5-13-2006

A Framework For Assessing The Impact Of Software Changes To Software Architecture Using Change Classification

Byron Joseph Williams

A FRAMEWORK FOR ASSESSING THE IMPACT OF SOFTWARE CHANGES TO SOFTWARE ARCHITECTURE USING CHANGE CLASSIFICATION

By

Byron Joseph Williams

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

May 2006

A FRAMEWORK FOR ASSESSING THE IMPACT OF SOFTWARE CHANGES TO SOFTWARE ARCHITECTURE USING CHANGE CLASSIFICATION

By

Byron Joseph Williams

Approved:

Jeffrey Carver Assistant Professor of Computer Science and Engineering (Major Professor)	Roger King Associate Dean of the College of Engineering
Edward Allen Associate Professor of Computer Science and Engineering (Committee Member and Graduate Coordinator)	
Thomas Philip Professor of Computer Science and Engineering (Committee Member)	

Name: Byron Joseph Williams

Date of Degree: May 13, 2006

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Jeffrey Carver

Title of Study: A FRAMEWORK FOR ASSESSING THE IMPACT OF SOFTWARE CHANGES TO SOFTWARE ARCHITECTURE USING CHANGE CLASSIFICATION

Pages in Study: 83

Candidate for Degree of Master of Science

Software developers must produce software that can be changed without the risk of degrading the software architecture. One way to address software changes is to classify their causes and effects. A software change classification mechanism allows engineers to develop a common approach for handling changes. This information can be used to show the potential impact of the change. The goal of this research is to develop a change classification scheme that can be used to address causes of architectural degradation. This scheme can be used to model the effects of changes to software architecture. This research also presents a study of the initial architecture change classification scheme. The results of the study indicated that the classification scheme was easy to use and provided some benefit to developers. In addition, the results provided some evidence that changes

of different types (in this classification scheme) required different amounts of effort to implement.

DEDICATION

I would like to dedicate this work first to God, as all that I do is to please Him. Secondly, I would like to dedicate this work to my family and friends that have supported me throughout my educational endeavors.

ACKNOWLEDGMENTS

I would like to acknowledge Dr. Jeffrey Carver for his support and guidance that he provided throughout the research process. I would also like to thank the Empirical Software Engineering Research Group at Mississippi State University and the Fraunhofer Center for Experimental Software Engineering, Maryland for their support. This research is funded by NSF Grant CCF-0438923.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

TABLE Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

1.1 Problem Statement

Late lifecycle changes are unavoidable in software intensive systems. The flexibility of a system will determine whether these late changes will cause an architecture violation, i.e., a defect in the software architecture that reduces flexibility. These violations are caused by many reasons, including making late changes to the architecture without a thorough understanding of the architecture and not knowing how the change will affect the architecture. A change that is implemented in software that reduces system flexibility will cause the architecture of the system to degrade and will ultimately lead to the system becoming harder and more expensive to maintain. As more changes are required, system quality will continue to degrade and eventually the system must be reengineered or retired. The goal of this research is to help developers understand how certain types of changes will affect software architecture. A better understanding of the system will enable developers to make changes and account for the architectural impacts of the change and implement quality assurance techniques that will allow the system to receive the change without a reduction in flexibility.

1.2 Background and Motivation

The nature of a software intensive system is that it will change over time. Making changes to software intensive systems is a crucial aspect of maintenance and has become an ever increasing challenge for software developers. For this reason, software maintenance has been regarded in part as the most expensive phase of the software lifecycle [11]. The more a system changes, the more complex it becomes. This increased complexity can make the system less understandable for the developers and ultimately result in decreased system quality.

Changes that must occur late in the software lifecycle pose an especially high risk for developers, but are often unavoidable. These late changes are those that occur after at least one cycle of the development process has been completed and a working version of the system exists. Understanding late changes is important, due to the high cost of these changes, both in money and effort, especially when they occur at the requirements level. The most crucial changes tend to occur later in the lifecycle, as the customers and end-users are better able to determine their needs. Implementation of these changes often results in a less flexible system and forces deviation from the original design [8]. There are many sources for late changes including defect detection, changing market conditions, changing software environment, and evolving user requirements over time. Due to the timing of these crucial late changes, it is often not possible to fully evaluate their impact on the system architecture [10]. As a result, system quality decreases as the software architecture is degraded, making defects more likely and future changes more difficult [15].

When dealing with late changes, a focus on software architecture is important because the architecture defines the structure and interactions of the system. When a change affects the architecture of the system, the original architectural model prescribed by the developers must be maintained to ensure that the system remains flexible and will continue to function as originally designed. When a change affects the structure of the system causing the interactions to become increasingly complex, the architecture tends to degenerate and become unmaintainable [28]. When an architecture degenerates, the actual functions of the internal and external system interfaces may not match their original purposes, resulting in confusion for developers and forcing the software system to be retired or undergo a major reengineering.

Developers need to better understand the effects of making a change before the actual change is made. A change classification mechanism that allows developers to conceptualize a change to a system before implementation will allow them to predict the effects of the change on the software architecture. The developers can then come to a consensus on exactly how the change request should be implemented, and take the necessary precautions to ensure that the change does not degrade the system architecture. This scenario can be accomplished by each developer first classifying the change request, agreeing on the impact categories, and then using historical change data, if available, to compare the impact of similar changes.

In general, change classifications have been created to help developers better assess the impact and risk associated with making certain types of changes to software. Several benefits of classifying changes have been identified in the literature, such as

identifying risks associated with change implementation and determining change acceptability [25]. Software change classification mechanisms also allow engineers to group changes based on different criteria, e.g., the cause of the change, the type of change that needs to be made, the area where the change must take place, and the potential impact of the change. Another benefit of change classification is that it allows engineers to develop a common approach to deal with the items in each class, resulting in less effort than if each change was addressed individually. Accurately classifying changes will also enhance our ability to quantify the effects of particular change classes [25].

Classifications have been successfully used for change impact analyses on the effects of source code modifications [3, 5, 6, 13, 14, 22, 23]. Often these source code changes affect the software architecture because there is not currently a classification scheme that focuses specifically on the impact of changes on the architecture. Better understanding the impact of the types of changes that affect architectures will enable developers to model the proposed changes, predict the implementation cost and, facilitate a deeper understanding of the software architecture change process. As a result, higher quality software will be developed throughout the product lifecycle.

This research presents an initial architecture change classification scheme developed to address some of the problems associated with architectural degeneration and provide a foundation for a future decision support model for handling changes. An exploratory study was conducted to assess the usefulness of the scheme and to improve the change scheme for further study.

This research attempts to minimize the effects of architecture degradation by providing a means of understanding software changes and their effects on software architecture. Architectural degradation is a problem caused in part by late changes to software systems. These changes are unavoidable, but the effects of these changes can be controlled and the negative effects minimized by knowing how the changes will affect the architecture and whether to allow, defer, or not allow a change. There have been few rigorous studies on the problem of guaranteeing quality while making changes late in the software lifecycle. This research makes an initial effort to do so.

1.3 Research Plan and Related Goals

The objective of this research was to develop a detailed classification scheme that modeled the effects of changes to software architecture. This research was conducted in the following three phases: conception, refinement, and observation. These three phases followed the goals set forth by the research.

The *conception* phase focused on the creation of an initial architectural change classification scheme. The *refinement* phase included the analysis of historical data in order to identify trends in certain classification types and to improve the scheme by adding, subtracting, or modifying change categories based on historical change data. The *observation* phase served to evaluate the usefulness of the classification scheme and identify areas of weakness.

The goals of this thesis have been described using the Goal Question Metric (GQM) format. Each GQM goal contains an object of study, a model of one or more foci, a point of view, a purpose, and an optional description of the context [2]. Using the GQM

framework ensures that this research focuses on specific measurable goals. For each GQM goal, questions of interest are presented to characterize the object of measurement. Finally, the metrics associated with each question enable it to be answered through the specific data collected.

1.3.1 Conception

The conception phase started with a literature review to understand the change process, understand how changes affect software systems, and learn about current classification schemes. The second part of the conception phase included a more detailed review of the literature with the specific goal of identifying various types of software changes and their affect on software architecture. These changes were then categorized and used as inputs to the architecture change classification scheme. The areas of the software environment affected by each change were also included in the scheme to provide a complete picture of the effects changes have on the software system.

