary to include the correct file to get the desired result.

6.2. Incorrect environment variable setting: In order to configure the environment for correct execution of the software, we might need to set appropriate values for some environment variables. Wrong variable assignment or incorrect values for the variables may result in faults that fall under this category.

Examples:

i. Incorrect setup of the mode in which the software works.

ii. Incorrect inclusion of deprecated or wrong files.

7. Platform: This fault type was found during our domain-process implementation on projects under our chosen domain. 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.

8. Documentation: Beizer in his textbook [5] states that the most common kind of coding bug, and often considered the least harmful, are documentation bugs. When we refer to documentation we not only refer to the comments inside the source code but also any external documentation that describes the components, data structures, or any tool used by the program.

An example of a documentation fault is a misleading or erroneous comment.

Copyright © Inies Raphael Chemmannoor Michael 2004

Chapter Three

Processes

<u>Process to extend a taxonomy</u>

We have built and adopted a method for extending or tailoring a taxonomy, as mentioned earlier. In our research, we applied the processes discussed here to both the component and fault taxonomies.

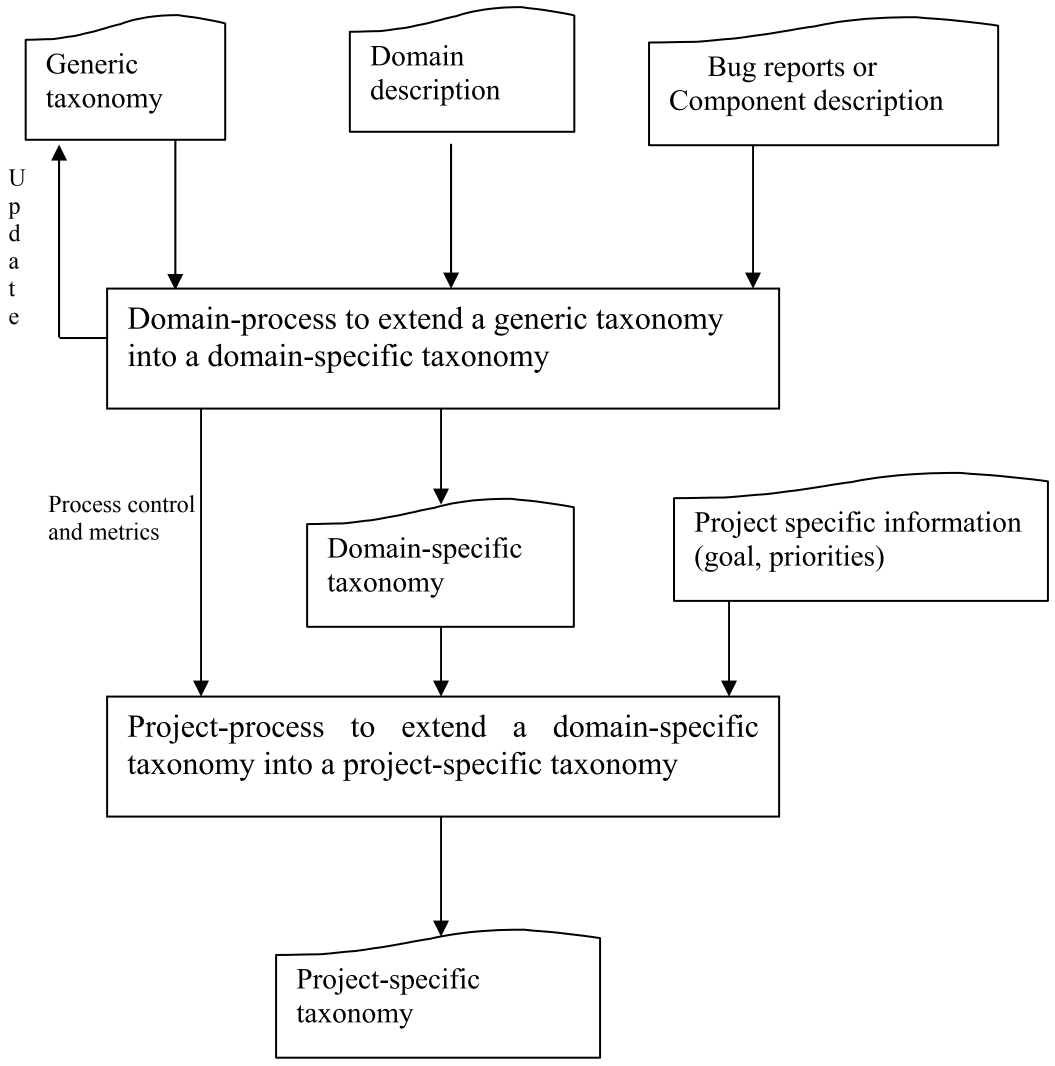

Generic taxonomy
Domain description
Bug reports or Component description
Update
Domain-process to extend a generic taxonomy into a domain-specific taxonomy
Process control and metrics
Domain-specific taxonomy
Project specific information (goal, priorities)
Project-process to extend a domain-specific taxonomy into a project-specific taxonomy
Project-specific taxonomy

Figure 3. Process outline

We split our process for extending the taxonomy into two parts: Domain-process and Project-process. Domain-process discusses all the activities that are to be carried out to develop a domain-specific taxonomy. Project-process discusses all the activities that are to be carried out to develop a project-specific taxonomy. The outputs of Domain-process are inputs to Project-process. Our process for extending the taxonomy is shown in figure 3.

The Domain-process was built on our generic taxonomy that was discussed in section 2. First, we perform Domain-process using the generic taxonomy, domain description, and bug reports for the projects. The result is a domain-specific taxonomy. We also perform some process control activities and collected related metrics. In parallel to Domain-process we update the generic taxonomy when we find any bug reports representing a new category. This is followed by the execution of Project-process is performed. The inputs to Project-process are the outputs from Domain-process and some process control and metrics maintained in Domain-process. The result of Project-process is a project-specific taxonomy. We believe that there will be a substantial difference in the proportion of categories between our initial generic taxonomy and the final extended taxonomy.

Table structure

The processes that are used in our research follow the table structure discussed in [NAS2-98028]. The table consists of six fields: entry criteria, activities, exit criteria, inputs, outputs, and process controls and metrics. The structure of the table is shown in Table 1.

Table 1. Process Table Structure.

Entry Criteria	Activities	Exit Criteria
Inputs	**Process Controls/Metrics**	**Outputs**

The “entry criteria” field describes a checklist of pre-conditions that must be met before the process activities begin. The “activities” field describes the list of actions that need to be carried out in order to come up with the desired result. The “actions” listed in the activities field must be carried out in the order in which they are given.

The “exit criteria” field describes a checklist of pre-conditions that must be met before the process activities can stop. The “inputs” field describes the checklist of items that are needed for the process activities. The “outputs” field describes the checklist of items that the process will provide after satisfying the exit criteria. The process controls ensure version control and configuration control of the taxonomy and also quality control of the activities. The process metrics keeps note of some standard effort and quality metrics information for the process.

Domain-Process

The process for developing a domain-specific taxonomy is shown in Table 2. All the information and data needed, such as the generic taxonomy, projects description, list of projects from the domain, and domain definition must be available before the process starts. If we are working with fault taxonomy, then we need bug reports for all the projects. If we are working with a component taxonomy, we need component descriptions and source code for all the projects. In addition, it is necessary to have authorization from all the project owners to implement the processes on their projects.

The activities to be performed include selecting a domain and generic taxonomy, selecting a project from the list of chosen projects, examining component descriptions (or problem reports), categorizing components (or faults) based on the generic taxonomy, updating the generic taxonomy when new categories are found, determining the frequency of categories plus their percentage of occurrence, and identifying crucial categories. These activities are performed for each project in the list one by one. The process controls and metrics section keeps a log of the results obtained for each project. In the end, we used this result log to establish a domain-specific taxonomy.

We also estimate the component (or fault) frequency for all available projects in the chosen domain. Table 3, shown below, illustrates the accumulation of fault frequency information for the domains. Then, we identify the code fault types, fault frequency count, and percentage of fault occurrences for the domain, by accumulating the corresponding values for each project.

Overall, 1000 code faults were found for the domain. The percentage of occurrence of data faults is therefore 4% for the domain. The table can also be used to determine the frequency count and percentage of occurrence of project components in the domain. In our research, we worked only with projects that belonged to a single domain (i.e., online course management), but the table can be used to implement multiple domains.

Finally, we determine the historically most probable categories for the chosen domain. For this purpose, we make us of Table 4 shown below. We list the top three major categories for the domain. We then assign a complexity of high, medium, or low depending on the category's frequency. When using the process on the fault taxonomy, if certain faults are found more frequently for a domain, then it is crucial to seek improvement in that area and to attempt to prevent and/or detect these fault types.

Table 2. Domain-process to Extending a Generic Taxonomy into a Domain-specific Taxonomy.

Entry Criteria	Activities	Exit Criteria
1. All inputs are available 2. Authorization to view the data of all projects 3. Authorization to implements process A on the data 4. Projects chosen belong to the domain 5. Projects chosen are real time projects 6. Projects have its code modularized into components with descriptions	1. Select a domain and a list of projects within the domain 2. Select a generic component (or fault) taxonomy 3. Select a project from the list of projects 4. Examine the component description (or problem report) for the project 5. Categorize the components (or faults) for the project according to the generic taxonomy 6. Update the generic taxonomy with new categories that cannot be categorized under existing categories 7. Repeat step 3 to 6 for each project in the list of projects 8. Determine frequency of categories for the domain and percent of fault occurrences 9. Identify crucial categories for the domain 10. Establish a domain-specific taxonomy	1. All outputs are produced
Inputs	**Process Controls/Metrics**	**Outputs**
1. Generic component (or fault) taxonomy 2. Project descriptions 3. Domain definition 4. If dealing with fault taxonomy we need bug reports for the projects 5. If dealing with component taxonomy, we need component descriptions and source code	Controls: 1. Maintenance of configuration control of taxonomy 2. Maintenance and management of project data 3. Maintenance and management of categorization results by project Metrics: 1. Person Hours of effort 2. # of projects 3. # categories 4. frequency of categories 5. % of category occurrence 6. Top 3 Historical category types for the domain	1. Frequency counts of categories and its percent of occurrences in the domain chosen 2. Crucial categories for the domain 3. Domain-specific taxonomy

The outputs of this process are the frequency counts of the categories, percentage of occurrence, and the crucial categories for the domain. We repeat the process until our exit criteria is met (i.e., we have developed a domain-specific taxonomy along with all our desired outputs). The process controls ensure that all versions of our taxonomy (both generic and domain-specific) are properly maintained under configuration control. Also, our Project-process requires that the results of the categorization be maintained by project. Process metrics include person hours for the effort, number of projects, number of categories, etc.

Table 3. Determination of Critical Code Faults for a System.

System	Historical Top 3 Most Probable Function Areas (Critical Code Faults)
Domain A : Open source web software (e.g., APACHE, MOZILLA)	1): Data .1: Data Definition .2: Data Representation 2): 3):

Domain B: Text Editors
(e.g., Notepad, MS word, Pico)
	1):
	2):

Table 4. Estimation of Fault Frequency for Software Code Fault Types.

S/W Code Fault Types	Count of Fault Frequency	% of Fault Occurrences
1) Major Fault: Data 0.1 Data Definition 0.2 Data Representation 0.3 Data Accessing	 20 10 10	 2 % 1% 1%
: :		
N) New Fault 0.n Subfault		
Totals	1000	100%

Project-process

As mentioned earlier, Project-process employs the outputs and the process controls/metrics data from Domain-process. The process for developing a project-specific taxonomy is illustrated in Table 5. In order for the activities of Project-process to commence, all information such as domain-specific taxonomy, project description and its priorities, and process controls/metrics data from Domain-process must be present. It is also necessary to have completed a successful implementation of Domain-process, and the necessary authorization to implement Project-process on the project data.

The activities performed include selecting a project from the list of projects chosen in Domain-process, obtaining the categorization results for the project from Domain-process, checking for any inconsistencies between the results and the domain-specific taxonomy, determining frequency of categories and their percentage of occurrence, identifying crucial categories, and finally, establishing the project-specific taxonomy.

We use the same tables and procedures as in Domain-process to determine the frequency and percentage of occurrence, and also to identify the crucial categories in the project.

The outputs of the process were category frequency and its percentage of occurrence, top three crucial categories, and a project-specific taxonomy. The process controls maintain the different versions of the taxonomy. The process metrics keep track of the number of person hours for the effort, number of categories, etc. The process was repeated until the exit criteria were satisfied (i.e., a project-specific taxonomy is established along with all the outputs).

Table 5. Project-process to Extending a Domain-specific Taxonomy into a Project-specific Taxonomy.

Entry Criteria	Activities	Exit Criteria
1. Domain-process is successfully implemented 2. All inputs are available 3. Authorization to implement process B on the data 4. Projects chosen are one of the projects used in process A that belongs to the chosen domain	1. Select a project from the list of projects chosen in Domain-process 2. Obtain the categorization result for the project from Domain-process 3. Check for any inconsistencies between the results and the domain-specific taxonomy 4. Determine frequency of categories for the project and its percentage of occurrence 5. Identify crucial categories for the domain 6. Establish a project-specific taxonomy	1. All outputs are produced
Inputs	**Process Controls/Metrics**	**Outputs**
1. Domain-specific taxonomy 2. Project descriptions 3. Project priorities 4. Domain definition 5. Process control /metrics from Domain-process	Controls: 1. Maintenance of configuration control of taxonomy 2. Maintenance and management of project data 3. Maintenance and management of categorization result for the project Metrics: 1. Person Hours of effort 2. # category 3. frequency of category 4. % of category occurrence 5. Top 3 Historical Fault areas for the project	1. Frequency counts of categories and percent of occurrences in the project 2. Crucial categories for the project 3. Project-specific taxonomy

Component-process to identify fault links.

