Fault Links: Exploring the Relationship Between Module and Fault Types

Jane Huffman Hayes[1], Inies Raphael C.M.[1], Vinod Kumar Surisetty[1], Anneliese Andrews[2]

[1]University of Kentucky
Computer Science Department
hayes@cs.uky.edu

[2]Washington State University
School of Electrical Engineering and Computer Science
aandrews@eecs.wsu.edu

Abstract. Fault links represent relationships between the types of mistakes made and the type of module being developed or modified. The existence of such fault links can be used to guide code reviews, walkthroughs, allocation of verification and validation resources, testing of new code development, as well as code maintenance. We present an approach for categorizing code faults and code modules, and a means for examining relationships between the two. We successfully applied our approach to two case studies.

1. Introduction

As we seek to develop ever more complex systems, some with grave consequences of failure, we must strive to improve our technologies for developing and ensuring robust, reliable software. Fault-based analysis and fault-based testing are related technologies that seek to address this challenge.

Fault-based testing generates test data to demonstrate the absence of a set of pre-specified faults. Similarly, fault-based analysis identifies static techniques (such as traceability analysis that should be performed to ensure that a set of pre-specified faults do not exist. As part of fault-based analysis, a project manager can use historical data to determine what fault types are most likely to be introduced or can perform a risk analysis to determine what fault types would be most devastating if overlooked. Note that fault-based analysis is an early lifecycle approach that can be applied prior to implementation [15]. 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.

Based on our work on a semantic model of faults [30], Offutt's work on testing coupling [29], our work on traceability [16], and on requirement faults [15], we developed a conjecture about faults: The types of mistakes made by programmers largely depend on the type of module that is being developed or modified. We refer to this as a "fault link". 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. Though this may seem intuitive or "not surprising," note that currently there are no empirical results to confirm it.

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 preventative 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 a computational-centric module is being examined, 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.

The remainder of the paper is organized as follows. Section 2 presents related work. Sections 3 and 4 will present the module taxonomy and fault taxonomy, respectively. Section 5 discusses the research conjectures. Section 6 discusses two open source software case studies. We found evidence in favor of four of the conjectured fault links (as well as weak evidence for an additional two), such as Data-centric modules having many Data faults. We also found evidence of six unexpected fault links. Conclusions and future work are presented in Section 7.

2. Related Work

Faults have traditionally been characterized by syntactic categories [4, 22, 19], including where in the program the faults appear [17], which software development phase generated the faults [25, 20], what testing phase found the faults [30], and what type of statement or language feature the faults occur on [12]. As part of a NASA-funded project, Hayes has developed a taxonomy of requirements faults that is based on syntactic problems in the requirements [15].

A few attempts have been made to classify faults based on the mental mistakes that programmers make. IBM's ODC is one such scheme [18]. It assigns mental mistakes as part of a larger classification scheme.

Researchers have also examined change patterns of modules. Gall et al [13] used information about changes covering a sizeable number of releases to uncover logical dependencies and change patterns among modules. This was used to identify logical coupling among modules to uncover structural shortcomings. The work does not discriminate between corrective maintenance or enhancement related changes, thus did not attempt to classify faults. Similarly, Bieman et al [5] identified change-proneness of C++ code based on intentional use of patterns (or lack thereof). While this analysis found that some patterns are more change-prone in different categories of maintenance (corrective versus enhancement related changes), these faults were not classified. Bieman et al [6] found a strong relationship between class size and number of changes; larger classes changed more frequently. Also, classes that participate in design patterns and/or are reused through inheritance are more change-prone. They did not identify the type of change or fault in these studies.

Ohlsson et al [31] 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 type of file). The analysis of the case study data showed that fault-prone modules showed 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 [24], 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 "fault prone" or not "fault prone [28]."

Ostrand et al [32], 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 examining a series of 13 releases of a large evolving industrial software system, they observed that: (i) faults are concentrated in small numbers of files and in small percentages of code mass, (ii) shortchanging 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"[32].

However, no effort was made to classify modules and faults. Fenton et al [11] have quantitatively analyzed the faults and failures of a major commercial system. Some of their observations were identical to those made by Ostrand et al [32]. 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.

3. Module Taxonomy

Any simple or complex program can be viewed as a combination of various modules. A module is just a part of a program, which aids in performing some action or in making decisions to perform actions. A module can be a single statement or a single function or procedure that contributes to the purpose of the program.

We identified two methods for categorizing modules by type:

- Method one: Program modules are classified based on their main purpose. We considered allowing modules to have a second category based on their secondary purpose, but decided against it for the present. This represents a possible area for future work. This method is easy to comprehend and apply and is also faster than method two. However, it does not easily lend itself to automation.
- Method two: Modules 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 module, select the category with the highest Lines Of Code, and assign the module to that category. For example, "IF (salary > 1000)" is a controller statement. This method provides information about the statements used in a program and is easily automated with some standard guidelines. Unfortunately, there are drawbacks including: (i) diffi-

cult to perform categorization, (ii) time consuming, (iii) tedious when performed manually, and (iv) not easy to understand.