> G1 Analyze change classification research and change data in order to characterize changes with respect to their impact on software architecture from the point of view of the researcher

This initial goal focuses on the creation of a preliminary change classification scheme that models the effects of changes to software architecture. This goal provides a baseline for the subsequent goals that will use its outputs as inputs.

Several questions of interest associated with this goal are as follows:

> Q1 What are the major categories of changes?
>
> Q2 What areas of the system does the change impact?
>
> Q3 What is the result of the code modification/change?

Q4 What other software engineering documents are affected by change?

To answer these questions, the following metrics were used:

M1 Change classification models in literature

M2 Qualitative description of the implementation detail

M3 Lines added, modified, or deleted from any supporting software engineering documents

M4 Change impact analysis data

1.3.2 Refinement

The objective of the refinement phase was to improve the classification scheme in order to prepare it for observation in a classroom study. This phase consisted of using data from a historical change database to further refine the classification scheme. The historical dataset included change requests and some implementation data that was used to measure the effort needed to implement the change. We used the classification scheme to classify the change requests. The implementation was used to characterize the effect each change class had on the system. Due to the time constraints (i.e., the observation phase had to occur within 2 months of beginning the refinement phase), only one dataset was used to obtain change data for analysis.

The second goal takes the classification scheme as an input to further characterize the scheme.

G2 Analyze software change data in order to characterize it with respect to the change classification scheme from the point of view of the researcher in the context of a real project

The main focus of the second goal was to evolve the initial classification scheme by classifying changes found in a historical data set. Questions of interest associated with this goal are as follows:

Q5 What is the motivation for the change?

Q6 Are the categories of changes grouped in orthogonal classes, should they be?

Q7 What quantitative/qualitative measures can be taken for each change?

Q8 What trends are related to each category of change?

Several metrics have been identified that provided answers to the questions posed for G2. These metrics are:

M5 Detailed change request information

M6 Classification of changes from historic data sets

M7 Lines of code added, modified, or deleted per change class

M8 Number of modules changed, modified, or deleted per change class

After further refinement of the classification scheme, it was used as the foundation of a classroom study identified by the third goal of this thesis.

1.3.3 Observation

The objective of the observation phase is to gather evidence in support or against any hypotheses made after the collection and analysis of historical change data. This observation occurred during a classroom study where students used the classification scheme to classify and implement changes in a real development environment. An

analysis of the scheme's usability along with the amount of effort involved in making each type of change was recorded and analyzed in order to test the hypotheses.

The observation phase concluded with data collection and analysis. This data analysis will determine if the characteristics identified by each change class match the experiences of the students.

> G3 Analyze the software change classification scheme in order to characterize it with respect to usability, effort prediction, and architecture impact estimation from the point of view of the researcher in the context of a classroom study

The third goal was accomplished by conducting a classroom study that provided an initial determination of the usability and predictive capabilities of the change classification scheme. Question of interest associated with G3 include:

Q9 Did the researcher classifications match the subjects' classifications?

Q10 Is it the scheme easy to use?

Q11 Are the groupings/categories for the changes logical?

Q12 Are there any categories missing from the scheme?

Q13 How can the scheme be improved?

The metrics associated with G3 are listed below:

M9 Classification of changes by researcher & subjects

M10 Change implementation survey

M11 Change classification survey

M12 Person-hours required to make the change

M13 LOC added, modified, deleted

M14 Modules added, modified, deleted

1.3.4 Summary

Each phase of this research provided meaningful results that were used in subsequent phases and will be used in future research. Each question of interest has been answered using the data collected and is reported on in the remaining sections.

1.4 Research Hypothesis

When conducting preliminary studies, the initial findings serve as building blocks for future research efforts. This research presents a preliminary study of a classification scheme with implications in the area of software architecture change prediction and modeling, as well as decision support modeling for assessing the viability of proposed changes to software. In creating this classification scheme, one fundamental precursor to any grand prediction is whether or not the differences in classes of changes will result in a difference in the amount of measurable effort. Another elemental aspect of the creation of this scheme is determining how well the change classifications match what is generally understood by developers.

The goal of the research is the development of a software change classification scheme that will ultimately provide a basis for understanding changes and how to handle them in any environment. An initial hypothesis developed in the creation of this scheme is as follows:

> An architecture change classification scheme will allow developers to consistently classify changes given a change request and make implementation decisions based on the differing impact the various change classes have because items that fall into different classes will require differing amounts of effort to implement.

This software architecture change classification scheme will help developers to better understand change requests and how to categorize them, and show that the

differences in classes will result in differences in the amount of effort. This hypothesis in turn can be used in future studies to predict more exact amounts of effort required to implement a change, but only by first knowing and understanding that different changes classes will require different amounts of effort.

CHAPTER II

LITERATURE REVIEW

There is one certainty in software development, and that is all software will undergo change. Change is inevitable in software systems, because of the many factors influence the need for changes to software. These factors can range from a change in the user's needs, a change in the operating environment, a change due to a problem in the software, or even a change to prevent the need for future changes. Whatever the case, software is going to change, and developers are developing increasingly sophisticated ways to handle changes.

2.1 Software Change

Software change has been studied throughout the history of software development. A pioneer of the study of software changes, Manny Lehman, created the Laws of Software Evolution [16]. These laws describe recurring issues related to the evolution, or changing, of E-type software systems. An E-Type system is a software system that solves problems or implements computer applications in the real world and must be continually evolved to maintain user satisfaction [18]. Laws I, II, VI, and VII will be addressed in this research by examining methods for controlling the negative effects of these laws.

Law I, Continuing Change, states that software undergoes never-ending maintenance and development that is driven by the variance of its current capability and environmental requirements [16]. There are various reasons why software must change to accommodate such variance. This variance could be due to the changes in the technology that contains the protocols and standards used by an application to communicate with other systems. It could also be due to the changes in hardware and the need for more efficient utilization of hardware resources. Understanding the reasons for change will aid developers in developing systematic processes to handle change in a software intensive system.

As systems change, they tend to become more complex. Because change is inevitable in software systems, complexity will increase if not properly handled. This situation leads to Law II, Increasing Complexity. This law simply states that changes imposed by system adaptation lead to an increase in the interactions and dependencies among system elements. These interactions may be unstructured and increase the system entropy [17]. If entropy is not addressed and properly handled, the system will become too complex to adequately maintain. Law II is one of the primary reasons why the maintenance phase is typically the most expensive phase of software development [11]. In order to reduce and better manage system complexity, developers need improved ways of understanding changes and how to incorporate change into system architectures.

It has been shown that the number of system modules increase linearly with each incremental system release [16]. The law of Continuing Growth, Law VI, focuses on user needs by stating that the functionality of software systems must continually increase to

maintain user satisfaction over the lifetime of a system [17]. This law, while similar to Law I, reflects a different phenomena. The law addresses the tendency of the user base to become increasingly sophisticated and demand a more robust set of features requiring that the software grow over time to meet such needs. This growth will also include features that are not deemed satisfactory to the user base and must be adapted to fit user demands. This adaptation causes the system to grow. This law is often made evident by the omission of a detailed model of the application in its actual or desired operational domain. This lack of information is due to limitations that arise from constraints on budget, milestones, and technology, thus causing the specification of requirements to be bounded. When such items are excluded and are later requested by users, the system must grow its capability to provide attributes that were not originally accommodated in the initial development [17].

Each of the previous three laws discussed feed into Law VII, Declining Quality [17]. As changes are made, system complexity increases. The introduction of new features to a system causes it to grow continuously. All of these factors can serve to reduce the perceived quality of a system. When the quality of the system is reduced, it becomes more expensive to maintain because of an increase in the number of problems encountered by users. To address these problems, changes must be made to the system. These changes are likely to further increase the complexity and growth of the system which will, in turn, further reduce the quality [17]. This cycle results in a continuous downward spiral of quality.