The process for identifying fault links is illustrated in Table 6, shown below. Fault links represent relationships between the types of code-faults and the type of component being developed or modified. For example, data-centric components from a particular project may historically have data faults or historically data faults may occur in data-centric components. The Component-process makes use of the same table structure discussed in Section 3.1.1. For the process activities to begin, we need to have certain information that include project-specific component taxonomy, project-specific fault taxonomy, comprehensive bug report for the project, list of components and their classification, and a list of faults and their classification. In addition to the above listed information, the process also requires that both Domain-process and Project-process were successfully implemented and the necessary authorization to implement Component-process on the project has been obtained.

Within Component-process, we then selected the project used in Project-process, the project-specific component taxonomy, and the project-specific fault taxonomy. We also collected the list of components that belonged to the project, selected a component from the list, and identified the historical bug types that occurred in the component. We kept track of the component type and the types of fault that occurred, repeating the steps for all components in the list and grouping them based on their type. We identified the top three fault types for each component type and finally established a component type-specific taxonomy.

We used Table 3 to determine the fault frequency for different component types under the chosen project. For example, we use the table for data-centric, computational-centric, controller, view, interaction, error handling, and environmental setup components under the project. Then, we identified the code fault type, fault frequency count and the percentage of fault occurrence for each component type.

Finally, we used Table 4 as before to determine the top three crucial faults for each component type. We listed the top three major code faults for each component type. We also assigned a complexity of high, medium, and low depending upon the fault's frequency. If certain faults were found more frequently for a certain type of component, then it is necessary to use methods in the future to either to prevent or detect such fault types for such component types.

Table 6. Component-process to Identifying Fault Links.

Entry Criteria	Activities	Exit Criteria
1. Domain-process and Project-process are successfully implemented 2. All inputs are available 3. Authorization to implement process C on the data	1. Select the project used in Project-process 2. Select the project-specific component taxonomy 3. Select the project-specific fault taxonomy 4. Collect the list of components that belong to the project 5. Select a component from the list 6. Identify the bug types that have occurred in the component historically 7. Keep note of the component type and the types of faults that occur in the component 8. Repeat steps 5 to 7 for all the components 9. Group components based on their type 10. Identify top 3 fault types for each component type 11. Establish a component type-specific taxonomy	1. All outputs are produced 2. A component type-specific taxonomy
Inputs	**Process Controls/Metrics**	**Outputs**
1. Project-specific component taxonomy 2. Project-specific fault taxonomy 3. Detailed bug report for the project 4. List of components and their classification 5. List of faults and their classification	Controls: 1. Maintenance of configuration control of taxonomy 2. Maintenance and management of project data 3. Maintenance and management of categorization result for the project Metrics: 1. Person Hours of effort 2. # category 3. frequency of category 4. % of category occurrence 5. Top 3 Historical Fault areas for the project	1. Component types and their relevant faults 2. Top 3 crucial fault types for each component type 3. Component type-specific taxonomy

The outputs of the process included component types and their relevant faults, major fault types under each component type, and component type-specific taxonomy. The process was repeated on all the components in the list until the exit criteria (i.e., all outputs and the component-type taxonomy are produced) were met. The process controls section maintains and manages the project data along with the different taxonomies from Project-process. The process metrics includes person hours for the effort, number of components, number of faults, etc.

Copyright © Inies Raphael Chemmannoor Michael 2004

Chapter Four

Experimental Validation

In this chapter, we demonstrate the effectiveness of our approach through the applications of the processes discussed before. We begin with the experimental design, followed by our research hypotheses, and the results obtained by applying each process to the input data. We will use the results from Component-process to evaluate our listed hypotheses. Finally, we will evaluate the correctness and usefulness of the results obtained from Component-process, with the results obtained from our experiment conducted using a group of subjects.

Experimental Design

We chose a domain, which we named online course management system. The course management system is a software designed to help educators create quality online courses [60]. Such e-learning systems are sometimes also called as Learning Management System (LMS) or Virtual Learning Environment (VLE). The data set for the experiment (or processes) came from two sources (projects) and belonged to the chosen domain. The two projects are Electronic Personal Organic Chemistry Homework (EPOCH) and Integriertes Lern-, Infomations- und Arbeitskooperations System (ILIAS). In English, ILIAS means Integrated Learning, Information and Cooperative working System

EPOCH is an online homework management program and serves as a teaching aid in an organic chemistry course at the University of Kentucky [59]. EPOCH attempts to give students 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.

ILIAS [60] is a web−based learning management system (LMS) implemented in PHP, which was originally developed in the VIRTUS project at the University of Cologne and has now become an Open Source project. ILIAS consists of tools for learning, authoring, information access and co−operative work. It presents an integrated environment for learning and teaching on the Internet. ILIAS authors can create entire courses within a team and publish them on the web. Students can create groups to work through learning material and communicate with each other or with their tutors.

Domain-process and Project-process were applied to both component and fault taxonomy. We also used the generic taxonomies shown in Appendix A and Appendix B as input for Domain-process. As mentioned earlier, Domain-process not only determines the domain-specific taxonomy but also updates the generic taxonomy whenever it finds a new category. Component-process employs the outputs from the execution of Project-process to the domain-specific component taxonomy and those from applying Project-process to the domain-specific fault taxonomy.

The output from Component-process consists of a list of all component types (obtained from Project-process for the chosen project) and a list of the major fault types that can occur in each component type of component present in the project. This output was used to evaluate the list of hypotheses discussed in the next sub-section. We will also show the results obtained from an experiment to determine the correctness of the outputs.

Research Hypothesis

After developing the fault and component taxonomy along with the processes for extending the same, we noticed a strong correlation between the categories. This raised the following research questions: “Are the final results obtained from our processes correct and useful?” If so, “Does the component type have any effect on the fault type one encounters?” In order to address these issues, we came up with several research conjectures regarding the correctness of the results and the usefulness of fault links. The following 10 fault links were posited.

H1.1 – Data-centric components have a higher percentage of Data faults.

H1.2 – Data faults occur more frequently in Data-centric components.

H2.1 – Controller components have a higher percentage of Control/Logic faults.

H2.2 – Control/Logic faults occur more frequently in Controller components.

H3.1 – Computational-centric components have a higher percentage of Computational faults.

H3.2 – Computational faults occur more frequently in Computational-centric components.

H4.1 – Interaction components have a higher percentage of Interface faults.

H4.2 – Interface faults occur more frequently in Interaction components.

H5.1 – View modules have a higher percentage of User interface faults.

H5.2 – User interface faults occur more frequently in View components.

We also posited 10 secondary research conjectures. These are not as intuitive as the above, and some are contrary to the above conjectures.

H6.1 - Utility components have a higher percentage of Control/Logic faults.

H7.1 - View components have a higher percentage of Interface faults.

H7.2 – Interface faults occur more frequently in View components.

H8.1 – Construction faults occur more frequently in Controller components

H9.1 – Error Handling components have a higher percentage of Control/Logic faults.

H9.2 – Control/Logic faults occur more frequently in Error Handling components.

H10.1 – Environmental Setup/Configuration components have a higher percentage of Construction faults.

H10.2 – Construction faults occur more frequently in Environmental Setup/Configuration components.

H11.1 – Platform faults occur more frequently in View modules

H12.1 - Environmental Setup/Configuration components have a higher percentage of Platform faults.

Figure 4 below is a schematic representation that summarizes all the research hypotheses listed above. In the figure, we use rectangular boxes for component types and fault types, and arrows to illustrate the relationships between them. Boxes in the top row of the figure form the list of component types and those in the bottom row form the list of fault types. For example, the controller component may *have* control/logic faults and the control/logic faults may *occur in* the controller component.

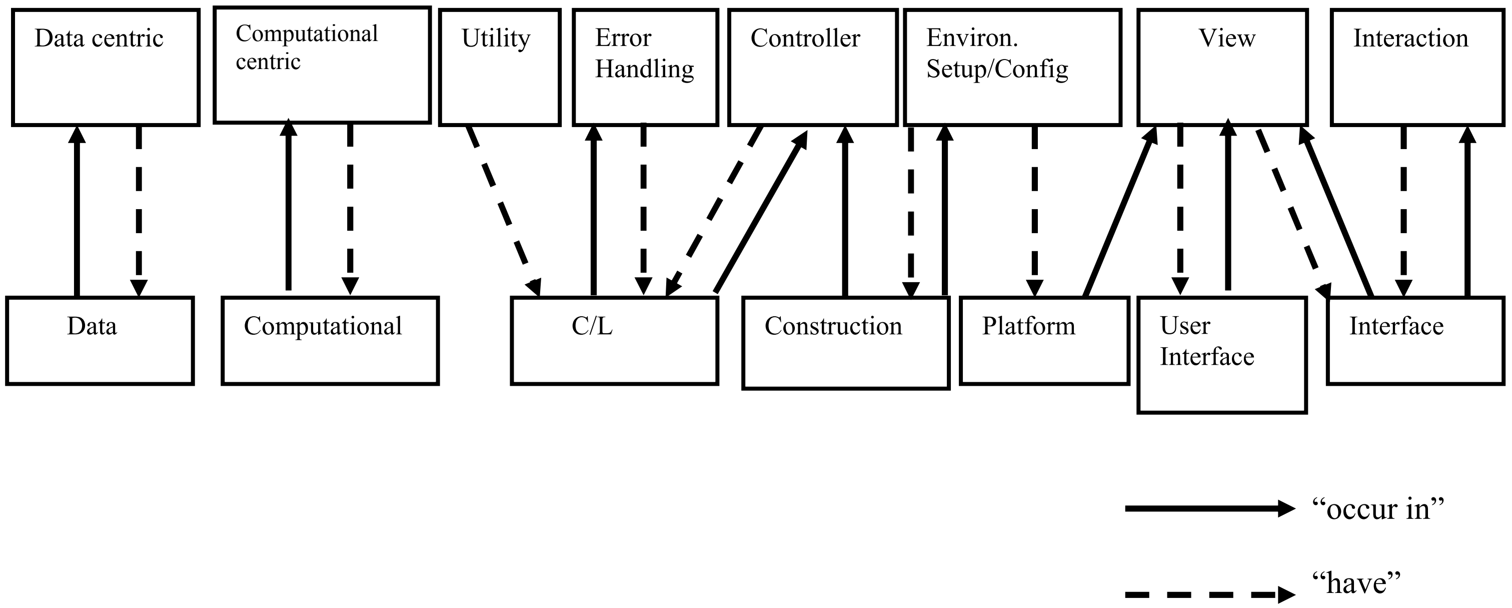


Figure 4. Research Hypotheses

Besides the aforementioned hypotheses that were validated directly by results obtained from executing the processes on the input data of a particular domain, we have formulated other research hypotheses that focus on the usefulness of the results. Here, we concentrate only on the final results obtained from Component-process.

H13 – The results will be useful for Testers in testing similar projects of the same domain.

H14 – The additional knowledge provided in our tailored checklist will help the experimental team in our code inspection process.

The following sections will present us with the results obtained from Domain-process, Project-process, and Component-process. The validity of the hypotheses and the usefulness of the results will be discussed in section 4.

Establishing a Domain-specific Taxonomy

Implementing Domain-process on a collection of projects from a particular domain, results in a domain-specific taxonomy. In this section, we present the results obtained from the execution of Domain-process on the project data obtained from the chosen collection of online course management projects. As mentioned in section 3, we, in our research, executed Domain-process to establish both the domain-specific component taxonomy and the domain-specific fault taxonomy. In this section, we discuss the implementations and the results of both the executions individually. One of the main inputs to the process is a generic taxonomy. Appendix A and Appendix B show the generic component taxonomy and the generic fault taxonomy, respectively. The following sub-sections present the results from Domain-process on both the component and the fault taxonomy.

Domain-specific Component Taxonomy

We applied Domain-process on the generic component taxonomy discussed in section 3.1.2 and on the project data from the online course management projects: EPOCH and ILIAS. Prior to the execution of the process activities, we made sure that we met both the entry criteria and had all the inputs listed in Table 2. After meeting the criteria and the input requirement, we proceeded with the process activities.

The domain and the projects within it were chosen. As mentioned before, the domain was online course management and the projects chosen under it were EPOCH and ILIAS. We used the generic component taxonomy (Appendix A) for categorizing the components of the projects. As a first step towards categorization, we chose the EPOCH project from the list of projects. We categorized its components one by one using the component description and the generic taxonomy. During the process of component categorization, we found that certain components did not belong to any of the existing categories. Therefore, we had to come up with new categories and definitions to accommodate these components. The newly found category was added to the generic taxonomy as a part of the update process. Then the updated generic taxonomy was used until all the components in EPOCH were categorized. The EPOCH project had a total of 45 components. After the component categorization in the EPOCH project, we carried out the same steps of categorization for all the components in the ILIAS project. The total number of components that were present in the ILIAS project was 39. The process controls and metrics were maintained for each of the projects separately. The results from the process are presented below.

At the end of the process execution, we found two new categories of components: Utility and Error Handling.