This paper classifies modules using method one. We followed a subset of the steps in [15] to develop module and code fault taxonomies: select a fault taxonomy as the basis for the work, examine sample code faults, adopt or build a method for extending the fault taxonomy, and implement the method for tailoring a taxonomy.

Our original module and fault taxonomy was influenced by the prior work discussed in Sections 2 and 4. We also performed a pilot study on an industrial partner's project as well as on student programming assignments to further construct the taxonomies. We applied the two taxonomies to categorize the faults and modules in two open source web-based projects, and detected new categories for both the fault and module taxonomies. Two new module categories were added including error handling and environmental setup. Fig. 1 shows the resulting generic module taxonomy. It is applicable to most programs and domains, but could be tailored to a specific domain or application using the process in [15]. 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.

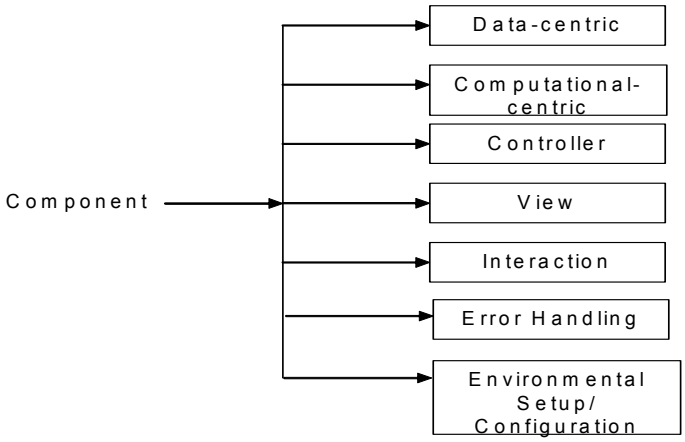

Fig. 1. Taxonomy of program modules

- Error Handling: The main purpose of modules in this category is to handle exceptions or errors that are likely to occur.
- Computational-centric: 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.
- Controller**:** Any module whose main purpose is to control the sequence of program execution falls under this category.
- Environmental setup/configuration: The main purpose of the modules is to set up an appropriate environment for the software to function efficiently.
- View**:** 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.

- Interaction**:** 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.

4. Fault Taxonomy

Our fault taxonomy does not include errors that can be caught by the compiler at compile time. We attempted to make the module and fault taxonomies generic enough to be language independent and method independent. Fig. 3 presents a graphical depiction of the taxonomy. 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 [3, 9, 10, 14, 21, 23, 35, 36].

Data**:** *Incorrect data definition.* Data definition involves assigning a name, type, and size for a data item. Since some data types are compatible with others (e.g., float can take an integer value), misuse can result in errors that are not detected at compile time. *Improper data initialization* is caused by the failure to initialize or reinitialize a data structure properly upon module entry or exit [3]. Examples of this include control blocks, registers, or switches not cleared or reset before transition [10]. *Improper data representation*. By representation we mean the ways in which the data is stored, i.e., data structure. The information or data can be stored in different ways, e.g., structured as a database or unstructured in flat files. Program statements that don't properly account for data representation may compile, but could result in runtime problems.

Computational**:** Errors that lead to a wrong value being calculated for a variable or register or switch belong in this class.

Control/logic**:** "Errors that cause an incorrect path in a module to be taken are considered control errors [3]." We group logic errors here also. *Statement logic* [36] faults cause the executable statements to be executed in the wrong order or not at all. For example, a program may fail to perform validation before returning the data. *Sequence errors* [36] exist when the order in which messages and control information are sent is erroneous. For example, the server program in a client-server environment may send an acknowledgment without receiving any request from the client. *Unreachable code* [25] occurs due to errors in control or logic statements. *Performance* faults may affect the overall performance of the software.

Interface**:** Here we include "errors associated with structures existing outside the module's local environment but which the module used" [3] and errors in the communications between modules. For example, incorrect subroutine or module call, insufficient data transfer [25], "incorrect declaration of COMMON segment" [3] all fall under this category.

User interface: Faults that interfere with the efficiency, performance and appearance of the user interface of the software. *Large response time* [23, 35] causes the in-

terface controls to respond with delay. *Lack of naturalness* [21] is caused by a number of factors such as illogical grouping of information, use of uppercase, use of arbitrary abbreviations, etc. A natural interface does not cause the user to significantly alter his or her approach to the task in order to interact with the system. *Inconsistency* [35, 21, 14] refers to the lack of a pattern of familiarity designed throughout a product. *Redundancy* [21] in a user interface requires the user to enter unnecessary information for an operation. For example, a user should never have to supply leading zeros ("00090.45" instead of "90.45"). *Complexity* [35] leads to interfaces that are not simple and easy to work with. The interface must be simple. The complexity of a user interface is based on the following factors**:** ease of use, ease of learning and understanding, and ease of navigation. *Lack of support* [21] refers to the limited amount of assistance the interface provides to the user. *Not flexible* [21, 14] refers to a user interface that narrows the types of users that can work on the software. The user interface must be able to tolerate different levels of user familiarity and performance. *Unpredictable flow* is when the flow of control in the user-interface gets beyond the scope 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. *Visual stimulation* [35, 21] refers to faults dealing with the improper use of color, fonts, graphics, control layout, etc. The determination that a fault exists is based on a bug report. Thus, we do not need to define metrics to measure attributes like "ease of use" or "ease of navigation".