The Laws of Software Evolution have been studied and its claims have been supported throughout software engineering literature [16, 26]. In understanding these laws, and the necessity of software change, researchers have developed methods of handling changes, e.g., using change classification schemes, impact analysis, and effort prediction models. These methods are continuing to improve. As more research is done in the area of understanding changes, more can be done for practitioners to help them implement system changes. Practitioners then will not have to suffer from the uncontrollable increase in complexity or decline in quality.

2.2 Current Research in Software Change Classification

There have been several change classification and change impact analysis schemes identified in the literature. Lam and Shankararaman identified several key change implementation areas and their relevance to software change classification research. These areas include process models used in change management, configuration management issues, defining change categories, and developing additional strategies to handle changes to software [14]. Each key area was taken into consideration when creating the architecture change classification scheme described in Section 3.1

Process models focus on modeling the change process in relation to the software engineering process. The PRISM model provides insight on the main components of the environment necessary to cope with changes. The two-tier approach of the PRISM model first attempts to identify items in the environment that will be affected by a change. Second, it classifies and records analytical data related to the change. Both aspects include feedback mechanisms used to influence changes and project future changes [20].

This model also focuses on how change propagates across the software environment, including the effects certain environment variables have on changes, such as people, policies, laws, and resources. The FEAST model (Feedback, Evolution and Software Technology) identifies the need for dynamic feedback in the software process model as a continuous system varying over time [17]. The architectural change process described by Nedstam describes the change process as a series of steps [24]:

1. Identify an emergent need
2. Prepare resources to analyze and implement change
3. Make a go/no-go feasibility decision
4. Develop a strategy to handle the change
5. Decide what implementation proposal to use
6. Implement the change.

An architectural change classification scheme will address steps 2, 3, and 4 in the above list by helping developers conceptualize the impact of a proposed change through the process of classifying the change request.

The goal of the study described in this paper is to analyze a new classification scheme designed to model the effects of changes to software architecture. Our change classification scheme will build on features of current change classification schemes that provide insight into aspects of change that in some way affect software architecture. Kung, et al. researched the impact of code changes on class inheritance structure within a software system where code changes resulted in small changes in the larger modules of the system [13]. Briand and Basili classified changes that occurred during the

maintenance process. They were able to approximate component changes that were isolated in the system architecture [3]. Nedstam, et al. identified changes to be either *architectural*: affecting the structure of the system; *functional*: affecting only user-observable attributes; or *somewhere in between*: affecting both user-observable attributes and the system architecture [24]. Many of the above change characteristics were considered in creating the architecture change classification scheme described in this research.

CHAPTER III

ARCHITECTURAL CHANGE CLASSIFICATION SCHEME

As a starting point for our research, we created an architectural change classification scheme to model changes that affect software architecture. This scheme built on change and defect classifications published in the literature. Although the focus of the scheme is on architectural changes, it also covers defects and functional changes that may or may not affect system architecture. Defects and functional changes were included to allow the scheme to be robustly used to model all types of changes, not exclusively architecture changes.

The questions of interest pertaining to the conception phase were answered by collecting and analyzing metrics presented throughout the remainder of this section. Each metric was found using various sources in the literature.

3.1 Classification Scheme Overview

The high-level classes for the architectural change classification scheme are derived from Mohagheghi's 4 change categories: perfective, preventative, corrective, and adaptive. *Perfective* changes result from new or changed requirements. These changes improve the system to better meet user needs. *Preventative* changes ease future maintenance by restructuring or reengineering the system when a potential problem is identified. *Corrective* changes occur in response to defects. *Adaptive* changes occur when

moving to a new environment or platform or to accommodate new standards or platforms [23]. These classes are orthogonal.

The goal for our classification scheme was to provide support for system developers and maintainers to assess the potential impact of a proposed change and to decide whether to implement the change. In cases where the change is crucial to the system, the scheme will help generate consensus on how to approach change implementation and provide an idea of its difficulty. The classification scheme is a collection of change and defect attributes. The scheme has been modeled as a decision tree that a developer can traverse while choosing values for each attribute.

Section 3.1.1 describes the process for arriving at the classification of a specific change. Some of the attributes of the categorization are orthogonal and others are not. The high level characterization of the change is an orthogonal choice. The remainder of the attributes that provide a more detailed view of the parts of the system that will be affected by the change are not orthogonal. For non-orthogonal attributes, each will be given a value from Table 1. The combination of these values then gives the developer an overall idea of the impact of that change. The classification produced by the decision tree will be based on the developer's experience with the system. It will help determine the difficulty of implementing the change and provide some reasoning about the amount of effort that will be required to make the change.

Table 1

Change Impact Scale

0 = No impact
1 = Small impact
2 = Significant impact
3 = Major focus of change

3.1.1 Classification Process

In addition to developing a classification scheme, we have also developed a process for using that scheme. Figure 1 illustrates the first two steps of the process. Given a change request, the developer first documents the lifecycle phase in which the change was requested. Next, the developer must decide whether the change is an enhancement or defect. An *enhancement* is a change that has the goal of improving the system for its stakeholders, while a *defect* is caused by an error, fault, or failure.

Within the two major change categories, additional subcategories exist to further detail the change request. In Figure 1, Figure 2, and Figure 3 each diamond represents an attribute in the classification scheme and a decision point. Attached to each diamond are the potential values for that attribute with the shape of the subtypes indicating whether the attribute is orthogonal. Attributes with rectangles are orthogonal attributes (i.e., only one choice is made), while attributes with circles are non-orthogonal attributes (i.e., multiple selects can be made). Sections 3.1.2 and 3.1.3 explain the detailed attributes for the enhancement and defect classes respectively. In both cases, during this initial study, we assume that each attribute is independent of the others. This assumption will remain until further analysis of the change classes provides evidence of a dependency.

3.1.2 Enhancement Details

After determining that the change is an enhancement, the first attribute to be determined is the *class* of that change. The class attribute is orthogonal and has a value of either *perfective, adaptive,* or *preventative*. This attribute is chosen first because, in the future, it will be likely to provide a basis for constraining the values of the other attributes. For example, as we gather more data, it may become clear that an adaptive change will always affect the dynamic properties of a system.

The next attribute to be determined is the *type* of modification that must take place. This attribute refers to whether information is *added* to, *deleted* from or *modified* in the artifacts of the system. Specifically, this characterization refers to code changes to methods, classes, data, or libraries [13]. Because it is possible, and likely, that more than one of these actions will occur for a given change, this attribute is non-orthogonal. The type attribute allows a developer to prescribe a solution for the change by identifying what must be done to the code.

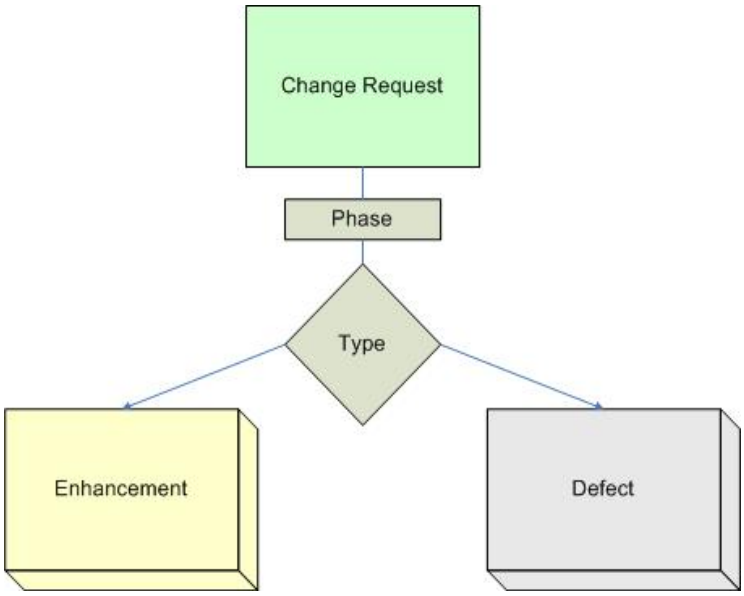

Figure 1 High-Level Change Categories

The next attribute that is important for an enhancement is the *impact* attribute. This attribute takes advantage of the developer's knowledge of the system to associate the change with the area(s) of the system that will be affected. Because the parts of the system affected by the change are more likely to have new defects in them, the value(s) of this attribute help testers focus their regression testing efforts to ensure that the system will still function correctly after the change [5]. There were various software impact areas identified in the literature that could be affected by a software change including: *processes, hardware, data, system, protocols/standards, programs/sub-systems, file systems, interfaces, documentation,* and *source code* [1, 3, 20, 22, 27]. It is expected that the design of a system will influence which aspect of a system are affected by a particular change [5].