- Utility: The main purposes of the modules that fall under this category are to provide additional services for the enhancement of the entire software and to support other modules to perform their functions efficiently.
- Error Handling: The main purpose of the modules that belong to this category is to handle exceptions or errors that are likely to occur, when the software is either dormant or active.

The newly found categories were added to both the generic component taxonomy and also to the domain-specific taxonomy. Table 7 shows the frequency count and the percentage of component occurrence for the domain (i.e., online course management). We can see that the data-centric component type has a frequency count of 57 and a percentage of occurrence of 34%, etc.

Table 7. Frequency Count and Percent of Occurrence of Components (Domain-process).

S/W Code Component Types	**Count of Component Frequency**			**% of Component Occurrences**
	EPOCH	**ILIAS**	**Total**	
1) Data-centric	37(43%)	20(25%)	57	34.34 %
2) Computational-centric	4(5%)	8(10%)	12	7.23 %
3) Controller	14(16%)	10(12.5%)	24	14.46 %
4) View	2(2%)	27(34%)	29	17.47 %
5) Interaction	4(5%)	1(1%)	5	3.01 %
6) Error Handling	16(18%)	1(1%)	17	10.24 %
7) Utility	7(8%)	11(13%)	18	10.84 %
8) Environmental Setup/Configuration	2(2%)	2(3%)	4	2.41 %
Total	86	80	166	100

Table 8. Top three Historically Critical Component Types (Domain-process)

System	Historical Top three Most Probable Function Areas (Critical Code Components)
Domain A : Online Course Management (e.g., EPOCH, ILLIAS)	1): Data –centric 2): View 3): Controller

Table 8 shows the top three component types for the domain online course management. The online course management domain has Data-centric, View, and Controller as the top three component types.

The process controls and metrics information, such as, number of components, and their categorization, were maintained for each project individually. The data from these metrics serve as a part of the input to Project-process. The values of the process metrics for the entire domain (i.e., both projects together) are: person hours of effort were 20hours, number of projects was two, and number of components was 166.

Domain-specific Code-Fault Taxonomy

We applied Domain-process to the generic code-fault taxonomy discussed in section 3.1.2 and also to the relevant project data from the online course management projects: EPOCH and ILIAS. Prior to the execution of the process activities, we made sure that the entry criteria were met and the inputs were available (listed in Table 2). We then executed the process activities.

The domain and the projects within it were chosen as before. We used the generic code fault taxonomy (Appendix B) for categorizing the components of the projects. As a first step toward fault categorization, we chose the EPOCH project from our list of projects. We categorized its faults one by one using the fault description from the bug reports and the generic taxonomy. During the process of categorization, we found that certain faults did not fall under any of the existing categories; hence we had to propose new categories and definitions to accommodate these faults. The newly found category was added to the generic taxonomy as a part of the update process. The updated generic taxonomy was then used until all the reported faults in EPOCH were analyzed and categorized. Just as a note at this point, we would like to mention that not all of the bugs reported turned out to be actual coding faults. For example, the reported bug may either be an enhancement request or a non-coding fault or just a suggestion to improve the software, etc. The total number of actual code faults in the EPOCH project was 86. After completion of the EPOCH project, we performed the same steps of categorization for all the faults in the ILIAS project. The total number of code faults reported in the ILIAS project was 39.

The process controls and metrics were maintained for each of the projects separately. The results from the process are presented below.

At the end of the process execution, we found one new category of faults: Platform. This fault type was found during our Domain-process implementation on projects under our chosen domain. The software product works fine under one operating environment but does not do well in another. The fault type is not due to varying environment settings, but due to lack of options to set the environment.

The newly found categories were added to the generic fault taxonomy and also to the domain-specific code fault taxonomy. Table 9 shows the fault frequency count and the percentage of fault occurrence for the domain (online course management). From Table 9, we observe that the control/logic faults have a frequency count of 31 and a percentage of occurrence of 36.9%, the data faults have a frequency count of 24 and a percentage of occurrence of 28.57%, etc.

Table 9. Frequency Count and Percent of Occurrence of Faults (Domain-process).

S/W Code Fault Types	Count of Fault Frequency			% of Fault Occurrences
	EPOCH	ILIAS	Total	
1) Data	11(28%)	13(29%)	24	28.57 %
2) Computational	1(2.5%)	2(4%)	3	3.57 %
3) Control/Logic	15(38%)	16(35.5%)	31	36.9 %
4) User Interface	6(15%)	6(13%)	12	14.29 %
5) Interface	1(2.5%)	6(13%)	7	8.33 %
6) Platform	5(13%)	1(2%)	6	7.14 %
7) Construction	0(0%)	1(2%)	1	1.19 %
8) Documentation	0(0%)	0(0%)	0	0 %
Total	39	45	84	100 %

Table 10 shows the top three historical fault types in the domain. From the table, it is evident that the top three historical fault types for the online course management domain were control/logic, data, and user interface faults. These results were not surprising because the main characteristic of software under consideration was to perform decision-making tasks on data. Moreover, we are dealing with web-based software, which tends to involve numerous user-interface modules. However, it is an important knowledge when trying to enforce software quality.

Table 10. Top three Historically Critical Fault types (Domain-process)

System	Historical Top three Most Probable Function Areas (Critical Code Faults)
Domain A : Online Course Management (e.g., EPOCH, ILLIAS)	1): Control/Logic 2): Data 3): User Interface

As a part of Domain-process's process controls and metrics, we maintained information such as number of faults, and their categorization for each projects, individually. The data from these metrics were used as inputs to Project-process, to establish a project-specific code fault taxonomy. The process metric values for the entire process on fault taxonomy were: person hours of effort were 33hours, number of projects was two, and the number of faults was 84.

Establishing a Project-specific Taxonomy

In this section, we present the results from the implementation of Project-process on the project data of an online course management project; the project chosen was EPOCH. We chose EPOCH because it had all the necessary and sufficient information required to carry out the process. As in Domain-process, we applied Project-process to both the domain-specific component and domain-specific fault taxonomies obtained from Domain-process. We also used some of the process controls and metrics information from Domain-process. The results from the execution of Project-process to establish both the project-specific component taxonomy and the project-specific fault taxonomy are presented separately.

Project-specific Component Taxonomy

We executed Project-process (discussed in section 3.1.3) to establish the project-specific component taxonomy. Some of the outputs from Domain-process, namely the domain-specific component taxonomy, the process controls and metrics information, form part of the inputs to Project-process. Prior to the execution of the process activities, we confirmed that the entry criteria were met and the inputs were available listed in Table 5. We then performed the process activities.

We obtained the categorization list for EPOCH from the process controls of Domain-process. We used the domain-specific component taxonomy, also obtained from Domain-process, to verify for any inconsistencies in the results obtained. After the results were verified for inconsistency, we determined the frequency of the component categories and their percentage of occurrence. Finally, we identified the crucial component categories for the EPOCH project and established the project-specific component taxonomy. The results from the process are presented below.

Table 11 shows the component frequency count and the percentage of component occurrence. From the table, we can see that the data-centric component type has the highest frequency count of 37 and the highest percentage of occurrence of 43%, and both View and Environmental setup/configuration component type have the lowest frequency count of 2 and the lowest percentage of occurrence of 2%, etc.

Table 12 shows the list of the top three component types that are historically crucial for the project EPOCH. From the table, we can see that for the EPOCH project, the crucial component types are Data-centric, Error Handling, and Controller.

As the process was carried out, the process controls field maintained all the information listed in Table 5 of section 3.1.3. The process metrics results are: the person hours of effort was 2hours, and the number of components was 86.

Table 11. Frequency Count and Percent of Occurrence of Components (Project-process)

S/W Component Types	Count of Component Frequency (EPOCH)	% Component Occurrences
1) Data-centric	37	43 %
2) Computational-centric	4	5 %
3) Controller	14	16 %
4) View	2	2 %
5) Interaction	4	5 %
6) Error Handling	16	18 %
7) Utility	7	8 %
8) Environmental Setup/Configuration	2	2 %
Total	86	100 %

Table 12. Historically Top three Critical Component Types (Project-process).

System	Historical Top 3 Most Probable Function Areas (Critical Code Components)
Project: Electronic Personal Organic Chemistry Homework (EPOCH)	1): Data-centric 2): Error Handling 3): Controller

Project-specific Fault Taxonomy

We executed Project-process (discussed in section 3.1.3) to establish the project-specific fault taxonomy. As mentioned before, the outputs from Domain-process, namely the domain-specific fault taxonomy, the process controls and the metrics information form a part of the inputs to Project-process. Prior to the execution of the process activities, we made sure that the entry criteria were met and the inputs were available (listed in Table 5). We then carried out the process activities.

The chosen project was EPOCH. We obtained the code fault categorization list for EPOCH from the process controls of Domain-process. We used the domain-specific code fault taxonomy, also obtained from Domain-process, to verify for any inconsistencies in the results obtained. After the results were checked for inconsistency, we determined the frequency of the fault categories and their percentage of occurrence. Finally, we identified the crucial fault categories for EPOCH and established the project-specific code fault taxonomy. The results from the process are presented below.

Table 13 shows the fault frequency count and the percentage of fault occurrence for the EPOCH project. From the table, we observe that the Control/Logic fault type has the highest frequency count of 15 and the highest percentage of occurrence of 38%, and both Construction and Documentation fault types have the lowest frequency count of 0 and the lowest percentage of occurrence of 0%, etc.

Table 13. Frequency Count and Percent of Occurrence of Faults (Project-process)

S/W Code Fault Types	Count of Fault Frequency (EPOCH)	% of Fault Occurrences
1) Data	11	28 %
2) Computational	1	2.5 %
3) Control/Logic	15	38 %
4) User Interface	6	15 %
5) Interface	1	2.5 %
6) Platform	5	13 %
7) Construction	0	0 %
8) Documentation	0	0 %
Total	39	100 %

Table 14 shows the top three fault types that were historically crucial for the project EPOCH. They were control/logic, data, and user interface. The process controls field maintained the configuration control, the project data, and the categorization of faults for the project as listed in Table 5 in section 3.1.3. The process metric results are: person hours of effort were 2hours, and number of faults was 39.

Table 14. Historically Top three Critical Fault Types (Project-process)

System	Historical Top three Most Probable Function Areas (Critical Code Faults)
Project: Electronic Personal Organic Chemistry Homework (EPOCH)	1): C/L 2): Data 3): User interface

Thus, we have presented the results from the processes (Domain-process and Project-process) that were intended to extend or tailor the component and the fault taxonomies. Some of the important outputs from the successful execution of both the processes were a project-specific component taxonomy, a project-specific fault taxonomy, a list of components and their classifications for each project, and a list of faults and their classification for each project. The final two results are shown in Appendix C and D, respectively. These results will be used as inputs to Component-process for identifying fault links.

<u>Component-process to identify Fault Links</u>

We applied Component-process discussed in section 3.2 to the results obtained from the previous two processes and other inputs listed in Table 6. Before we began to execute the list of activities to identify the faults links for a chosen project, we made sure that the entry criteria listed in Table 6 were met. The inputs to the process were also made available prior to process execution.

The project for which the fault links were identified was EPOCH. We collected the list of components along with their categorization and the list of faults with their categorization from Project-process. As a part of the fault links identification process, we chose a component one by one from the list of components. Using the project-specific fault taxonomy and the bug report, we identified the types of bugs that have occurred in the component. The total number of components present in the list was 45 (i.e., all the components in EPOCH). After we identified the bug type for all the components from the list, we grouped them based on their type using the project-specific component taxonomy. Finally, we determined the top three fault types for each component type and established a component type-specific taxonomy.

As usual, the process controls of the process were maintained as discussed in section 3.2. The results of the process metrics were: person hours of effort were 5hours, number of components was 86, and the number of faults was 39.

Some general observations were made. First, many of the bug reports did not document bugs. Some of the bug reports represented enhancement requests. Bug reports had been generated by users who were "just trying out the bug tracking system." Second, many of the bug reports did not relate to code faults. For example, bug reports were written due to poor documentation in the user's manual. Third, many bug reports duplicated other existing ones. Fourth, many of the bug reports were not deemed errors by the developers. Finally, many bug reports documented more than one code fault and should have been separated into multiple bug reports.

We adjusted our approach to accommodate these findings. We first weeded out the "non-bug reports." Next, we disregarded bug reports not related to code. We then examined each fault in isolation, even if several had been grouped in one bug report.

As we did not examine the same number of modules for each type (e.g., we examined 37 Data-centric, but only 4 Computational-centric modules), we looked at the faults as a function of the number of faults per module. In other words, we examined 15 faults for two View modules. The 15 faults were categorized according to the fault taxonomy. The resulting values were then scaled to reflect 7.5 faults per module.

Let us first examine the columns of Table 15. The values are listed as [row, column] followed by the number of faults of that type per module type. The highest value in the row is bolded and the highest value in the column is bolded. For example, for the View/Data cell, View modules had 20% of Data faults, 74% of the Data faults occurred in View modules, and a total of 1.5 faults of the 7.5 faults per View module were categorized as Data faults.

It is clear that control/logic faults dominate this case study, regardless of module type. Though we had not conjectured this, it is not a surprising result. In our own experience as programmers, teachers, and lab assistants for junior level programming courses, we have also noticed that these errors dominate.