Framework [9]**:** There are certain languages that make use of the concept of packages or reusable code. In such a language, a particular program imports or includes some of the packages to avoid unnecessary work. The set of statements used for this purpose is classified as "framework." *Missing framework elements* are caused when, upon integration, some modules might not have included required setup files. When all the modules are compiled together or individually, the compiler does not show any errors. However, at run time when the module calls or tries to communicate with another module, an error occurs. As mentioned before, only the leaves of the classification tree may be language dependent. Thus the fact that a framework element is missing is language independent, while the specifics of element mismatch will be language dependent.

5. Research Conjectures

After developing the fault taxonomy and module taxonomy, we noticed a strong correspondence between the categories, resulting in the following research question: "Does the module type drive the fault type one encounters?" We developed several research conjectures about fault links based on this. They are justified by prior work and the pilot studies mentioned earlier. The following 10 fault links were posited:

C1.1 – Data-centric modules will have a higher percentage of Data faults.
C1.2 - Data faults will occur more frequently in Data-centric modules.
C2.1 – Controller modules will have a higher percentage of Control/Logic faults.
C2.2 - Control/Logic faults will occur more frequently in Controller modules.
C3.1 – Computational-centric modules show a high percentage of Computation faults.
C3.2 – Computation faults occur more frequently in Computational-centric modules.

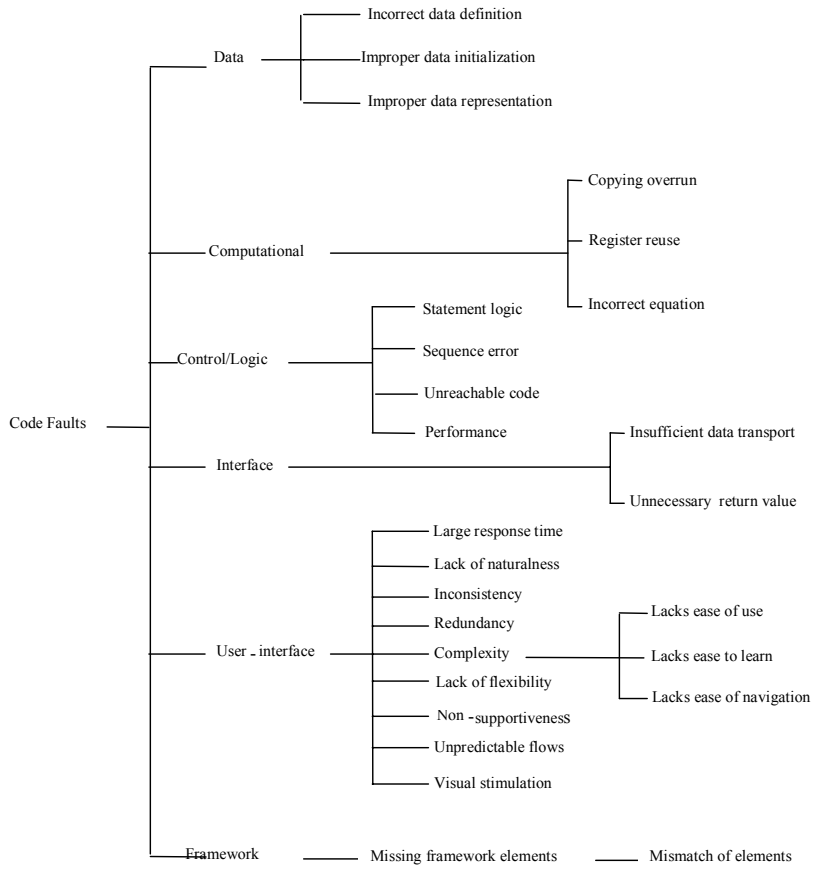

Fig 3. Fault Taxonomy

C4.1 - Interaction modules will have a higher percentage of Interface faults.
C4.2 – Interface faults will occur more frequently in Interaction modules.
C5.1 – View modules will have a higher percentage of User Interface faults.
C5.2 – User Interface faults will occur more frequently in View modules.

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

C6.1 – Interaction modules will have a higher percentage of User Interface faults.
C6.2 – User Interface faults will occur more frequently in Interaction modules.
C7.1 – View modules will have a higher percentage of Framework faults.
C7.2 – Framework faults will occur more frequently in View modules.
C8.1 – Error Handling modules will have a higher percentage of Data faults.
C8.2 – Data faults will occur more frequently in Error Handling modules.