After determining the impact, the next step is to describe the *properties* of change as either *static*, *dynamic*, or a combination of the two. A value of *static* indicates that the change affects the static structure of the system, i.e., class, package, and object diagrams, while a *dynamic* value indicates an effect on the dynamic properties, i.e., collaboration, state-chart, and activity diagrams.

The last attribute to be determined is *architectural* **vs.** *functional* characteristic of the change. For this attribute, we use a scale ranging from purely architectural changes that affect the structure of a system, but not how it functions to purely functional changes that affect the user-observable attributes or functions of a system [24]. The value of this attribute may be anywhere in the range between those two extremes, indicating how much it will affect the architecture. For this initial study, we created a primitive scale that classifies a change as purely functional (-1), purely architectural change (1), or somewhere in between (0). Figure 2 shows the enhancement classification process graphically.

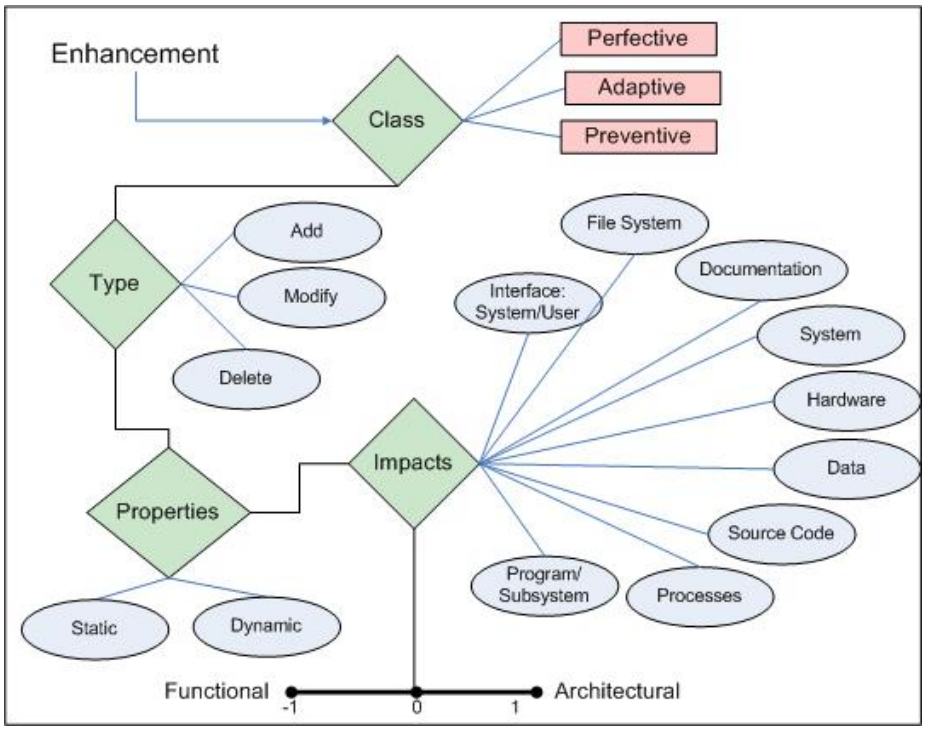

Figure 2 Detailed View of Enhancement Attributes

3.1.3 Defect Details

Similar to the enhancements, the defect category has a set of more detailed attributes. These changes focus on repairing an error, fault, or failure in the system and can therefore also affect the architecture of the system. As each defect fix will also be an enhancement, the first step is to classify the change using the attributes described in Section 3.1.2. Afterwards, the developer will classify the change using the additional attributes needed for defects as shown in Figure 3 and described in detail in the rest of this section.

Similar to enhancements, the first attribute is the *class*. The only remaining high-level attribute identified by Mohagheghi is *corrective*, because all changes due to defects are *corrective* by definition [23]. The class attribute is included in the initial version of

the classification scheme for completeness. However, if no additional values for this attribute are discovered in future research, the class attribute may be removed for simplicity.

The next attribute, *found*, identifies the stage in the development process in which the defect was discovered. This value is either *inspection*, *testing*, or *user-reported*. Following the same idea, the next attribute is the *origin*, or part of the documentation in which the defect originated. The defect can originate in the *requirements*, *design*, or *code*. These attributes are included because of previous research identifying a correlation between the time and location of defect discovery and the effort required to repair that defect. Changes later in the lifecycle tend to be more difficult and have a larger impact because more artifacts must be changed, requiring input from multiple stakeholders [9].

The final attribute for a defect is *issues*, which is similar to the impacts attribute for enhancements. This attribute is useful for prescribing a general means of handling the changes needed to fix the defect. Each value for the issue attribute is found throughout software engineering literature as causing defects and are as follows: *design, data accessibility, environment, problem definition, domain knowledge, technology, interface: system/user,* and *data transmission* [9, 10, 21].

This initial architectural change classification scheme will be continually refined based on the results of studies similar to the one described in Section 5.1.

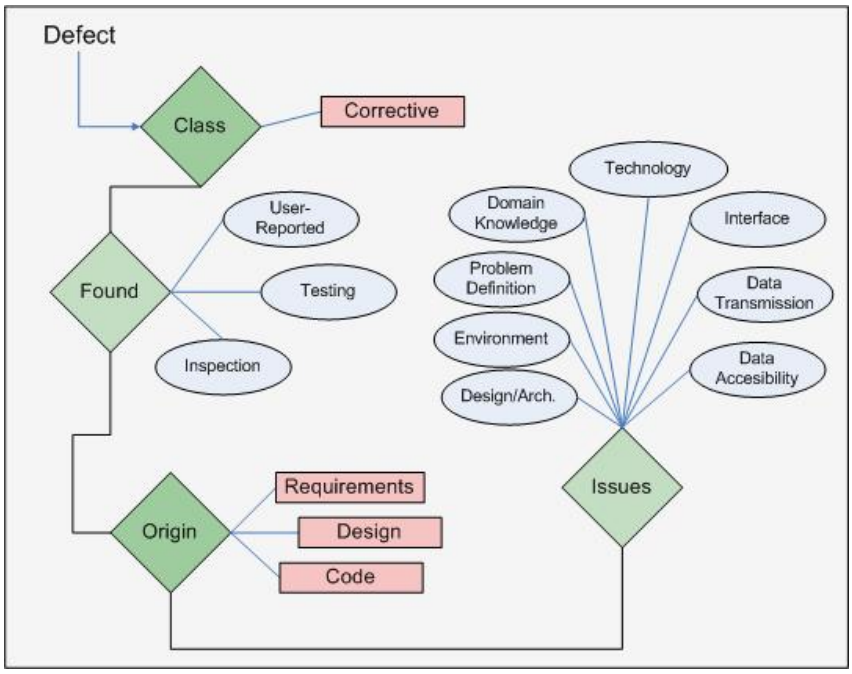

Figure 3 Detailed View of Defect Attributes

CHAPTER IV

HISTORICAL DATA ANALYSIS

The second goal of this research focuses on the analysis of historical data to further refine the created change classification scheme. This refinement occurred by classifying historical change requests in a single dataset and analyzing the implementation data to identify trends in each change class. The classification scheme from Section 3.1 is also subject to change with the identification of new categories, updating existing categories, or removing categories.

To do this type of analysis, this historical dataset must include change requests and implementation data that can be traced back to each change request. It must also include a way to measure the effect of each change request. An ideal dataset would contain the following information on changes:

- Effort required to make the change
- Calendar time to complete change
- LOC changed/added/deleted
- Modules changed/added/deleted
- Artifacts changed
- Origin of error
- Source of change request

No dataset has been found that contains all of this information. A suitable dataset should contain enough of these features that allow effort to be measured. These different effort measurements could then be used as inputs to determine the difficulty of the change.