The results obtained by executing Component-process are shown in Table 15. As can be seen, the majority of the Data faults occur in the View modules (74%). The next highest value is 12.3% for Computational-centric modules. This finding does not support H1.2. The majority of Control/Logic faults occur in View modules (53%) with Computational-centric modules falling second at 26.5%. This finding does not provide support to H2.2 and H9.2. Computation faults occur 100% of the time in Controller modules and this does not support H3.2. Interface faults accounted for 100% of the View module faults, strongly supporting H7.2. User-interface faults occur 97.2% of the time in View modules. This strongly supports H5.2. Platform faults accounted for 82% of the total faults present in the View modules. EPOCH project did not have any Construction and Documentation faults to identify other fault links.

Table 15. Component-process Fault Links Identification (EPOCH).

Module Types	**# modules**	**Fault Types**								**Total Faults**	**Total faults /modules**	**%**
		Data	**Compn**	**C/L**	**UI**	**IF**	**PF**	**Cosn.**	**Doc**			
Data-Centric	37	[**40%**, 2.6%] 0.054	[0%, 0%] 0	[**60%**, 2.9%] 0.081	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	5	0.135	1.4%
Computational- centric	4	[20%, 12.3%] 0.25	[0%, 0%] 0	[**60%**, 27%] 0.75	[0%, 0%] 0	[0%, 0%] 0	[20%, 14%] 0.25	[0%, 0%] 0	[0%, 0%] 0	5	1.25	13%
Controller	14	[27%, 11%] 0.214	[9%, **100%**] 0.071	[**45%**, 13%] 0.356	[9%, 3%] 0.071	[0%, 0%] 0	[9%, 4%] 0.071	[0%, 0%] 0	[0%, 0%] 0	11	0.785	8%
View	2	[20%, **74%**] 1.5	[0%, 0%] 0	[20%, **53%**] 1.5	[**33%**, **97.2%**] 2.5	[6%, **100%**] 0.5	[20%, **82%**] 1.5	[0%, 0%] 0	[0%, 0%] 0	15	7.5	76.5%
Error Handling	16	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	0	0	0%
Interaction	4	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	0	0	0%
Utility	7	[0%, 0%] 0	[0%, 0%] 0	[**100%**, 5%] 0.142	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	1	0.142	1.4%
Environmental setup	2	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	[0%, 0%] 0	0	0	0%
Total	86	2.018	0.071	2.829	2.571	0.5	1.821	0	0		[9.8, 9.8]	
%		21%	0.7%	29%	26%	5%	19%	0%	0%			

Next, we examined the rows of the Table. The most frequently occurring fault type in Data-centric modules was Control/Logic at 60% with Data faults falling second with 40%, which does not support H1.1. The most frequently occurring fault type in Controller modules was Control/Logic at 45% with Data taking second place with 27%, strongly supporting H2.1. The most frequently occurring fault type in Computational-centric modules was Control/Logic at 60% with Data and Platform faults being second at 20% each. This does not support H3.1. The most frequently occurring fault type in View modules was User-interface at 33% with Data, Control/Logic, and Platform at a close second position at 20% each. This supports H5.1, but does not support H7.1. The Error Handling, Interaction, Utility, and Environmental setup modules did have any faults reported, which resulted in a few unconfirmed hypotheses.

Our conjecture findings are summarized in Table 16. Recall that we are trying to answer the question: "Does the module type drive the fault type?". The column "supported" from the table takes in either "yes" or "weak" or "no" or "-" as its value: "yes" indicates a definite fault link, "weak" indicates a weak link, "no" indicates that there is no fault link, and "-" indicates that from the data available for the project under examination, it is not possible to provide any conclusion. Seven conjectured fault links were supported, at least weakly. Thus we found evidence for answering "yes." A fault link that appeared universally, though not conjectured, was Control/Logic faults being the most prominent fault type for all module types. One could view this as an additional six fault links (data modules have Control/Logic (C/L) faults, computational-centric modules have C/L faults, Interaction modules have C/L faults, View, Error Handling, and Environment Setup/Configuration modules have C/L faults, etc.). This finding would lead one to answer the overarching question "no." Our results are still inconclusive, but appear to hold promise.

Apart from the predicted and the universal fault links, we also discovered new fault links that existed in the EPOCH project. These additional fault links are shown in Table 17. The following section makes use of all the identified fault links to verify the usefulness of the same.

Table 16. Conjecture Results.

Conjecture	Conjectured Fault Link	Supported?
H1.1	Data modules have Data faults	Weak
H1.2	Data faults occur in Data modules	No
H2.1	Controller modules have C/L faults	Yes
H2.2	C/L faults occur in Controller modules	No
H3.1	Computational modules have computational faults	No
H3.2	Computational faults occur in Comput. Modules	No
H4.1	Interaction modules have Interface faults	-
H4.2	Interface faults occur in Interaction modules	-
H5.1	View modules have User Interface faults	Weak
H5.2	User Interface faults occur in View modules	Yes
H6.1	Utility modules have C/L faults	Yes
H7.1	View modules have Interface faults	No
H7.2	Interface faults occur in View modules	Yes
H8.1	Construction faults occur in Controller modules	No
H9.1	Error handling modules have C/L faults	-
H9.2	C/L faults occur in Error Handling modules	-
H10.1	Environ. Setup/Config. Modules have Construction faults	-
H10.2	Construction faults occur in Environ. Modules	-
H11.1	Platform faults occur in View modules	Yes
H12.1	Environ. Setup/Config. Have Platform faults	-

Table 17. Newly Found Fault Links for the EPOCH project.

New Conjecture	Conjectured Fault Link	Supported?
N1	Data modules have C/L faults	Yes
N2	Computational modules have C/L faults	Yes
N3	Computational faults occur in Controller modules	Yes
N4	Data faults occur in View modules	Yes
N5	Utility modules have C/L faults	Yes

Verifying the Usefulness of Fault links

In this section, we present an experiment that was conducted to verify the usefulness of fault-component relationships established using our methodology. As defined earlier, we make use of the term fault links to refer to the fault-component relationships, which exist between the component and fault types. We strongly believe that the knowledge about fault links will aid software engineers – developers, testers, and maintainers – to enhance the software development process, and thereby, produce better quality software product. Thus, by proving the usefulness of the established relationships, we also prove indirectly, but intuitively, the usefulness of our methodology.

Experimental Design

Rationale: Although, our method establishes relationships that can be used in different phases of the software development life cycle, we in our current study due to limited resources, focus on verifying the application of fault links in helping software testers improve their testing techniques. We present a single experiment to verify our hypothesis. In our research, we hypothesize (H13) that the results (fault links data) obtained from our methods will be useful for software testers in testing similar projects in the future. We, therefore, concentrate on one of the many important testing processes called the code inspection. Software testers and/or software quality analysts to improve the quality of the software carry out the process of code inspection. Hence, the experiment that was conducted to verify our aim and that is about to be discussed below is a code inspection process.

Experimental Design: For the experiment, we had two teams of code inspectors and two supervisors. We named one as the control team and the other as the experimental team. Furthermore, within both the teams, the members were divided into different two-member groups. This grouping system helped us determine and compare the performances of both the teams in great detail. During the inspection process, every member of a team was restricted to communicating only with his/her group partner. Two supervisors to aid the inspectors with timely clarification of questions supervised the inspection process. The members of both the teams were provided with materials that provide a set of information not only to aid them with the understanding of the code but also to perform the code inspection with ease. In addition to the shared set of information between the teams, the experimental team was provided with some additional data and guidelines from our research. We believe that the possession of these additional data will improve the performance of any team; in our research we hypothesize a better performance from the experimental team. The format and content of all the materials provided to the teams followed current industrial standards. The student's t-test was used to analyze the performances between the two teams.

Experiment

As mentioned earlier, the main aim of the experiment is to answer the question "Do fault links established by our methodology aid in effective code inspection?" We believe that the process of code inspection, when carried out with the knowledge of fault links, delivers a high quality software product. The subjects for the experiment were upper division Computer Science students, who were enrolled in the course – CS499 Software testing – at the University of Kentucky. The students were taught the latest software testing concepts. They had sufficient knowledge and experience with code inspection and related testing techniques to carry out the experiment.

Days before the actual inspection, the inspectors were provided with materials that would help them understand the piece of code that they would be inspecting later. The materials distributed were: a component description (shown in Appendix C) describing the code about to be inspected and a tutorial of programming languages (esp. Java and SQL) briefly describing the concepts necessary for the understanding of the component. In addition to the above two items, the inspectors were also provided with a questionnaire (shown in Appendix G), which was completed by the inspectors before the code inspection. The answers to the questions present in the questionnaire will not only help determine the inspector's experience with code inspection, but will also be useful in evaluating their understanding of the required programming language concepts. Analysis of the completed questionnaire resulted in the selection of qualified inspectors for the experiment.

The total number of inspectors (i.e., students) selected for the experiment was 26. We randomly assigned 14 to the control team and 12 to the experimental team. The teams were further divided into groups, so that the control and the experimental team had 7 and 6 two-member groups, respectively. The groups were formed randomly by using a tool called the groupgenerator. The tool was developed as part of the experiment in PERL. The groupgenerator program accepts as input a text file that contains the names of all the inspectors. Using the time as a seed it randomly groups the inspectors into different groups and, finally, writes the list of groups into an output file.

After establishing the teams and the groups within them, the next step was to choose a component of a particular type from a project, followed by fault seeding. Since the EPOCH project was used to establish our fault links data, we chose a component from this project for our experiment. The component was chosen because it could be reasonably inspected within the available time. Using our component taxonomy (section 2), the component chosen was categorized as a data-centric component by our Project-process. The fault links data derived from the EPOCH project, indicate that a data-centric component type historically has 60% control/logic faults and 40% data faults. Using this information plus some analyses on the bug reports and help from the developer of the EPOCH project, we seeded faults into our chosen component. Of the total number of faults that were seeded, 60% of them were control/logic and 40% of the faults were data. Thus, we seeded a total of 16 faults, which were made up of 10 control/logic faults and 6 data faults.

On the day of the experiment, the supervisors distributed the necessary documents to help the inspectors carry out the inspection process. The documents that were common between the two teams (control and experimental) were: component source code, component type definition, fault taxonomy definitions and criteria, generic checklist, fault report sheet, and survey questionnaire. The source code was seeded with faults in the ratio mentioned earlier and was also line numbered (including blank lines and comments) to aid the inspectors to locate faults. The fault taxonomy definitions and criteria (discussed in section 2) aided the inspectors to categorize the faults found during inspection. The generic checklist (Appendix D) contains a list of generic questions related to generic issues in software code. The questions or items indicate the current set of knowledge available to software engineers to perform code inspection. The inspectors used the fault report sheet (Appendix F) to provide a description of the discovered faults. It was also used to indicate the difficulty (easy, medium, or hard) in finding faults. The survey questionnaire (Appendix H) was used to get feedback from the inspectors after the inspection process. For example, the answer to the question “Any feedback on the documents provided to you?” helped us to evaluate the usefulness and correctness of the documents distributed.

Besides the above-mentioned common documents, the experimental team received some additional documents. According to our hypothesis, this additional information should guide the experimental team to more readily identify faults. The additional documents were: tailored checklist, component taxonomy definition, and the results from our methodology. The tailored checklist is constructed based on our results from Component-process. From the results, we know that a Data-centric component historically would have 60% of control/logic faults and 40% of data faults. Since the component under inspection is a data-centric component, the items in the tailored checklist make sure that the inspectors using the checklist will be able to efficiently identify the faults. For example, the item “Are variables used in the IF statements correct?” helps the inspectors to ensure the correctness of the IF statement (i.e., a control/logic statement). The component taxonomy definitions (discussed in section 2) aid the inspectors to understand the main purpose of the component to be inspected. Therefore, the inspectors in the experimental team will not only be able to save time by looking only for control/logic and data faults, but also will be able to locate almost all the faults that were seeded. Thus, the additional documents plus the common documents should help the experimental team members to carry out the inspection process with ease and efficiency.

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. The experiment began on-time and went on smoothly for the allotted time. The supervisors at the end of the process promptly collected the populated fault report sheet and survey questionnaire.

<u>Results and Discussion</u>

In this section, we present the results obtained from our code inspection process, and perform a statistical analysis. Finally, we present our views on the analysis performed. The results from the process of code inspection are shown in Table 18. For convenience sake, we named the experimental team 'A' and the groups within the team 'A1', 'A2', 'A3', etc. Similarly, the control team is named 'B' and its groups 'B1', 'B2', 'B3', etc.

As mentioned earlier, the total number of faults seeded into the component that was inspected was 16. The table (Table 18) reports the number of faults found by every group within a team. For example, in team A (experimental team), group A3 found 12 faults out of the 16 seeded. Team B (research team) group B6 found 3 out of 16 faults seeded, etc.

Table 18. Validation Results and Analysis.

Teams	**Experimental Team**						**Control Team**						
Groups	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**
# of faults found	10	6	12	6	9	5	9	8	1	1	8	3	1
Total # of faults seeded	16	16	16	16	16	16	16	16	16	16	16	16	16
Average	**8**						**4.428571**						
Standard Deviation	**2.75681**						**3.735289**						
Standard error of mean	**1.125463**						**1.411806**						

The student's t-test was used for statistical analysis of the results. The null hypothesis (H0) is that the number of faults found will not vary between the experimental and control team. The alternative hypothesis (HA) is that a significant difference in the number of faults found will exist between the experimental and control group. We will reject the H0 in favor of HA when the probability that the observed results are due to chance is 0.1 or less. The use of alpha=0.1 is appropriate for our small sample size. The averages, standard deviations, and the standard error of means were calculated for both the teams and are shown in the table. The p-value was calculated to be 0.079809 for the results obtained.