C9.1 – Environmental Setup/Configuration modules will have a higher percentage of framework faults.
C9.2 – Framework fault occur more frequently in Environmental Setup/Configuration modules.

6. Case Studies

We applied the taxonomy to two open source software systems, Apache and Mozilla, to evaluate whether this taxonomy can be applied to common types of software and to see whether bug reports typical for such applications are able to reveal enough information. Both systems are largely written in C/C++. The Apache sever is a powerful, flexible, HTTP/1.1-compliant web server [2]. Mozilla is an open-source web browser, designed for standards compliance, performance, and portability [26]. Bugzilla is a "Bug-Tracking Systems" used in the Mozilla project. It allows individual or groups of developers to effectively keep track of outstanding bugs in their product.

6.1 Apache Case Study (Modules and Faults)

We examined all 30 modules that existed at the time of the study (100%). The size of the modules ranged from 250 LOC to 4500 LOC. We randomly selected two releases for which to examine bug reports, releases for the years 1999 and 2000. For those years, there were 2300 bug reports. Of these, we examined 300 (13%). Of these 300, only 177 bug reports provided enough information for fault categorization.

We classified all modules of the Apache 1.3.24 server [2] based on module purpose (method one). Table 1 presents the distribution of the Apache module classification. In this table, the percentage column denotes the percent number of modules of a particular type. For example, 10% of the 30 (3) modules were categorized as View modules. The largest module categories were Controller and Computational-centric, at 26.7% each.

Next, we applied method two to classify a subset of the Apache modules. Though many modules were categorized as belonging to the same categories when using method one and method two, some modules were not. For example, the following module was categorized as Computational-centric using method one, but was typed as Controller using method two.

Module**:** mod_unique_id
Main purpose**:** generate unique request identifier for every request
Method one**:** classified as computational module
Method two**:** As you can see from Table 2, the number of lines of code performing control/logic (Controller) functions is greater than the number of lines performing other functions. Therefore it is categorized as Controller.

The advantage of method two is that categorization can be automated. However, the results of method two are not always intuitive. Method one is more subjective than method two. While subjective measurement can and should be systematic, it lacks the rigor of objectively measurable and quantitative scales [38]. To account for this, one normally develops reliability indicators for such scales (for example, inter-rater reliability) [1]. To that end, we performed an inter-rater reliability survey. We

had five software engineers apply method one to this same module. The engineers were given the code for the module (including in-line documentation) and a list of and definitions for the module types in our taxonomy. All five engineers labeled the module as "computational." This convinced us that our subjective method exhibits reliability, so we continued using the results from method one.

Because of our interest in the relationship between module type and fault type, we performed a second step for the Apache case study. We went back to the 30 modules we had categorized and attempted to locate bug reports or problem reports for each. The problem reports provide information on identified faults. These have not necessarily been fixed. Several hundred bug reports were listed for each module. We examined a subset of these bug reports for a subset of the 30 modules (cf. Table 3).

Some general observations can be made. Many bug reports did not document bugs. Some bug reports represented enhancement requests. Bug reports had been generated by users who were "just trying out the bug tracking system." Many bug reports did not relate to code faults, but to poor documentation. Some bug reports did not relate to the version of Apache that we were examining or did not state the version number. Bug reports were duplicated or not deemed errors by the Apache developers. Finally, many bug reports documented more than one code fault and should have been separated into multiple bug reports. On average there were 1.5 faults per bug report.

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 eliminated bug reports that did not relate to version 1.3.24 of Apache or were not actual errors per the Apache engineers. 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 of each type (e.g., we examined eight Computational-centric, but only two Interaction modules), we looked at the faults as a function of the number of faults per module. That is, we examined 33 faults for four Data-centric modules. The 33 faults were categorized according to the fault taxonomy. The resulting values were scaled to reflect 8.25 faults per module.

Table 3 shows the module and fault classification for the Apache study. Module types are shown in the rows. The columns indicate: the total number of modules of different types that were examined; the number of faults, by fault type, for each module type; the total number of faults for the module type; and the percentage of faults found in a particular module type. For example, the Controller module row indicates that six such modules were examined, that 28 Control/Logic faults were found in the Controller modules, that a total of 47 faults were found in controller modules accounting for 26.6% of all faults classified. The highest value in each row has been bolded, and the highest value in each column has been italicized. In the above example, the value "28" has been bolded and italicized as it is the highest value for both the row and column. The bottom row indicates the percentage of each fault type classified. For example, 91 Control/Logic faults were found and they accounted for 51.5% of all 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 such 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.