One repository of software development projects that includes change requests and implementation data is sourceforge.net® (SourceForge). SourceForge is an open source software project repository that contains over 100,000 registered projects and over 1 million registered users. Each project's development status scale ranges from planning (1) to mature (6) and includes a rating for inactive (7) projects. For each project, an automated tracker keeps up with feature requests (change requests), bug fixes, and a CVS repository of the code and available documentation. There are also forums that contain details and user/developer comments for each area managed by the tracker. These forums include status information for each change request and bug listing for the selected project. The status information details whether the change request is open, closed, or pending.

An example from the SourceForge repository is shown in the figures below. Figure 4 shows a screenshot of the SourceForge CVS web-based repository viewer. This page shows the project dataset that was used during the refinement phase, NSIS or Nullsoft Scriptable Install System, which is a script-based Windows® installer system. This project was chosen because of the detail the development team used in documenting code changes in the CVS repository. This information, along with the SourceForge forums, was used to trace an enhancement requests back to the actual code changes. Figure 4 also shows the changes made to *clzma.cpp* a source file that was changed twice

since Revision 1.1 of the NSIS system. For each revision, the changes made to the source file along with a description of the change are displayed. The repository also includes the LOC changes from one revision to the next. The SourceForge repository contains data suitable for metrics M5 (detailed change request information), M6, (classification of changes from historic data sets), M7 (lines of code added, modified, or deleted per change class), and M8 (number of modules changed, modified, or deleted per change class). This information will be gathered by discreet examination of the classes that have been changed.

Figure 4 CVS Repository View – NSIS

Another screenshot, shown in Figure 5, shows the use of the *diff* tool from one revision of the *clzma.cpp* to the next. This tool can be used to see what types of modifications were made to the code in addition to the change in LOC.

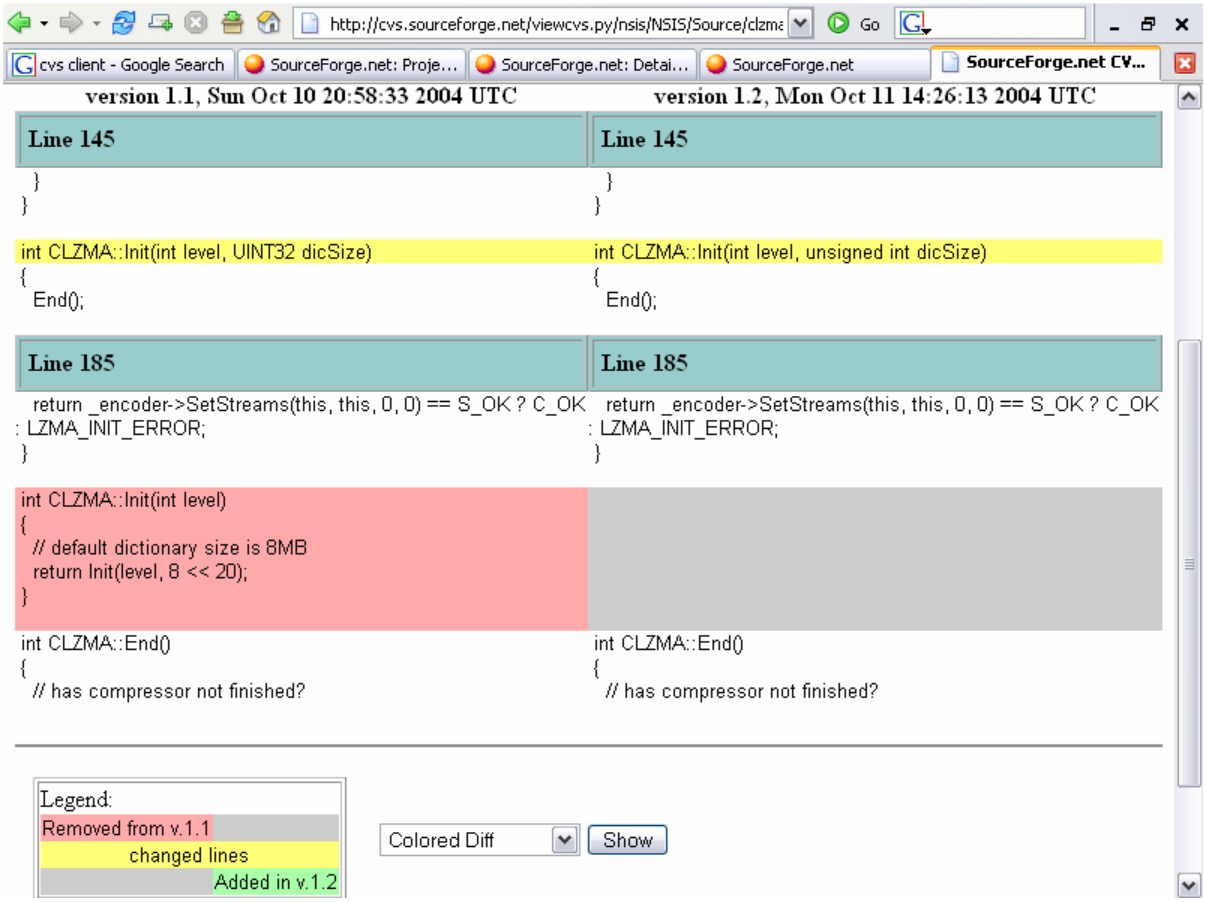

Figure 5 CVS Diff Tool

The last SourceForge screenshot in Figure 6 shows an example change request made for the NSIS project. Requests such as this one were used as inputs to the classification scheme and the implementation data that it matches in the CVS repository was used for analysis.

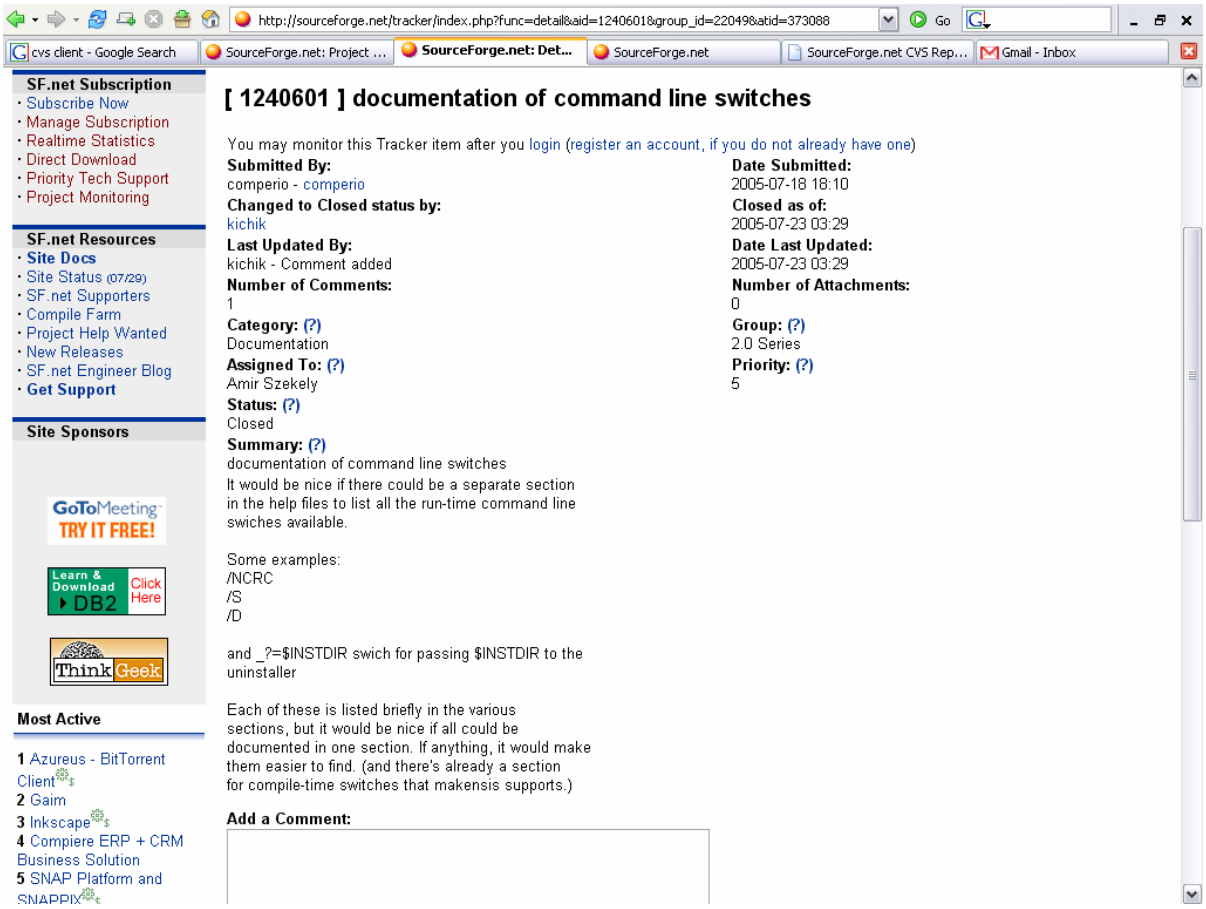

Figure 6 NSIS Closed Change Request

CHAPTER V

EMPIRICAL CLASSROOM STUDY

The observation phase of this research was done via an empirical classroom study. This study will use the change classification scheme and will provide evidence either in support of or against the hypothesis. The details of the study are listed in the following subsections.