The p-value obtained from our statistical analysis, indicate that the results obtained were statistically significant (i.e., there is a significant difference in the number of faults found between the teams). The obtained results indicate a consistent performance from the experimental team. In order to perform a statistical analysis with alpha=0.05, we need a large number of groups and therefore a large number of qualified inspectors. In order to view and compare the performance consistency of the experimental and control groups, we constructed a graph with number of faults found versus the groups. Graph A depicts the results from the code inspection. From the graph, on a average, we can see that the performance of the groups within the experimental team is far more consistent and better than the performance of the groups within the control team and the statistical significance of the results based on our hypothesis. Thereby suggesting that our results promise to be helpful in producing a quality software product.

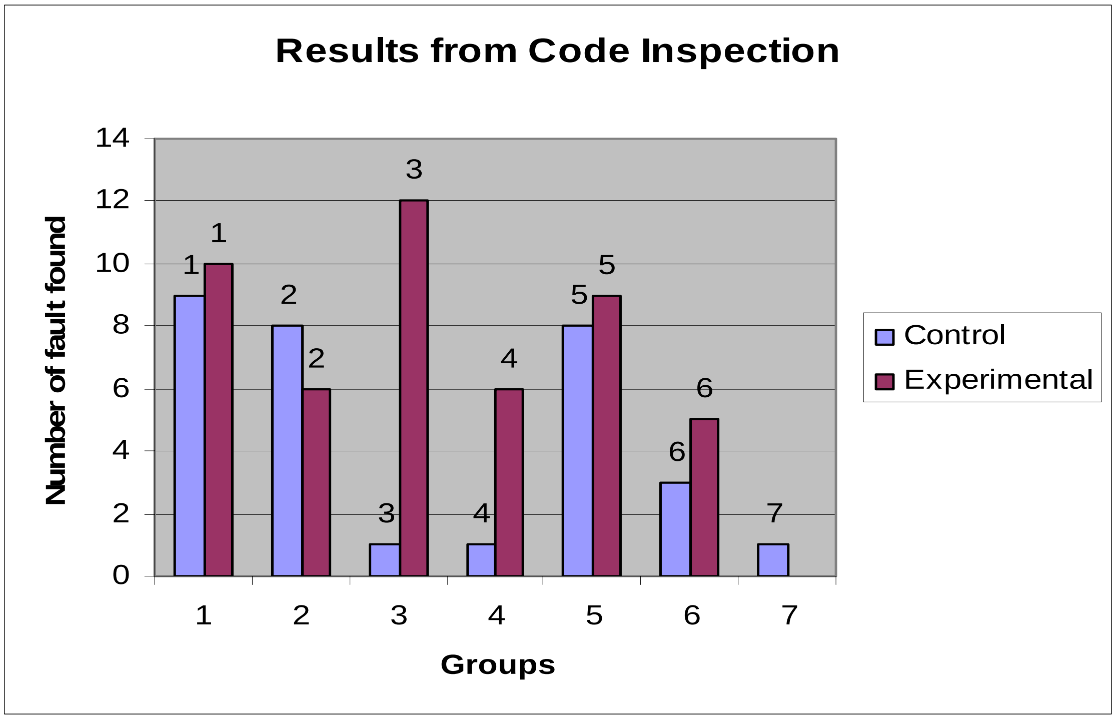


Figure 5a. Code Inspection Results.

We extended our analyses to determine the teams performance on difficult to find faults. We assigned an attribute to the faults based on our experience with the Java language (used in EPOCH), with code inspection process, and with software testing. The attribute indicated whether a particular fault should be easy, medium or hard to find, i.e. the difficulty in finding a fault can be either easy, or medium or hard. "Easy" means that the fault should be easy to find, "medium" means that the fault should not be either easy or hard to find, and "hard" means that the fault will be very difficult to find. Hard fault requires longer time to be found and cannot be found using regular knowledge about the component being inspected. Finding hard faults requires knowledge of fault links. Medium fault requires just more time to be found. The fault report sheet used by the subjects during the experiment also had columns that allowed them to indicate the level of difficulty in finding the faults. At the beginning of the experiment we decided on the difficulty in finding for each of the 16 faults based on our own experiences. According to our initial attribute value assignment (i.e., fault categorization based on difficulty in finding), we had 7 faults categorized as hard faults, 3 as medium and 6 as easy faults. But, the information gathered from the fault report sheets aided in validating and reconsidering our attribute value assignment. Finally, out of the 16 faults that were seeded into our component for inspection, we finalized that 11 of them were hard to find faults, 1 of them were medium, and 4 of them were easy.

Table 19. Team Performance on Hard to Find Faults.

Teams	**Experimental Team**						**Control Team**						
Groups	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**
# of faults found	6	4	8	4	5	4	5	5	0	1	5	2	1
Total # of faults seeded	11	11	11	11	11	11	11	11	11	11	11	11	11
Average	**5.166667**						**2.571429**						
Standard Deviation	**1.602082**						**2.370453**						
Standard error of mean	**0.654047**						**0.895947**						

The table (Table 19) reports the number of hard faults found by every group within a team. For example, in team A (experimental team), group A1 found 6 faults out of the 11 seeded, in team B (research team) group B4 found 1 out of 11 fault seeded, etc.

The student's t-test was used for statistical analysis of the results. The null and alternate hypotheses are the same as before expect that here we are dealing with only hard faults. The t-test revealed significantly better performance on the part of the experimental team compared to the control team ($P < 0.1$). The p-value was calculated to be 0.040198 for the results obtained. The ability of the experimental team to detect the hard to find faults was

2-fold greater than the control team. This further emphasizes the usefulness of the fault links identified using our methodology.

In order to view and compare the performance consistency of the experimental and control groups on hard to find faults, we constructed a graph with the number of hard faults found versus the groups. Graph B depicts the results from the code inspection.

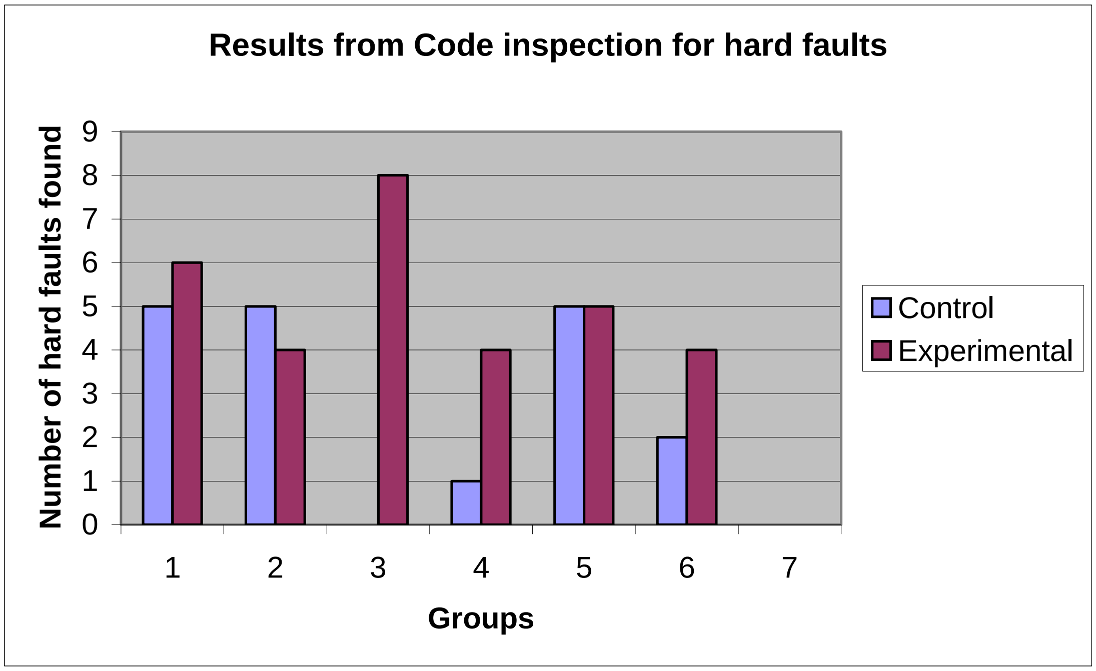


Figure 5b. Code Inspection Results for Hard Faults.

Copyright © Inies Raphael Chemmannoor Michael 2004

Chapter Five

Related Work

In this chapter, we present the results of a comprehensive literature survey of existing relevant research areas. More specifically, we focus on that research aiding in the improvement of the quality and reliability of a software system using approaches with software components and/or faults as important sources. At the end of this section, we discuss the uniqueness and usefulness of our research ideas and approach, compared to the existing approaches. We present our survey results in three categories: Fault surveys (involve research that only uses defects, faults, or errors), Component/module surveys (involve research that only uses software components), and both component and fault surveys (involve research that use both).

Fault Surveys

Faults have traditionally been characterized by syntactic categories [5, 41, 31], including the position in the program where faults occur [29], which software development phase generated the faults [45, 36], what testing phase found the faults [54], and what type of statement or language feature in which the faults occur [23]. As part of a National Aeronautics and Space Administration (NASA) funded project, Hayes [27] developed requirements faults taxonomy.

Hayes [27] presents a methodology for requirement fault-based analysis and its application to NASA. This fault-based analysis technique provides guidelines to prevent and/or detect different classes of requirement faults prior to implementation. Requirement faults from six large NASA industrial systems were examined to build NASA-specific requirement fault taxonomy. Processes to tailor the taxonomy to a class-specific or a project-specific taxonomy were presented. The study concentrates on requirement faults as opposed to our current study that adopts the processes and ideas discussed in [27] to focus on code-based faults and to identify the relationships between component and fault types.

IBM's Orthogonal Defect Classification [30] attempts to classify faults based on the mental mistakes that programmers make by assigning mental mistakes as part of a larger classification scheme.

Endres [21], tries to identify and analyze the different types of errors and possible causes for their occurrence in order to improve software reliability. Endres concentrates only on system programs. The system program chosen for his study was the operating system DOS/VS, developed in the IBM Boeblingen laboratory. The investigators categorize the errors into 3 main groups: Group A, Group B, and Group C. Group A deals with errors in the understanding of the problem and in the choice of an algorithm to solve it. Group B deal with errors that are specific to the implementation process. Group C deals with non-programming errors in the strict sense. However, the errors identified by Endres were confined to system programs and were not extended to form a generic classification to suit all types of software application. Moreover, the errors categorized involve only design errors but do not involve implementation and code errors as is the case in our research.

Marick [41] presents the results gathered from an exhaustive software fault survey. He has presented the software fault classifications adopted by several researchers in their respective studies. These classifications encompass the faults that can originate in almost all stages of the software development life cycle, viz., requirements, design, implementation, and testing. Marick also introduced two broader classifications of fault categories: faults by omission (i.e., faults that were caused by failing to do something) and faults by commission (i.e., faults that were caused by doing incorrect things). We, in our research, tried to include these concepts as attributes to the faults, but we failed to obtain any real application from them so far. However, we feel that the code-fault taxonomy discussed in this paper is not as exhaustive and generic as our code-fault taxonomy. Also, the main focus of Marick's paper was only to survey faults, therefore, no considerable efforts were made to use the knowledge gained from the survey to improve software reliability or quality.

Shooman and Bolsky [63] present the results obtained from an experiment that was conducted in order to collect some basic information on software errors. The main objectives of the experiments were to develop and utilize a set of terms for describing possible types of errors, their nature and frequency, to perform a pilot study to determine if data of the type reported can be collected, to investigate the error density (error density of a module is denoted as the percentage of the module's total number of lines of code), and to develop data on how to use the available resources in debugging. Shooman and Bolsky reported that a large percentage of errors were found by hand processing (without the aid of computer testing techniques). However, no effort was made to categorize the errors and the authors are not definitive about the results presented.

Lutz [39] analyzed the root causes (i.e., requirement faults) that lead to safety-related software errors (program errors found during integration and testing) in a safety critical, embedded system. Lutz's main goal was to reduce safety-related software errors and to enhance the safety of complex, embedded systems. In order to achieve her goal, Lutz tried to analyze the root cause of the software errors by adopting the classification scheme proposed by Nakajo and Kumis [50]. The classification scheme traces backward in time from the evident software error to an analysis of the root cause. However, Lutz presents only a high level classification for both program and requirement faults with the classification scheme working only with embedded systems.

In an effort to generate effective test cases, to detect errors and thereby produce quality software, two complementary studies on specification-based test generation methods were conducted. Kuhn [35] presents proofs that faults in Boolean specifications constitute a hierarchy with respect to detectability and further concluded that missing condition faults should be hypothesized to generate effective tests. Tsuchiya et al., in their paper [65], feel that the conclusion drawn by Kuhn is premature and tried to investigate the relationship between missing condition faults and faults in other classes. They complemented Kuhn by showing that missing condition faults need not be hypothesized to generate effective tests. However, the fault classes presented by both these studies are not exhaustive and no efforts were made to apply the results obtained to improve software quality.

Chillerage et al. [13], in the aim of improving the software development process and thereby software quality, described a concept called Orthogonal Defect Classification (ODC). ODC enables in-process feedback to developers by extracting special features on the development from defects. The paper illustrates the use of the defect type distribution to measure progress of a product through a process and also demonstrates the use of the defect trigger distribution to evaluate effectiveness and completeness of the verification process such as inspection or testing. However, this paper presents only a high level classification of software defects with no effort to establish fault links.