Table 1. Classification of Apache Modules by Type

Module	Number	Percentage
Data-centric	6	20 %
Controller	8	**26.7%**
Computational-centric	8	**26.7%**
View	3	10%
Interaction	3	10%
Error Handling	1	3.3%
Environmental Setup/Configuration	1	3.3%
Total	30	100%

Table 2. Mod_Unique_ID Categorization-Method Two

Module	LOC (Lines Of Code)
Data-centric	62
Controller	**68**
Computational-centric	12
View	0
Interaction	11
Error Handling	0
Environmental Setup/Configuration	0
Total	153

Table 4 illustrates the "have" relationship that exists between the module and fault types. For example, a Data-centric module has more Control/Logic faults than any other type of fault, and these account for 48% of the faults typically found in a Data-centric module. The module types are listed in the rows of the table. For ease of illustration, two columns (the total number of modules of different types that were examined and the total number of faults grouped by the module in which they occur) have been repeated here from Table 3. We also show the total faults per module, followed by the percentage of faults found in a particular module type. Note that changes have been made to the values in the module-fault cells. Each cell has two values, a percentage value and a fault-per-module value. The percentage value represents the "have" relationship. The fault-per-module value indicates that out of N total faults in a module of a particular type, X of them belongs to a particular fault type. For example, let us examine the row for the Data-centric module. The number of data-centric modules examined was four, the total number of faults from the four modules was 33, and therefore the total faults per module (33/4) is 8.25 per module. This total fault-per-module value is distributed across the fault types based on their count from Table 1. As far as the fault distribution across different fault types is concerned, the data-centric module had about 18% data faults, 48% C/L faults, 6% computational faults,

9% interface faults, 18% framework faults, and zero percent GUI faults. As before, the highest value in each row and column is bolded and italicized respectively.

Table 5 illustrates the "occurs-in" relationship that exists between the fault and module types. For example, Data faults tend to occur in Data-centric modules most frequently (46%). The table is very similar to Table 4 except that it illustrates the "occurs-in" relationship from the fault type to the module type. The cells have the same two types of values as before, the percentage value and fault-per-module value. The fault-per-module value has the as meaning as before, but the percentage value in this case represents the "occurs-in" relationship. For example, let us examine the row for the Control/Logic fault. We can see that 16.8% of the C/L faults occur in data-centric modules, 19.6% of the C/L faults occur in controller modules, 11% of the C/L faults occur in computational-centric modules, 14.7% of the C/L faults occur in interaction modules, 21% of the C/L faults occur in view modules, 12.6% of the C/L faults occur in error-handling modules, and 4% of the C/L faults occur in environmental setup modules. The total faults-per-module and faults-per-module values of each are calculated as before.

Next, we assess the "have" relationship (from Table 4). The most frequently occurring fault type in Data-centric modules was Control/Logic at 48% (no close second). This does not support C1.1. The most frequently occurring fault type in Controller modules was Control/Logic at 59.6% with no close second. This does strongly support C2.1. The most frequently occurring fault type in Computational-centric modules was Control/Logic at 48% with no close second. This does not support C3.1. The most frequently occurring fault type in Interaction modules was Control/Logic at 41% with no close second. This does not support C4.1 or C6.1. The most frequently occurring fault type in View modules was Control/Logic at 50% with no close second. This does not support C5.1 or C7.1. The most frequently occurring fault type in Error Handling modules was Control/Logic at 75%. This does not support C8.1. There was a tie for most frequently occurring fault type in Environmental Setup/Configuration modules, 50% for both Control/Logic and Computational (no support for C9.1).

Table 5 illustrates the "occurs-in" relationship that exists between the fault and module types. For example, Data faults tend to occur in Data-centric modules most frequently (46%). The table is very similar to Table 4 except that it illustrates the "occurs-in" relationship from the fault type to the module type. The cells have the same two types of values as before, the percentage value and fault-per-module value. The fault-per-module value has the as meaning as before, but the percentage value in this case represents the "occurs-in" relationship. For example, let us examine the row for the Control/Logic fault. We can see that 16.8% of the C/L faults occur in data-centric modules, 19.6% of the C/L faults occur in controller modules, 11% of the C/L faults occur in computational-centric modules, 14.7% of the C/L faults occur in interaction modules, 21% of the C/L faults occur in view modules, 12.6% of the C/L faults occur in error-handling modules, and 4% of the C/L faults occur in environmental setup modules. The total faults-per-module and faults-per-module values of each are calculated as before.

Next, we assess the "have" relationship (from Table 4). The most frequently occurring fault type in Data-centric modules was Control/Logic at 48% (no close second). This does not support C1.1. The most frequently occurring fault type in Controller modules was Control/Logic at 59.6% with no close second. This does strongly sup-

port C2.1. The most frequently occurring fault type in Computational-centric modules was Control/Logic at 48% with no close second. This does not support C3.1. The most frequently occurring fault type in Interaction modules was Control/Logic at 41% with no close second. This does not support C4.1 or C6.1. The most frequently occurring fault type in View modules was Control/Logic at 50% with no close second. This does not support C5.1 or C7.1. The most frequently occurring fault type in Error Handling modules was Control/Logic at 75%. This does not support C8.1.

There was a tie for most frequently occurring fault type in Environmental Setup/Configuration modules, 50% for both Control/Logic and Computational. This does not support C9.1.