5.1 Study Description

The overall goal of this study was to gain insight into the use of the architecture change classification scheme. The GQM goal for this study is:

> G3 Analyze the software change classification scheme in order to characterize it with respect to usability, effort prediction and architecture impact estimation from the point of view of the researcher in the context of a classroom study

Our purpose for running the experiment was to determine if the classification scheme described in Section 3.1 would be of any practical use to a developer making a software change. We focused on architectural impacts in this study because architectural changes tend to have an adverse effect on system quality when implemented without taking the necessary precautions to prevent degradation. The questions of interest are as follows:

> Q9 Did the researcher classifications match the subjects' classifications?
>
> Q10 Is it the scheme easy to use?

Q11 How accurate are the effort predictions?

Q16 Are the groupings/categories for the changes logical?

Q17 Are there any categories missing from the scheme?

Q18 How can the scheme be improved?

Answers to these questions give us an idea about whether students understood the scheme and help identify its strengths and weaknesses. To address these questions we collected both qualitative and quantitative data. The qualitative data was obtained from questionnaires and surveys as well as from reports the student's submitted about their experience. The quantitative data was provided by the students keeping track of the implementation details for each change, including the number of modules and components changed and the amount of time required. Finally, we collected the modified architecture and code from each student to perform our count of modules changed and LOC changed for each change request.

5.2 Hypothesis

One fundamental requirement for using a classification scheme to make a prediction about the effort required to make a change is that changes from different classes must require different amounts effort to implement. If different classes of change do not require different amounts of effort, then the classification scheme is not useful as an effort predictor. The other fundamental aspect of a valid classification scheme is whether it matches the general understanding of the developers. Different developers should have a similar notion of how to classify a given change in terms of its effect on the system.

The architecture change classification scheme was designed to force the developers to understand the impact of a change prior to implementation. After classifying the change, the scheme should provide information useful to an impact model that helps to determine the effect of the change on the code and the required amount of effort. Based on these requirements, our research hypotheses are as following:

H0: Developers will consistently classify changes in the architecture change classification scheme

H1: Changes of different classes will require different amounts of effort to implement.

5.3 Study Setup

This study was conducted in the Software Architecture and Design Paradigms class at Mississippi State University in the Fall 2005 semester. This class focuses on software architecture development methodologies and analysis methods including model representations, component-based design, design patterns, and frameworks. There were 18 subjects (13 seniors and 5 graduate students) who participated in the study.

During the experimental tasks, described in Section 5.5, the students worked with the artifacts from the Tactical Separation Assisted Flight Environment (TSAFE), a tool designed to aid air-traffic controllers in detecting and resolving short-term conflicts between aircraft [7]. Prior to the beginning of the study, the students were given several weeks to familiarize themselves with the TSAFE system by completing an assignment to create their own architecture for the TSAFE system based on the given requirements document. This assignment required the students to create the logical and runtime

structure, map that information to the hardware and file system, and provide a rationale for their decisions.

5.4 Training

Before using the classification scheme, the students were given two 1.25 hour class periods of training. The first session focused on explaining the purpose for the classification scheme and defining the attributes. The second session allowed the students to get hands-on practice using the classification scheme to ensure they understood it. During this session the students were given descriptions of 10 change requests for a fictional system (for which they were given requirements and architecture), and asked to classify those change requests. This session ended with a discussion of the classification scheme to allow the students to provide feedback that helped us clarify some definitions.

5.5 Experimental Tasks

The change classification study took place during the final homework assignment of the semester. The students were given feedback on the TSAFE architectures that they created, and then were given the "gold standard" TSAFE architecture created by the researchers. The students were given 2 successive change requests of different types to implement. The order in which the students received the change request was randomized. The change requests were designed to be complex enough to require architectural changes while being simple enough to be completed in the allotted time. To verify these properties, we implemented both change requests prior to the assignment.

After receiving the change request, the students first used the architecture change classification scheme to classify it. Next, they modified the architecture to allow the

change to be implemented. Finally, the students implemented the change in the codebase. During these activities, the students were asked to keep track of how many modules they changed and the amount of time they worked.

After completing and submitting the first change request, the students were told to return to the original version of the architecture and code (i.e., not including Change 1) and were given their second change request. The students were given 3 weeks total to complete both changes.

After completing both changes, the students were given a survey to gather their opinion about the difficulty of each change, the ease of using the change classification scheme, and whether the change categories "made sense" to them. Here are the steps the students followed to complete the assignment:

1. Review the "gold standard" TSAFE Architecture and corresponding code
2. Classify change request using architectural classification scheme
3. Provide rationale for classification in Step 2
4. Submit change request classification and rationale
5. Make changes to TSAFE architecture and document effort
6. Submit modified architecture
7. Make changes to TSAFE code
8. Submit code changes and any additional architectural changes made
9. Provide report detailing changes made
10. Repeat steps 1-9 for second change
11. Complete and submit survey

5.5.1 Change Requests

Change #1 – Conformance Monitor. This change involved adding a class to the system to calculate whether the flights managed by TSAFE are staying on their set courses. The students were given the code for the mathematical calculations, but to correctly implement the change, the students had to determine where the class should connect to the system and how the calls should be made to the member functions.

Change #2 – Feed Display. This change required adding additional functionality to the TSAFE system to allow the GUI to display raw flight coordinates data. This change required the students to transfer data from low-level classes that handled the raw flight data up the class hierarch through several classes to the higher level display module.

Table 2

Researcher Classifications

Enhancement		Change 1	Change 2
Type:	Add	1	3
	Modify	1	2
	Delete	0	1
Properties	Static	2	1
	Dynamic	2	2
Impacts	Program/Subsystem	0	0
	Processes	0	0
	Interfaces	1	3
	Source Code	1	2
	File System	1	0
	Hardware	0	0
	System	0	0
	Data	0	2

	Documentation	1	1
	Other	0	0

Using the classification scheme from Section 3.1, we classified the changes (prior to implementing them in the code). Both changes were perfective enhancements that occurred during the maintenance phase of development, and Table 2 shows the remainder of the attributes. The numbers in Table 2 correspond to the impact scale in Section 3.1.

5.5.2 Data Collection and Analysis

The data gathered from conducting the empirical classroom study must be statistically and objectively analyzed to provide sound evidence in support or against the research hypothesis.

There were several questions of interest associated with each goal of the research. These questions have been answered using both quantitative and qualitative metrics. The changes presented in the classroom study were classified by the researchers as well as the subjects for comparison. This data will satisfy metric M9 (classification of changes by researcher and subjects) identified in Section 1.3.3. Several comparison charts were made to assess the degree of similarity between each student's classification and the researcher's classification. This information tells us how well the students understood the attributes of the classification scheme. The charts also tell us how consistently different developers classify the same change request.

For the qualitative measurements, an implementation survey that asks questions pertaining to the difficulty of implemented each change was provided to the students as well as a classification survey on the ease of using the classification scheme. These two

pieces of information will satisfy metrics M10 (change implementation survey) and M11 (change classification survey). Several examples of questions from each survey are show below:

Implementation Survey:

1. Which change request was more difficult to implement? Why?

Classification Scheme Survey:

1. What additional change attributes would you recommend adding to the classification scheme?

2. On a scale of 1 to 5, with 1 being the easiest and 5 being the most difficult, rate the scheme in terms of its ease of use.