Fenton and Ohlsson [22] have quantitatively analyzed the faults and failures of a major commercial system. Some of their observations were identical to those made by Ostrand and coworkers [56]. Furthermore, Fenton et al provided strong evidence to suggest that software systems that are developed under the same environment result in similar fault densities, when tested in similar testing phases. Hamdioui et al. [26], in an effort to aid test engineers to deal with new dynamic-fault classes, tried to mathematically analyze these fault classes based on the fault primitive concepts. The study emphasizes dynamic memory related faults. But, in our research, we only deal with static run time faults that can impede the performance of the software product.

Briand et al. [10] evaluated the capture-recapture models that are used to predict the number of remaining defects in an inspected software artifact that can aid in decision making. According to Briand, the decisions based on objective information, such as whether the inspection can stop or whether it should continue to achieve a suitable level of quality, are significant to control the inspection of software artifacts. However, the study only highlighted the analysis of numerous capture-recapture models to improve inspection and thereby improve software quality and no attempts were made to use software defects knowledge to improve software quality.

Nakagawa and Hanata [49] describe a software reliability model, called the error complexity model, to measure the reliability of the software. The model estimates software reliability with the ratio of complex to simple errors. According to the model, errors are classified by error complexity, which is a measure of error detectability.

Nakagawa et al. also proposed new criteria for error complexity classification. They classified error complexity into three classes: static, conditional, and composite without classifying errors per se.

Agresti and Evanco [1], with an aim to improve software quality, describe various models for projecting software defects by analyses of Ada design. The models predict defect density based on product and process characteristics. However, no effort was made to classify software defects.

Dehlinger and Lutz [17] introduced a technique called product line software fault tree analysis to improve software quality. A product line is a set of systems that are developed from a common set of core requirements and share a suite of common traits among the members. Software Fault Tree Analysis (SFTA) [17] is a technique that has been successfully used to investigate causes contributing to potential hazards in safety-critical applications. The work investigates an adaptation of the SFTA technique to product lines in order to derive reusable analysis assets for future systems within the existing product line. Specifically, the paper focused on how and to what extent the product line SFTA can be used by software engineers as a reusable safety analysis asset. However, the application of the method discussed was not evident for software implementation.

Marick [42] presented a hypothesis based on fault-adequate testing called weak mutation hypothesis and attempted to evaluate the same. In fault-adequate testing, a fault is said to be detected if a test case satisfies three conditions: reachability, necessity, and sufficiency. According to the weak mutation hypothesis, test cases that satisfy the reachability and necessity condition will satisfy the sufficiency condition. For the purpose of hypothesis verification, Marick studied 100 faults gathered from five sizable and widely used programs and found that the hypothesis holds true for 60 of them. Based on these experiments, the authors concluded that the combination of specification-based testing and weak mutation testing will discover 90% of the faults that strong mutation testing would discover. However, the authors did not categorize the faults.

Offutt and Alexander [53] studied the characteristics of the program faults that occur in object-oriented software in order to improve the available object-oriented testing techniques. They believe that a full understanding of the characteristics of faults is crucial to several research areas. The paper presented a model for the appearance and realization of object-oriented faults and defined specific categories of inheritance and polymorphic faults. According to the authors, the models and categories presented can be used to support future empirical investigations on object-oriented testing techniques, to inspire object-oriented testing and analysis research, and to help improve object-oriented software design and development process. However, the fault categories presented list faults that are related only to object-oriented software and, moreover, they concentrate only on inheritance and polymorphic faults.

Munoz [48] presented an approach to software product testing. The method used numerous techniques. The technique that is relevant to our research (i.e. dealing with defects) is defect circumvention (i.e., correction of defects in the test cases instead of in the product). Software testing tasks aiding defect circumvention are defect detection, isolation, and identification. However, the drawbacks of this approach were that the study did not categorize the defects and did not address product defects.

Offutt and Hayes [54], in an effort to analyze the characteristics of program faults, proposed a semantic model for fault categorization based on the syntactic and semantic size of the fault. They believe that viewing faults through this model characterization can solve most of the problems faced by fault-based testing techniques. The authors are hoping that the model presented will lead to new insights in testing and might even foster new research into the discovery and use of faults.

Xie and Engler [68] illustrated the seriousness as well as the usefulness of redundant errors. They believe that redundant errors are as serious as other errors (termed as hard errors). Thus, in order to experimentally verify and prove their hypothesis, they developed and applied five redundant checkers on large open source projects. The open source projects used were: Linux, OpenBSD, and PostgreSQL. They also showed the usefulness of redundant errors in finding mistakes and omissions in specifications. Although the study discovered new fault types, it was not as exhaustive and generic as the one presented in our research.

Dunsmore et al. [19], in an effort to improve the effectiveness of the object-oriented code inspection process, developed three techniques: one based on checklist, another on constructing an abstract specification, and the last based on the route taken by a use case through the program. The techniques discussed address three significant issues: (i) the identification of chunks of code to be inspected, (ii) the order in which the code is read, and (iii) the resolution of frequent nonlocal references. Among the three techniques discussed, the checklist technique proved to be the most effective when compared to the other two. The authors suggest that, for any practical situation, a combination of techniques is always useful. However, the study focused only on object-oriented code inspection enhancement. Although the study eventually aimed at improving software quality the approach presented is different from the one presented in our research.

Dalal et al. [15], in order to improve the software development process and thereby software reliability, examined the software development process and suggested areas for process improvement by using a combination of statistical and other process control techniques. This research in the event of fulfilling its goal, presented a high level fault classification along with its severity levels (serious, moderate, or minor). However, the faults classified do not entirely focus on software code faults.

Component/Module Surveys

Khoshgoftaar and Allen [32, 33, 34] classified a software module based on its quality either as a fault prone module or as a non-fault prone module. In [32], they demonstrate how module-order models can be used for classification, and compare them with statistical classification models, discussed in [33]. In [34], they attempt to control the overfitting problem that causes the classification models [33] to miscalculate the fault-proneness of a component. However, no effort was made to classify the faults and the module classification presented was a more superficial classification than to our component classification. We present two methods to classify software components, one based on the percentage of lines of code that perform a specific function, and the other based on the component description. In this research, we make use of the latter method to classify components.

Damiani et al. [16] presented a hierarchy-aware classification schema for object-oriented code, where a software component is classified based on its behavioral characteristics such as service provided, algorithm employed, and data needed. These characteristics can either be constructed from the application models or can be extracted semi-automatically from the class interfaces. Damiani et al. name the set of characteristics associated with a component as its software descriptor. The classification of the components was supported by a thesaurus acting as a language-independent unified lexicon. However, the classification method presented can only be used for object-oriented software projects. In our research, we present generic methods that can be applied to both procedural and object-oriented software projects. In our methods for component classification, we take into consideration only the behavioral aspect of the component or the lines of code, but not other factors like algorithms used, required data, etc. as highlighted in [16]. Nevertheless, our research addresses issues beyond component classification.

Long and Hoffman [38] presented a method and support tool for testing concurrent Java components. The support tool is offered through Concurrency Analyzer, to generate drivers for unit testing Java classes that are used in a multithreaded context. On lines similar to our research, Long et al. also considered a single Java class to be unit or a component. However, they neither try to classify the components and faults nor try to identify the relationships between them, but only concentrate on testing concurrent Java components. The results obtained from our research may not be completely useful for testing concurrent software components and involves static analysis of the software project code.

Briand and Basili [9] presented the optimized set reduction approach for constructing models that can classify software components as either high-risk or low-risk components. According to Briand et al, one needs to be able to differentiate low/high fault frequency components so that testing/verification efforts can be spent where needed. This strategy will not only improve software quality but also guarantee efficient utilization of available resources. In their approach to classify, they measured the software system and built multivariate stochastic models for predicting high-risk components. However, as one can see, the component classification discussed was more at the higher level.

According to Cardelli [11], for a software system to satisfy or reach a level of quality, its modules (assuming the system is modularized) need to be compiled, linked and tested independently. He states that although various module mechanisms have received considerable theoretical attention, the associated concepts of separate compilation and linking have not received sufficient emphasis and moreover, software components are not separately type-checkable and compilable. In his paper [11], Cardelli presented a framework where each module was separately compilable to a self-contained entity called a linkset, and he also showed how separately compiled modules could be linked together. However, they did not attempt to classify the software modules and to further study the existence of fault links.

Zaremski and Wing [70] presented a method to compare two software components based on their behavioral descriptions. The method is called Specification Matching. They use formal specifications to describe the component behavior and hence determined whether two components match. The applications of the method are two fold: First, in the context of software reuse and library retrieval, it can help to determine whether one component can be substituted for another or how one can be modified to fit the requirements of the other. Second, in the context of object-oriented programming, it can help to determine when one type is a behavioral subtype of another. However, no effort was made to classify components to improve software quality.

Lew et al. [37] presented a software complexity metric that included both the internal and external complexity of a module. The authors believe that software complexity directly affects the reliability of the software, and hence, there is a need to decompose a software system into modules to control complexity and produce reliable software. Lew has shown that the complexity metric presented will be useful in quantifying the design of the software and provides a guide to system decomposition. However, the investigators did not present any module classification.

According to Eisenbarth and Koschke [20], for one to exhibit full understanding of a program, one has to locate and understand certain features (the term feature, according to the authors, means a realized functional requirement of a system) that are exhibited by the program code. The paper presents a semiautomatic technique that constructs the mapping between the feature and the computational unit. The authors believe that this mapping is not injective in general, i.e., a computational unit may contribute to more than one feature. According to the paper, a computational unit is defined as an executable part of a system, for example, basic blocks, routine, etc. However, no efforts were made to categorize these units.

Large software systems during maintenance undergo continuous modification and considerable increase in size, complexity, and behavior. Gal et al. [24] believe that in order to determine the impact caused by these changes, one needs to understand the dependencies that exist between modules that compose the system. According to them, current existing code-based measures (cohesion and coupling) only reveal the syntactic dependencies, but do not determine the logical dependencies between them. The logical dependencies are also necessary to estimate the impact. Therefore, Gal and his team, present an approach to uncover logical dependencies and change patterns of modules using information in a release history of the system. In order to develop this approach the authors have worked with 20 releases of a large telecommunication switching system. However, the work does not discriminate between corrective maintenance and enhancement of related changes, thereby not classifying faults. Furthermore, software modules were not categorized that can aid effective maintenance.

Similar to Gal, Bieman et al. [6] identified change-proneness of C++ code based on intentional use of patterns (or lack thereof). During this analysis, he found that some patterns are more change-prone in different categories of maintenance (corrective versus enhancement related changes). However, no attempt was made to classify these faults. Bieman et al. [8] also found a strong relationship between class size and the number of changes; larger classes changed more frequently. Additionally, classes that participate in design patterns are more change-prone, and classes that are reused through inheritance are more change-prone. But the investigators did not identify the type of change or fault in these studies.

Component and Fault Surveys

Basili and Perricone [4] tried to analyze the relationships between the frequency and distribution of errors during software development, the maintenance of the developed software, and a variety of environmental factors (such as, complexity of software, developers experience with the application, and the reuse of existing design and code). They believe that these relationships can not only improve the reliability and quality of the software, but also provide an insight into the characteristics of computer software and the effects that an environment can have on the software product. The paper defined a module as a named sub function, subroutine or the main program of the software system. They classified a module to be either modified (i.e., modules that were developed for previous software projects and then modified to meet the requirements of the new project) or as new (i.e., modules that were developed specifically for the software project under analysis). However the module classification is a high level classification when compared to the one presented in this paper. The authors have also classified software errors into five different categories. We have made use of some of these categories and definitions.

Ohlsson et al. [55] modeled fault proneness statistically over a series of releases. This included a variety of 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 involved more than one module, and impact of change measures (how many files affected, how many changes for each, various size of change measures by file type). The analysis of the case study data showed that fault-prone modules exhibit higher system impact across four releases, where system impact is defined as total number of changes to .c and .h files in a release per module. This motivated construction of a fault architecture [22], which determines fault coupling and cohesion measures at the module and subsystem levels, within a release and across releases. Nikora and Munson presented a predictor for fault prone modules. They used a set of metrics and a reduced set of domains to build their predictor. They did not classify faults though and did not classify modules beyond being "fault prone" or not "fault prone [51]."

Mayrhauser et al. [44], in an effort to aid efficient software maintenance, presented methods to eliminate software architecture problems. They believe that such problems are very expensive to fix and would be desirable to track them down early and across multiple releases. The paper developed measures and methods to build fault architectures from existing defect reports, define measures to rank the most fault-prone relationships between components and subsystems in a number of releases, and finally, develop a fault component directory structure to investigate the fault-prone relationships. Mayrhauser et al. used a large commercial system consisting of over 800KLOC of C, C++, and microcode to illustrate their technique. However, the component categorization – fault-prone and not fault-prone – is very high level and, moreover, no efforts are made to categorize the faults.

Ostrand and Weyuker [56], with the aim of aiding organizations to determine the optimal use of their testing resources, have identified various file characteristics. These characteristics can serve as predictors of fault-proneness. By employing a series of 13 releases of a large evolving industrial software system, they observed that: (i) faults are concentrated in a small numbers of files and in a small percentage of the code mass, (ii) shortchange to the 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"[56]. However, no effort was made to classify modules and faults.

From the above survey, it is clearly evident that researchers around the world have undertaken numerous efforts to come up with various methods and/or techniques to improve the quality and/or reliability of the software using faults or problem reports. To summarize these studies, researchers aiming to provide guidance and help to software engineers to produce quality software products have tried to identify fault predictors, performed quantitative analysis of faults, developed models to measure the reliability of software, developed defect classification and schema, suggested methods to identify the root cause of faults and to generate effective test cases, indicated areas of improvement in the software development process, presented component classification and schema, identified methods to classify components and to compare them, and finally, have presented software complexity metrics.