Table 3. Module and Fault Type Classification for Apache Study

Module type	# modules	Fault type						Total Faults	%
		Data	C/L	Comput.	Interface	Framework	GUI		
Data-centric	4	*6*	**16**	2	3	*6*	0	33	18.7%
Controller	6	3	***28***	5	4	5	*2*	47	26.6%
Computational-centric	8	2	**21**	*7*	*7*	*6*	1	44	24.8%
Interaction	2	0	**7**	3	3	2	*2*	17	9.6%
View	3	3	**15**	4	1	5	2	30	17%
Error Handling	1	0	**3**	1	0	0	0	4	2.2%
Environ. Setup	1	0	**1**	**1**	0	0	0	2	1.1%
Total	25	14	91	23	18	24	7	177	100%
Percentage		8%	51.5%	13%	10%	13.5%	4%	100%	

As can be seen from Table 5 (the "occurs-in" relationship), the majority of the Data faults occur in the Data-centric modules (46%). The next highest value is 30.7% for View modules. This finding provides support for C1.2, but not for C8.2. The majority of Control/Logic faults occur in View modules (21%) with Controller modules bringing up a close second at 19.6%. This finding lends some support to C2.2, but not as strong as for C1.2. Computation faults occur 36% of the time in Interaction modules followed by View modules at 19%. This does not support C3.2. Interface faults accounted for 36% of the Interaction module faults with no close second. This strongly supports C4.2. The majority of Framework faults occurred in View modules (29%) with Data-centric modules close behind at 26%. This provides some support for C7.2, but not C9.2. 47% of the User Interface faults occur in Interaction modules (supports C6.2, but not C5.2).

Our findings are summarized in Table 6. The basic question was: "Does the module type drive the fault type?" Six 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). This finding would lead one to

answer the overarching question "no." Our results are still inconclusive, but appear to hold promise.

6.2 Mozilla Case Study (Faults and Modules)

Next, we examined problem reports for the open source software product Mozilla (web browser) using the bug tracking system Bugzilla [26]. Mozilla is a very large software system and provided a plethora of problem reports for sampling. We examined 70 bug reports, selected randomly using Bugzilla. From these, 75 faults were identified that were code-related. Note that the "fault per problem report" ratio was only 1.07 as compared to 1.5 for Apache. These faults were categorized using our fault taxonomy. Table 7 presents the high level distribution of the faults found in Mozilla. 53.4% of faults reported for the open source software Mozilla fall under the category of Control/Logic faults, reinforcing findings from the first case study.

We were not able to find the modules that tied to specific bug reports or vice versa, as we were able to do in Apache. So we next randomly selected 30 modules in the Mozilla directories and categorized them. As can be seen from Table 8, the majority of the modules fell under the category of Computational-centric (26.7%), with Controller just behind at 20%. This is consistent with our findings for the Apache study.

6.3 Comparison of Case Study

Both case studies exhibit strong similarities with regard to fault types and module types. For both systems, Control/Logic faults occurred most frequently: 50% for Apache and 53.4% for Mozilla. The next most frequent fault type for Apache was a tie between Interface and Framework at 14%. For Mozilla, it was Data at 17.3%. The fourth most frequent fault type for Apache was Data at 10%, and it was a three-way tie for Mozilla between Computational, Interface, and User Interface, all at 8%. A striking result was the dominance of the Control/Logic fault type, in both systems.

The most frequent module type for Mozilla was Computational-centric at 26.7%. Computational-centric was tied for most frequent with Controller at 26.7% for Apache. The next most frequent module type for Apache was Data-centric at 20%. For Mozilla, it was Controller at 20%. The third most frequent module type for Apache was a tie between View and Interaction, both at 10%. For Mozilla, Data-centric and View tied for 16.7%. Computational-centric and Controller occurred most frequently in both systems. A comparison of fault type percentages is shown in Fig. 2. In each category, the percent of faults in the two applications are similar (note that Apache does not report user interface bugs, since it is not interactive). For the common fault types, correlation analysis found a correlation value of 0.94 between the faults percentages. This is not surprising, as these applications share common characteristics: open source, web related. The result also confirms that our fault taxonomy is reasonable and applicable.

Table 4. "have" Relationship from Module to Fault Types for the Apache Study

Module type	# modules	Fault type						Total Faults	Total Faults/ module	%
		Data	C/L	Comput.	Interface	Frame-work	GUI			
Data-centric	4	*18%* 1.5	**48**% 4	6% 0.5	9% 0.75	*18%* 1.5	0% 0	33	8.25	18%
Controller	6	6% 0.5	**59.6** % 4.67	10.6% 0.83	8.5% 0.67	10.6% 0.83	4% 0.33	47	7.83	17%
Computat.-centric	8	4.5% 0.25	**48**% 2.63	16% 0.875	16% 0.875	13.6% 0.75	2% 0.12	44	5.5	12%
Interaction	2	0% 0	**41**% 3.5	17.6% 1.5	*17.6%* 1.5	11.7% 1	*11.7%* 1	17	8.5	18%
View	3	10% 1	**50**% 5	13% 1.33	3% 0.33	16.7% 1.67	6.7% 0.67	30	10	21.8%
Error Han-dling	1	0% 0	*75%* 3	25% 1	0% 0	0% 0	0% 0	4	4	8.7%
Environ. Setup	1	0% 0	**50**% 1	***50%*** 1	0% 0	0% 0	0% 0	2	2	4.5%
Total	25	3.25	23.8	7.035	4.125	5.75	2.12		[46.08, 46.08]	100%