3. Are there any inconsistencies in the attributes of the scheme and your experiences in software development?

Survey questions such as these provided qualitative feedback to the researchers. The detailed surveys can be found in the Appendix.

The person-hours required making the changes, along with lines of code, and modules changed were recorded by the subjects and included in the reports. This piece of information will provide metrics M12 (person-hours required to make the change), M13 (LOC added, modified, deleted), and M14 (modules added, modified, deleted). This data was analyzed for each change request. Finally, the effort (LOC) required to implement the changes from different classes will be compared using a t-test to determine if the change that was hypothesized to be more difficult really was more difficult in practice.

CHAPTER VI

RESULTS

6.1 Historical Data Analysis Results

The SourceForge repository provided an extensive selection of open source projects. Each project contained similar information relating to the implementation of changes and bug fixes, and it allows users to track when changes are made to each system. The NSIS project was chosen from a random search of the available projects in the SourceForge repository because of the perceived level of detail provided by the project developers. The amount of detail provided for each change made to the system in the NSIS project did appear to be at a greater level of detail than the other projects in the repository. However, there were also many cases where the implementation detail of the actual change requests listed in the system tracker could not be analyzed due to the lack of specific change information relating to the change request.

When we were able to find change requests that could be traced to the code, there existed the possibility of bias in the selection process, because not all project developers provided the level of detail needed to trace the change requests to the implementation change in the code. Analyzing a change request and finding every place in the source code where the request required a change was a very difficult and time consuming process. Due to time constraints, the NSIS repository could not be exhaustively examined

for change implementation detail. There were a limited number of change requests that had identifiable implementation detail; and there was a possibility that for those change requests, there may have existed implementation detail that was not found. Although the historical data analysis component of this research did not provide strong evidence of implementation effort for the various change attributes because of the problems described above, effectively using the classification scheme to classify real-world changes did provide a sanity check. We were able to classify the changes using the scheme, but in many cases, were not able to identify implementation detail.

This historical data analysis did enable us to change the original design of the change scheme to more of a decision tree-like process. The attempts we made to characterize the change requests were easily done by traversing through a tree of steps rather than just selecting any of the attributes at random.

We plan to continue using the scheme to analyze historical data. Other repositories have been identified in the open-source communities (Apache Software Foundation) that provide change implementation detail with many of their sponsored projects.

6.2 Classroom Study Results

We analyzed the data using quantitative analysis for the data collected during the change implementation, and qualitative analysis for the surveys and reports. We grouped the results based on hypothesis H0 and H1 posed in Section 5.2. Along with each hypothesis, we provide a series of observations drawn from the associated data analysis.

6.2.1 H0 Results

H0 was that the students would classify the changes consistently. Based on the data collected, we provide three observations related to that hypothesis.

Observation 1: The students classified the changes similar to the way the researchers classified them

This observation is based on the analysis of data submitted by the 15 students who submitted their classification for Change #1 and Change #2. The students did the classification prior to making the actual change. If the students correctly understood the attributes of the classification scheme and how changes to the TSAFE system would affect those attributes, then the students' change classifications should have been similar to the researcher's classifications. |

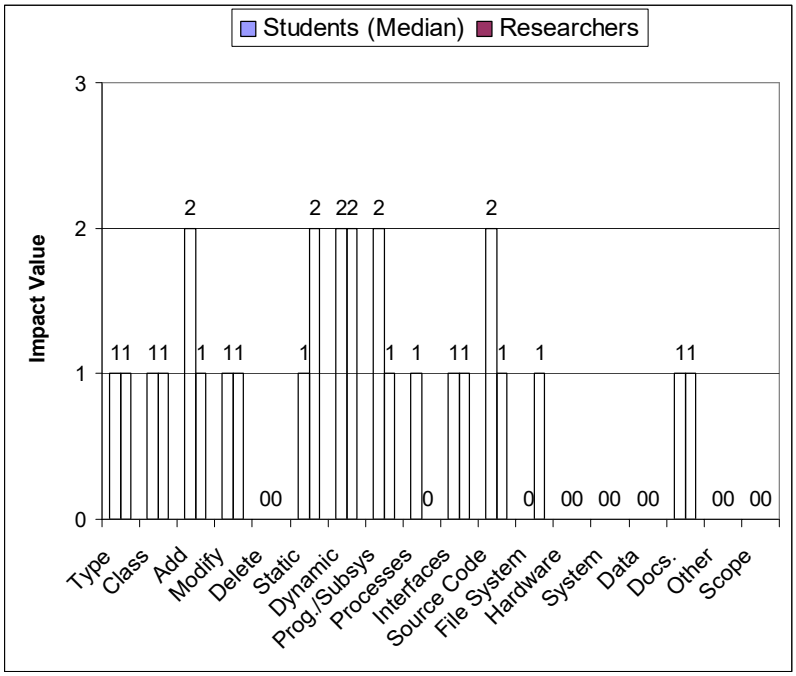

Figure 7 Classification Accuracy (Change #1)

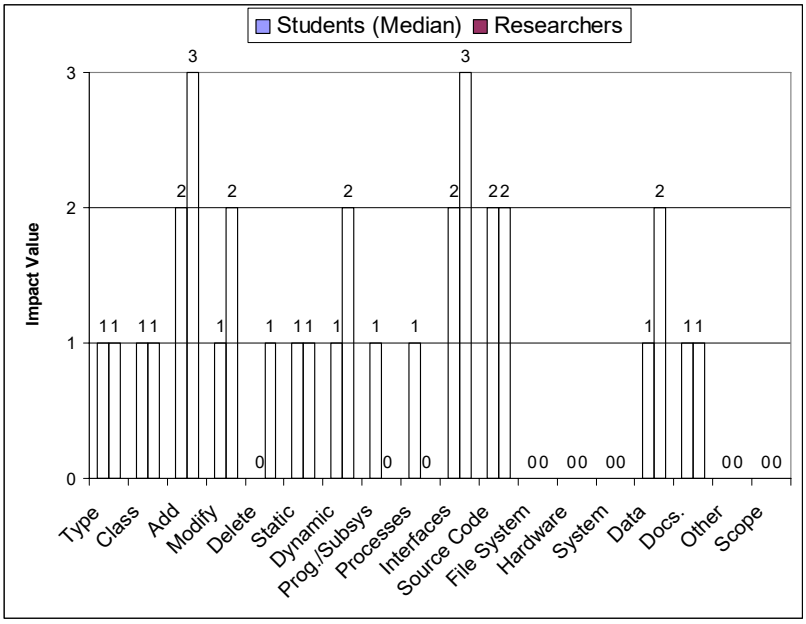

Figure 8 Classification Accuracy (Change #2)

We measured the student classifications in terms of consistency and accuracy. Figure 7 and Figure 8 show how closely the median of the student's classifications match the researchers for Change #1 and Change #2 respectively. In addition, Figure 9 and Figure 10 show the consistency of the students in their classification of the changes. The data in these charts show first, that because the students' classifications were similar to the researcher's, the students understood the classification scheme. Secondly, the results show that the students had a fair understanding of the level of impact each change would have on the TSAFE architecture and source code. Finally, while the median values of the students' classification were similar to the researcher's classification, they were not totally consistent as seen by the distribution of rankings. Additional charts created from

data from the training exercises showing the accuracy and consistency of 10 of the fictional change requests are shown in Appendix A.