However, the method and suggestions presented in the past are not generic, i.e., they are unique to a particular type of software application or domain. Moreover, the classifications and the classification schemas presented either do not focus on software run time code-related faults or the fault categories discussed are more of a high level classification than what is actually required. In our research, we introduced the concept of fault links (relationships between the types of code-faults and the type of component being developed or modified) that provided guidelines to software engineers during every phase of the development life cycle to ensure an effective development process, and thereby, produce a high-quality software product. We adopted a three-phase process to obtain our results. First, we developed a generic component and code-fault taxonomy, which can be applied to any type of software application. Second, we adapted processes from [27], to identify faults and components that are unique to a particular project under a particular domain, and finally, we developed a process to establish (or identify) the fault link relationship (if exists) between a component type and the fault types. The processes presented in this paper are generic, i.e., they can be applied to any project type that belongs to any domain. The results obtained from our processes can be used in different phases of the life cycle to aid in quality software development processes.

Copyright © Inies Raphael Chemmannoor Michael 2004

Chapter Six

Conclusions and Future Work

We have developed two taxonomies one for components and one for code faults. We presented two methods for component classification along with their advantages and disadvantages. We presented two processes: Domain-process and Project-process, to tailor or extend both the taxonomies into domain-specific and project-specific taxonomies, respectively. We classified modules and code faults of two online course management products (EPOCH and ILIAS) using our approach. We also presented a process (Component-process) to identify fault links. The results of these processes were presented and discussed. We selected the EPOCH project and applied Component-process to identify the existing fault links. We found evidence in favor of the existence of four conjectured fault links (and an additional two with weak evidence) and six fault links that were not conjectured (all related to Control/Logic faults). We have already capitalized upon the discovery of the Control/Logic fault links (for every module type) by augmenting our FTR checklists. Unfortunately, due to lack of data we were not able to verify the existence of 7 fault links that were conjectured. From the results, we found the need for more projects with sufficient data under a chosen domain and also the need for well-qualified and experienced software engineers to carry out the experiments.

We conducted an experiment to verify the usefulness of identifying fault links. The results from this experiment were discussed and analyzed using statistical methods. The analysis confirmed the usefulness of fault links in the process of code inspection or walkthrough. Although, we strongly believe in the application of fault links over different stages of the software development life cycle, we did not perform any experiments, due to limited resources, to verify the same.

We are still continuing to work on the fault taxonomy and the component taxonomy and hope that others will assist us in validating and improving them. We have examined the taxonomies with respect to the object-oriented methodology. We plan to examine languages such as Lisp that provide control abstraction. We are also convinced that the taxonomies are not 100% orthogonal. Evaluating this aspect of the taxonomy represents an area of future work.

We are still working on the processes to identify areas of improvement and methods to implement them. We have so far identified that some of the process metrics are not really useful and such metrics need to be eliminated. However, we also believe that the usefulness of the metrics depends wholly on the interests and priorities of the organization using the process.

We conducted an experiment to verify the usefulness of fault links to aid software testers. We need to conduct more experiments to verify the following hypotheses.

H15 - The results will be useful for Developers, in developing similar projects of the same domain.

H16 - The results will be useful for Requirement Engineers, in developing requirement specifications for similar projects of the same domain.

H17 - The results will be useful for Designers, in designing similar projects of the same domain.

We believe that the results provided in this paper will be useful for software engineers' especially software developers and testers who are working on online course management or any web-based software product. The component taxonomy developed will be useful to software maintainers, to organize large software products by grouping its component into various component categories.

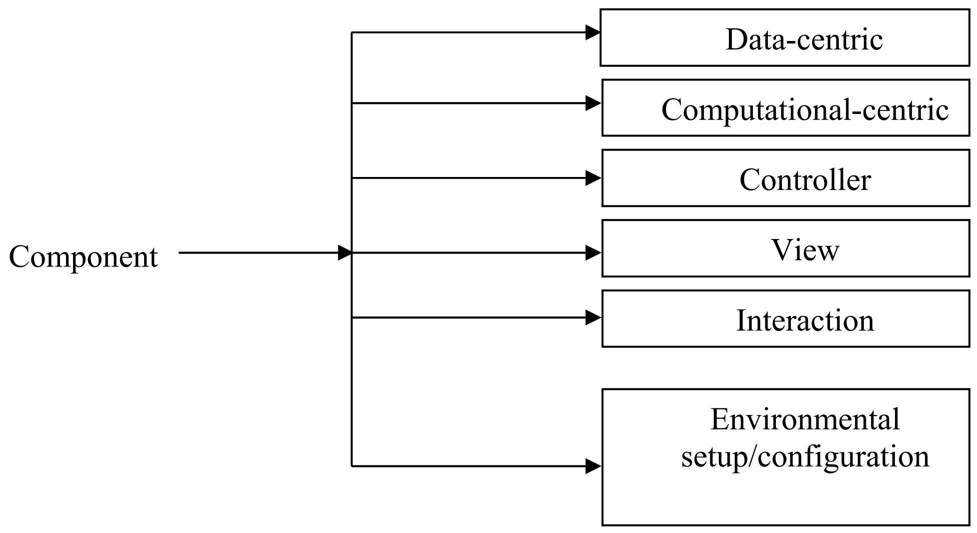
The main direction for future work is the expansion of the fault link idea into a study of fault chains. Faults rarely occur in isolation. They may be related longitudinally within a release (e.g., a design fault leads to a code fault) or across releases (e.g., incomplete fault repair). We refer to these relationships as fault chains. We have identified several types of fault chains, and will continue our work in this area. It should be noted that a larger scale study with a variety of industry projects across diverse domains is required before any broad conclusions can be reached.

The ultimate goal of this work is to identify V&V techniques or quality assurance activities that can take advantage of our knowledge of fault chains to prevent or detect faults as early as possible. That will assist us in developing reliable, software systems.

Copyright © Inies Raphael Chemmannoor Michael 2004

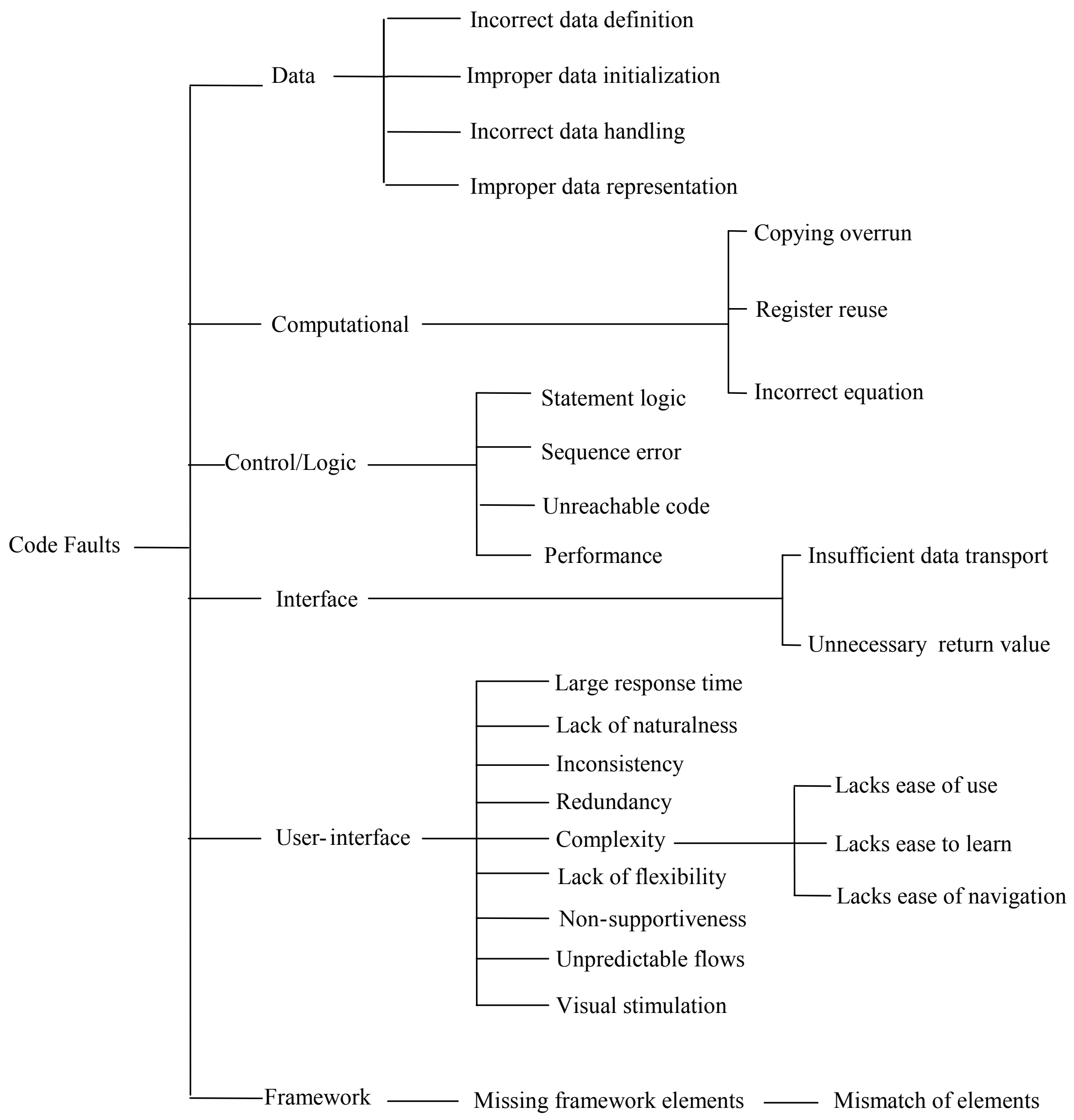
APPENDIX A

Generic Component Taxonomy



APPENDIX B

Fault Taxonomy



APPENDIX C

Component Description

Name of the component: GradeStore.java
Purpose: To read student grades.

The grades for homework performed by each student are stored in the table called RESPONSE. The table also keeps track of the scores obtained by each student for individual problems in the homework. Besides grades for the problems in a homework set, the table also contains fields that store the status of the answer given by each student (status), number of attempts made by the student (tries), feedback to the student based on his response (feedback), and the response of the student (response) for every problem in the set.

The status field takes in a value ‘C’ when the answer provided by the student for a particular problem in a particular homework set is correct, ‘P’ when the answer is partially correct, and ‘W’ when it is wrong.

The component has two functions with each function serving a specific purpose. The functions are listed below.

1. getStudentSumGrades()

   Input parameters:

   a. APPConfig conf - contains application configuration details

   b. int hwIds[]   - integer array that contains the IDs of all the

      homework (each homework is assigned a unique ID)

The function reads in the grades for the homework by each student. A minus one (-1) for the grades indicates that the student has not attempted that particular homework. The function examines all the homework assigned to each student individually. It then returns the grades obtained by each student for all the homework assignments. The function stores the grades imported from the RESPONSE table in an array called sums, where its index indicates the homework number in the request sequence (input array hwids). For example, sums[0] indicates the grade for the first homework in the request sequence, sums[1] indicates the grade for the second homework in the sequence, and so on.

Also, the function returns a hashtable that contains student ID and the array sums as its fields. Thus each student identified by his ID will have a separate array containing his grades for the homework assigned to him.

2. getResults ()

Input parameters:

a. APPConfig conf - contains application configuration details

b. int hwId - ID for a particular homework

This function can read in the results of all problems attempted by every student for the given homework, the latter being identified by its ID. It employs the same RESPONSE table as mentioned above. The information read in for each problem in the given set or homework is stored in a structure called Result. This Result structure for each problem is further stored as an object into an array called resArr. In short, the resArr array holds all the objects for the given set. For example, resArr[0] contains the object that has information about problem #1 in the given set.

The function returns a hashtable that contains student ID and the result array as its fields. Therefore, each student identified by his ID will have a separate result array, which stores information as previously described.

APPENDIX D

Generic Code Inspection Checklist

Group ID:________________________________

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)

APPENDIX E

Experimental Code Inspection Checklist

Group ID:______________________________

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)

Inspector #1 signature: ____________________________ Date: ______________

Inspector #2 signature: ____________________________ Date: ______________

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 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. Faults manifested due to errors in these statements fall under this category.

APPENDIX F

Fault Report Sheet

Line #	Fault Description	Difficulty in finding faults [check (√)the appropriate option]		
		Easy	Medium	Hard

APPENDIX G

Questionnaire

1. Was the component description easy to understand?

2. How much prior Java experience do you have?

APPENDIX H

Survey Sheet

1. What is your opinion about the experiment?

2. Any suggestions on how to improve the experiment?

3. Any feedback on the documents provided to you?

4. Any suggestions to improve the documents?

5. Do you think this code walkthrough session will be a useful experience?

6. Do you have any work experience?

7. How much experience do you have doing walkthroughs?

REFERENCES

[1] Agresti, W.W., Evanco, W.M. Projecting software defects from analyzing Ada designs. IEEE Transactions on Software Engineering, vol. 18, N0. 11, November 1992.

[2] Allen, M. and Yeh, W. *Introduction to Measurement Theory*. Brroks/Cole Publishing, 1979.