Table 5. The "occurs-in" Relationship from Fault to Module Types for the Apache Study

Fault Type	Module Type							Total	%
	Data-centric	Controller	Computat.-centric	Interaction	View	Error Handling	Environmental setup		
#modules	4	**6**	8	2	3	1	1	25	
Data	***46%*** 1.5	15% 0.5	7.7% 0.25	0% 0	30.7% 1	0% 0	0% 0	3.25	10%
C/L	16.8% 4	*19.6%* 4.67	11% 2.63	14.7% 3.5	**21%** 5	12.6% 3	4% 1	23.8	50%
Computational	7% 0.5	11.7% 0.83	12% 0.875	**36%** 1.5	19% 1.33	*14%* 1	*14%* 1	7.035	9%
Interface	18% 0.75	16% 0.67	*21%* 0.875	**36%** 1.5	8% 0.33	0% 0	0% 0	4.125	14%
Framework	26% 1.5	14% 0.83	13% 0.75	17% 1	**29%** 1.67	0% 0	0% 0	5.75	14%
GUI	0% 0	15.5% 0.33	5.6% 0.12	***47%*** 1	*31.6%* 0.67	0% 0	0% 0	2.12	3%
Total faults	33	47	44	17	30	4	2	177	100%
Total Faults/module	8.25	7.83	5.5	8.5	10	4	2	[46.08, 46.08]	

Table 6. Conjecture Results

Conjecture	Conjectured Fault Link	Supported?
C1.1	Data modules have Data faults	No
C1.2	**Data faults occur in Data modules**	**Yes**
C2.1	**Controller modules have C/L faults**	**Yes**
C2.2	**C/L faults occur in Controller modules**	**Weak**
C3.1	Comp. modules have computational faults	No
C3.2	Comput. faults occur in Comput. modules	No
C4.1	Interaction modules have Interface faults	No
C4.2	**Interface faults occur in Interaction modules**	**Yes**
C5.1	View modules have User Interface faults	No
C5.2	User Interface faults occur in View modules	No
C6.1	Interaction modules have User Interface faults	No
C6.2	**User Interface faults occur in Interaction modules**	**Yes**
C7.1	View modules have Framework faults	No
C7.2	**Framework faults occur in View modules**	**Weak**
C8.1	Error handling modules have Data faults	No
C8.2	Data faults occur in Error Handling modules	No
C9.1	Environ. Setup/Config. Modules have framework faults	No
C9.2	Framework faults occur in Environ. Modules	No

We conclude with some remarks about threats to validity of our case studies. As with any case study, there are unavoidable threats to validity. First, we cannot generalize the results to other application domains, systems, or languages. What we can say, however, is that we found support for our taxonomy in both the Apache and Mozilla systems. Second, a case study is limited in the amount of control over what data can be collected. We were limited by the available bug reports. While the random selection of defect reports for both systems does not bias the results, the quality and information content of the bug reports possibly could. Given the nature of case stud-

ies, we had no control over how bugs were reported. The analysis depends on the quality of the bug reports. They have to contain enough information for fault classification. While we have not performed a large scale inter-rater reliability analysis of the module classification, we used a team to classify them, in line with guidelines in [1]. The analysis is based on a theory about a fault link taxonomy that is based on existing knowledge and empirical studies as explained in section 4. It is thus possible that new fault links may be found, and, of course, some applications may not have certain faults. We consider our work a step towards building a more comprehensive theory.

7. Conclusions and Future Work

We have developed two taxonomies, one for modules and one for code faults. We introduced the notion of a fault link. We presented two methods for module classification along with their advantages and disadvantages. We classified modules and code faults of two open source, web-based software products using our approach.

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.

Table 7. Mozilla Fault Types

Fault	Number	Percentage
Data	13	17.3%
Computational	6	8%
Control/Logic	40	**53.4%**
Interface	6	8%
User interface	6	8%
Framework	4	5.3%
Total	75	100%

Table 8. Mozilla Module Types

Module Types	Number	Percentage
Data-centric	5	16.7%
Computational-centric	8	**26.7%**
Controller	6	20%
View	5	16.7%
Interaction	1	3.3%
Error Handling	1	3.3%
Environmental setup	4	13.3%
Total	30	100%