[3] Apache modules and problem reports, Apache HTTP server version 1.3.24, http://httpd.apache.org/docs/mod/index-bytype.html

[4] Basili, V.R. and Barry T. Perricone. ‘‘Software Errors and Complexity: An Empirical Investigation.’’ *Communications of the ACM*, 27, 1 (January 1984), 42-51.

[5] Beizer, B. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd Edition, ISBN 0-442-20672-0, 1990.

[6] Bieman, J., Andrews, A. and H. Yang. Analysis of change-proneness in software using patterns: a case study, submitted *Seventh European Conference on Software Maintenance and Reengineering* (Benevento, Italy, March 2003).

[7] Bieman,J., Jain, D., and H. Yang. Design patterns, design structure, and program changes: an industrial case study. *Proceedings of the International Conference on Software Maintenance* (Florence, Italy, 6 – 10 November 2001).

[8] Bieman, J., Straw. G., Wang. H., Mungar. P.W., and Alexander R.T. Design patterns and change proneness: an examination of five evolving systems.

[9] Briand, L.C., Basili, V.R., and Hetmanski, C.J. Developing interpretable models with optimized set reduction for identifying high-risk software components. IEEE Transactions on Software Engineering, vol. 19, No. 11, November 1993.

[10] Briand, L.C., El Emam, K., Freimut, B.G., and Laitenberger, O. A comprehensive evaluation of capture-recapture models for estimating software defect content. IEEE Transaction on Software Engineering, vol. 26, No. 6, pp. 518-540, June 200

[11] Cardelli, L. Program fragments, linking, and modularization. Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, p.266-277, January 15-17, 1997, Paris, France

[12] Centre of Software Maintenance, University of Durham, England. http://www.dur.ac.uk/computer.science/research/csm/rip/introduction.html

[13] Chillarege, R., Bhandafi, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., and Wong, M. Orthogonal defect classification A concept for in-process Measurement. IEEE Transactions on Software Engineering, vol. 18, No. 11, November 1992, pp. 943-956.

[14] Cooper, A. *About face: the essentials of user interface design.* IDG Books Worldwide, Foster City, CA, 1995.

[15] Dalal. S.R., Horgan. J.R., and Kettenring. J.R. Reliable software and communication: software quality, reliability, and safety. Proceedings of the 15th international conference on Software Engineering, p.425-435, May 17-21, 1993, Baltimore, Maryland, United States

[16] Damiani, E., Fugini, M.G., and Bellettini, C. A hierarchy-aware approach to faceted classification of object-oriented components. ACM Transactions on Software Engineering and Methodology, vol. 8, No. 3, July 1999, pp. 215-262.

[17] Dehlinger, J., and Lutz, R.R. Software fault tree analysis for product lines. Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE'04), March 25-26, 2004, Tampa , Florida.

[18] Duncan, IMM., and 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.

[19] Dunsmore. A., Roper. M., and Wood. M. The development and evaluation of three diverse technique for object-oriented code inspection. IEEE Transactions on Software Engineering, vol. 29, No.8, August 2003, pp. 677-686.

[20] Eisenbarth, T., Koschke, R., and Simon, D. Locating features in source code. IEEE Transactions on Software Engineering, vol. 29, No. 3, March 2003.

[21] 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, No. 6, pp. 327-336, June, 1975.

[22] Fenton N.E., and Ohlsson N. Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Transactions on Software Engineering, vol. 26, No. 8, August 2000, pp. 797-814.

[23] Freimut, B. "Developing and Using Defect Classification Schemes", Fraunhofer IESE *IESE-Report No. 072.01/E, Version 1.0*, September, 2001.

[24] Gall, H., Hajek, K., and M. Jazayeri. Detection of logical coupling based on product release history. *Procs. International Conference on Software Maintenance* (Bethesda, MD, November, 1998). IEEE Computer Society Press, 190-198.

[25] 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, USA, August 1995). Chapman & Hall, London, 1996, 293-304.

[26] Hamdioui. S., Gaydadjiev. G.N., van de Goor. Ad. J. A fault primitive based analysis of dynamic memory faults. In IEEE 14th Anual Workshop On Circuits, Systems and Signal Processing, pp. 84-89, Veldhoven, The Netherlands 2003.

[27] Hayes, J.H. "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, November 2003).

[28] Hayes, J.H., Dekhtyar, A., and J. Osbourne, "Improving Requirements Tracing via Information Retrieval," in *Proceedings of the International Conference on Requirements Engineering* (Monterey, California, September 2003).

[29] Hayes, J.H., Mohamed, N., and T. Gao, "The Observe-Mine-Adopt Model: An Agile Way to Enhance Software Maintainability", *Journal of Software Maintenance and Evolution: Research and Practice*, *15, 5* (October 2003), 297 – 323.

[30] IBM Research, Center for Software Engineering, "Details of ODC v5.11", http://www.research.ibm.com/softeng/ODC/DETODC.HTM

[31] IEEE Standard Classification for Software Anomalies, December 12, 1995. IEEE Std 1044.1-1995.

[32] Khoshgoftaar. T.M., and Allen. E.B. A comparative study of ordering and classification of fault-prone software modules. Empirical Software Engineering, 4, 159-186, 1999.

[33] Khoshgoftaar. T.M., and Allen. E.B. Classification of fault-prone software modules: prior probabilities, costs, and model evaluation. Empirical Software Engineering, 3, 275-298, 1998.

[34] Khoshgoshtaar. T.M., and Allen. E.B. Controlling overfitting in classification-tree models of software quality. Empirical Software Engineering, 6, 59-79, 2001.

[35] Kuhn, D. R., Fault classes and error detection capability of specification based testing. ACM Transactions on Software Engineering Methodology, vol. 8, No. 4, October, pp. 411-424.

[36] Lanubile, F., Shull, F., and V.R. Basili, "Experimenting with Error Abstraction in Requirements Documents", *Proceedings of the 5th Inernational. Symposium on Software Metrics* (Bethesda, Maryland, 1998).

[37] Lew, K.S., Dillon, T.S., and Forward, K.E. Software complexity and its impact on software reliability. IEEE Transactions on Software Engineering, vol. 14, No. 11, November 1988.

[38] Long, B., Hoffman, D., and Strooper, P. Tool support for testing concurrent Java components. IEEE Transactions on Software Engineering, vol. 29, No. 6, June 2003.

[39] Lutz, R.R. Analyzing software requirements errors in safety-critical, embedded systems. Re '93, the Proceedings of the IEEE International Symposium on Requirements Engineering, Jan 4-6, 193, San Diego, CA.

[40] Macaulay, L. ***Human -computer interaction for software designers.*** International Thomson Computer Press, London, 1995.

[41] Marick, B. A survey of software fault surveys. A technical report UIUCDCS-R-90-1651, University of Illinois, 1990; pp 2-23.

[42] Marick, B. The weak mutation hypothesis, Proceedings of the symposium on Testing, analysis, and verification, p.190-199, October 08-10, 1991, Victoria, British Columbia, Canada.

[43] Mayhew, DJ. *Principles and guidelines in software user interface design***.** Englewood Cliffs, N.J. Prentice Hall, 1992.

[44] Mayrhauser, A., Ohlsson, MC., and Wohlin, C.**:** Deriving fault architecture from defect history. *J. Softw. Maint. Res. Pract., 12*, (2000), 287-304.

[45] Miller, LA., Groundwater, EH., Hayes, J., and Mirsky, SM.**:** Guidelines for the verification and validation of expert system software and conventional software. SAIC 1995; 2**:** pp 100.

[46] Mozilla organization website, http://mozilla.org/

[47] Munch, J, Rombach, H.D., 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, Oregon, USA, May 3-4, 2003).

[48] Munoz, C.U. An approach to software product testing. IEEE Transactions on Software Engineering, vol. 14, Np. 11, November 1988.

[49] Nakagawa Yutaka, and Hanata Shuetsu. An Error Complexity Model for Software Reliability Measurement. Proceedings of the 11th International Conference on Software Engineering (1989),

[50] Nakajo, T., and Kumis, H., A case history analysis of software error causes-effect relationships. IEEE Transactions on Software Engineering, vol. 17, No. 8, August 1991, pp. 830-838.

[51] Nikora, A., and Munson, J. Developing Fault Predictors for Evolving Software Systems. Proceedings of the Ninth International Software Metrics Symposium (METRICS 2003) (Sydney, Australia, September 2003).

[52] Offutt, J. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering Methodology*, *1, 1* (January 1992), 3-18.

[53] Offut, J., and Alexander, R. A fault model for subtype inheritance and polymorphism. 12th IEEE International Symposium on Software Reliability Engineering (ISSRE '01), pp. 84-95, Hong Kong, PRC, November 2001.

[54] Offutt, J., and J. H. Hayes. A Semantic Model of Program Faults. *International Symposium on Software Testing and Analysis (ISSTA 96)* (San Diego, CA, January 1996).Ohlsson, M., Andrews, A., and C. Wohlin. Modelling fault-proneness statistically over a sequence of releases: a case study. *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, *13*, (June 2001), pp. 167--199.

[55] Ohlsson, M., Andrews, A., and C. Wohlin. Modelling fault-proneness statistically over a sequence of releases: a case study. *Journal of Software Maintenance and Evolution: Research and Practice*, Volume 13, June 2001, pp. 167--199.

[56] Ostrand, T.J., and Weyuker, E.J. The Distribution of Faults in a Large Industrial Software System. Proc. ISSTA02 & ACM SIGSOFT, vol. 27, No. 4, July 2002, pp. 55-64.

[57] Perry, D.E., and C.S. Stieg, "Software Faults in Evolving a Large, Real-Time System: a Case Study", AT&T Bell Laboratories (Garmisch, Germany, September 1993).

[58] Pressman, RS. Reengineering. In**:** *Software Engineering: A practitioner's approach,* Pressman, RS. ed. 5th ed. McGraw-Hill Companies, Inc. NY, 2001, pp 799-824.

[59] Project #1: Electronic Personal Organic Chemistry Homework (EPOCH), http://epoch.pearsoncmg.com/

[60] Project #2: ILIAS, http://www.ilias.uni-koeln.de/ios/index-e.html

[61] Rombach, H.D.., Basili, V., Selby, R. Experimental Software Engineering Issues: Critical Assessment and Future Directions. Lecture Notes in Computer Science. Springer Verlag, 1993.

[62] Shneiderman, B. *Designing the user interface***:** *strategies for effective human-computer interaction*. Addison-Wesley, Reading, MA, 1992.

[63] Shooman. M.L., and Bolsky. M.I. Types, distribution, and test and correction times for programming errors. Proceedings of the international conference on Reliable software, p.347-357, April 21-23, 1975, Los Angeles, California

[64] Sullivan, M., and 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).

[65] Tsuchiya, T., and Kikuno, T. On fault classes and error detection capability of specification based testing. ACM Transactions on Software Engineering Methodolgy, vol. 11, No. 1, January 2002, pp. 58-62.

[66] Warren-Smith, RF.**:** Starlink project, Rutherford Appleton Laboratory, http://star-www.rl.ac.uk/star/docs/sgp42.htx/sgp42.html#stardoctoppage.

[67] Wohlin, C. and Andrews, A. Analysing Primary and Lower Order Project Success Drivers. Proceedings of the Software Engineering and Knowledge Engineering (SEKE) 2002, Isclina, Italy, July 2002, CS Press.

[68] Xie, Y., and Engler, D. Using redundancies to find errors. IEEE Transactions on Software Engineering, vol. 29, No. 10, October 2003.

[69] Yu, WD., Barshefsky, A., and Huang, ST. An empirical study of software faults preventable at a personal level in a very large software development environment. Bell Labs Technical Journal 1997; 2**:** 221-232.

[70] Zaremski, A. M., and Wing, J.M. Specification matching of software components. ACM Transactions on Software Engineering and Methodology, vol. 6, No. 4, October 1997, pp.333-369.

VITA

Author's Name – Inies Raphael Chemmannoor Michael

Birthplace – Chennai, India

Birth-date – July 5th, 1978

Education

Bachelor of Engineering, Computer Science, University of Madras, Chennai, India. May 2000. Project: *Digital Image Processing and Compression*. Advisor: Ravi Kumar.

Higher Diploma in Software Engineering, Computer Science, APTECH LTD., Chennai, India. December 1999.

Areas of Specialization

Fault Based Analysis, Fault Classification, and Verification &Validation.

Research Experience

Research Assistant, UKY, Lexington, KY, 12/2003 – 7/2004.

Identified guidelines to write better software requirements. Conducted a comprehensive literature survey to provide guidelines for specific problems in requirements. This research was funded by NASA.

Teaching Experience

Teaching Assistant, UKY, Lexington, KY, 08/2003 – 12/2003.

The course was titled "CS115- *Introduction to Computer Programming*". Tutored C++ to undergraduate students from several departments. Maintained course grade web page, graded homework's and programming assignments.

Work Experience

Research Assistant, College of Ed. (COE), UKY, Lexington, KY, 7/2004 – present.

Job responsibilities include MySQL and MS Access database administration, maintenance of Access front-end, working on the DAME-portal and Retention projects, working with Web-focus a report generating tool, automating data transfers within the department and also from the university's Student Information System (SIS) and providing system support to the staffs in Academic Services and Teachers Certification at COE.

System Administrator and Programmer, Department of Veterinary Science, UKY, Lexington, KY, 01/2002 – 04/2003.

Managed data using *Visual Basic* as front-end and *Access Database* as the back-end. MySQL database was also used for rudimentary data storage. Performed hardware and software maintenance.

Professional Affiliations

IEEE Computer Society