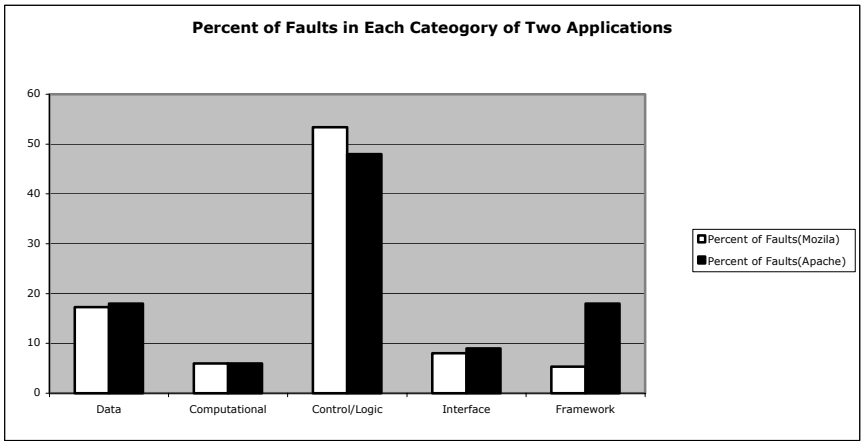

Fig. 2. Comparison of Fault Type Percentages

We continue work on the fault taxonomy and the module taxonomy and hope that others will assist us in validating and improving them. We plan to examine the taxonomies with respect to the object-oriented methodology. We plan to examine languages such as Lisp that provide control abstraction. We also are not convinced that the fault taxonomy is orthogonal. Specifically, we plan to evaluate mixed-purpose modules in the context of the fault link taxonomy. Our taxonomies might require tailoring to a specific domain or application, such as real-time or embedded systems, as discussed in [15]. We also plan to expand the fault link concept to 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. The ultimate goal of this work is to identify evaluation techniques 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, though complex, software systems.

8. Acknowledgements

We thank Jeff Offutt for never tiring of discussions about testing and faults. Thanks to Martin Feather and Alan Nikora of JPL who looked at early versions of these taxonomies. Thanks to Kirk Kandt, also of JPL, for excellent comments on an earlier version of this paper.

References

1. Allen, M. and Yeh, W. *Introduction to Measurement Theory*. Brooks/Cole Publishing, 1979.
2. Apache modules and problem reports, Apache HTTP server version 1.3.24, http://httpd.apache.org/docs/mod/index-bytype.html.
3. Basili, V.R. and Barry T. Perricone. ‘‘Software Errors and Complexity: An Empirical Investigation.’’ *Communications of the ACM*, 27, 1 (January 1984), 42-51.

4. Beizer, B. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd Edition, ISBN 0-442-20672-0, 1990.
5. 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).
6. 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).
7. Centre of Software Maintenance, University of Durham, England. http://www.dur.ac.uk/computer.science/research/csm/rip/introduction.html
8. Cooper, A. *About face: the essentials of user interface design.* IDG Books Worldwide, Foster City, CA, 1995.
9. 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.
10. 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.
11. 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.
12. Freimut, B. "Developing and Using Defect Classification Schemes", Fraunhofer IESE *IESE-Report No. 072.01/E, Version 1.0*, September, 2001.
13. 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.
14. 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.
15. 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).
16. 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).
17. 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.
18. IBM Research, Center for Software Engineering, "Details of ODC v5.11", http://www.research.ibm.com/softeng/ODC/DETODC.HTM.
19. IEEE Standard Classification for Software Anomalies, December 12, 1995. IEEE Std 1044.1-1995.
20. 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).
21. Macaulay, L. *Human -computer interaction for software designers.* International Thomson Computer Press, London, 1995.
22. Marick, B. A survey of software fault surveys. A technical report UIUCDCS-R-90-1651, University of Illinois, 1990; pp 2-23.
23. Mayhew, DJ. *Principles and guidelines in software user interface design***.** Englewood Cliffs, N.J. Prentice Hall, 1992.

24. Mayrhauser, A., Ohlsson, MC., and Wohlin, C.: Deriving fault architecture from defect history. *J. Softw. Maint. Res. Pract., 12*, (2000), 287-304.
25. 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.
26. Mozilla organization website, http://mozilla.org/.
27. 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).
28. 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).
29. Offutt, J. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering Methodology*, *1, 1* (January 1992), 3-18.
30. 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).
31. 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.
32. Ostrand, T. and Weyuker, W. The Distribution of Faults in a Large Industrial Software System. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA) 2002* and *ACM SIGSOFT*, vol. 27, No. 4, July 2002, pp. 55-64.
33. Perry, D.E., and C.S. Stieg, "Software Faults in Evolving a Large, Real-Time System: a Case Study", AT&T Bell Laboratories, *Proceedings of the 4th European Software Engineering Conference*, Garmisch, Germany, September 1993.
34. Rombach, H.D.., Basili, V., Selby, R. Experimental Software Engineering Issues: Critical Assessment and Future Directions. Lecture Notes in Computer Science. Springer Verlag, 1993.
35. Shneiderman, B. *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley, Reading, MA, 1992.
36. 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).
37. Warren-Smith, RF.: Starlink project, Rutherford Appleton Laboratory, http://star-www.rl.ac.uk/star/docs/sgp42.htx/sgp42.html#stardoctoppage.
38. 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.
39. 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