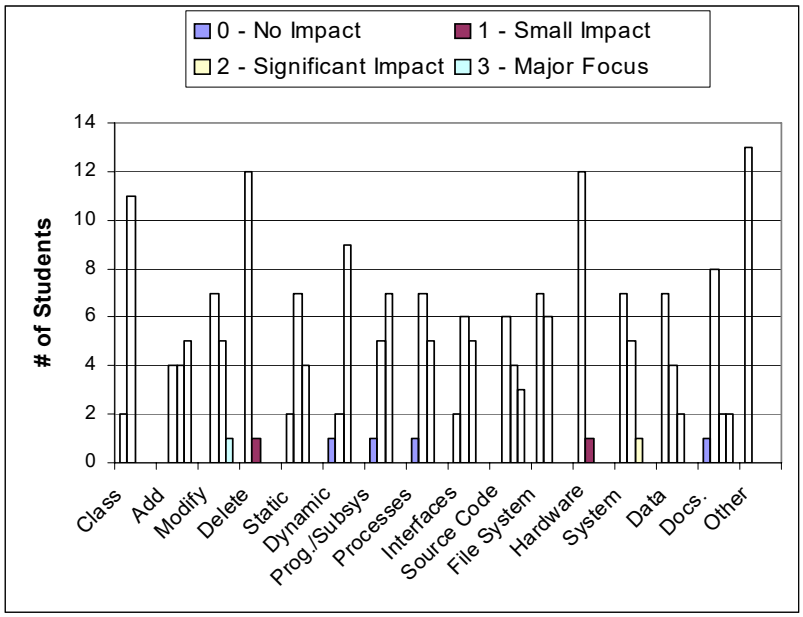

Figure 9 Classification Consistency (Change #1)

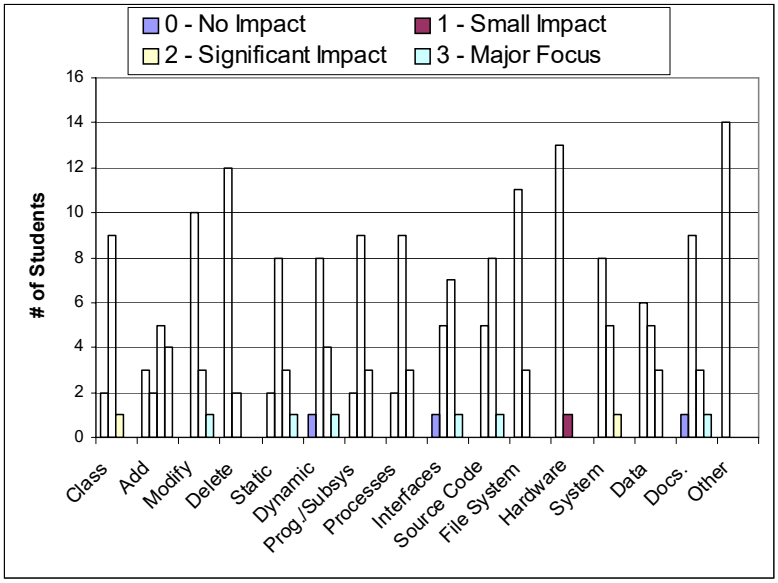

Figure 10 Classification Consistency (Change #2)

Observation 2: Survey results show that the change scheme was viewed as being useful

On the end of the assignment survey, we asked the students to describe any inconsistencies they found in the classification scheme and to tell us how it could be improved. We also asked the students to give us their level of agreement with several statements pertaining to the usefulness of the scheme. The statements read as follows:

- The attributes are logical and easily understood
- The scheme would be beneficial to a developer making a change
- The scheme has practical application in industry
- The scheme is easy to use
- After classifying both changes, I had an idea of which would be the most difficult to implement

The rating scale ranged from 1 (totally disagree) to 5 (totally agree). Figure 11 shows the results recorded for each student in the study.

These results give us some confidence in the usefulness and practicality of the classification scheme. For the first three statements, there was only one student who showed any level of disagreement. The fourth and the fifth statement had more disagreement than the first three statements. Observation 3 provides some rationale as to why the “easy to use” ratings were low.

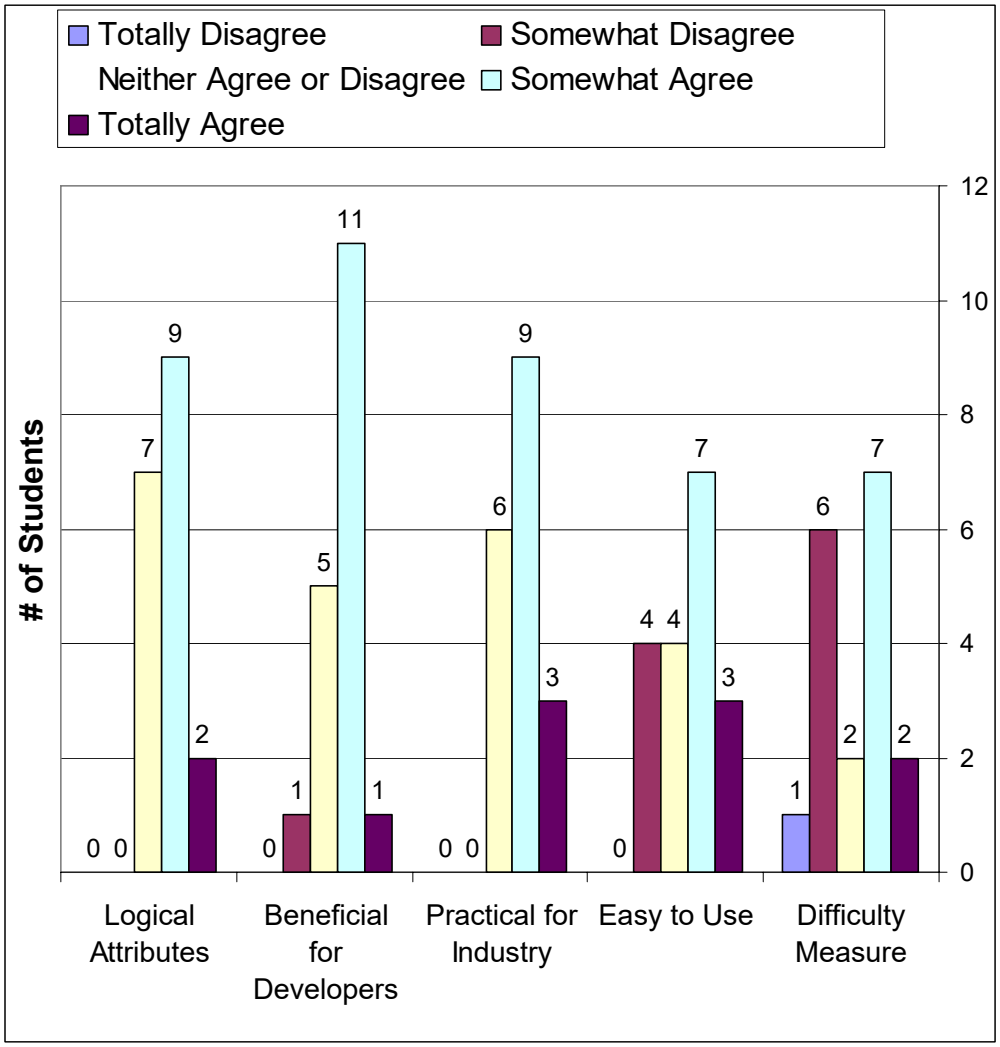

Figure 11 Student Survey Results

Observation 3: The classification scheme contains some overlapping attributes and ambiguous definitions

One of our survey questions asked the students to identify any ambiguity or inconsistencies in the definitions provided for the attributes of the classification scheme. This question was meant to elicit information that could be used to improve the

classification scheme. The answers that were given by the students along with a count (in parentheses) of the number of students that gave the response are shown below:

- Need a more clear definition of "System" for the Enhancement changes – could be system interface (5)
- Need a more granular scale for the "Functional vs. Architectural" attribute (5)
- Definitions of "Adaptive" and "Perfective" are too similar (4)
- Program/Subsystem attribute is not needed (2)
- "Documentation" and "Source Code" are the same attributes (2)
- "Corrective" class not needed for Defects category (2)
- "Interface" definition should be split into separate definitions for system interfaces and GUI interface (1)
- Should define the scale used for impact measurement of each attribute by providing examples of changes to similar systems and the results the chosen impact value had on the system (1)
- Need a clearer definition of "Static" and "Dynamic" properties (1)
- Need a rating for number of modules that will be affected by the change (1)
- "Requirements" should be added to the Issues section of the Defect category (1)

The confusion with some of the attributes, illustrated by the above comments, provides some insight into why the classification scheme was not viewed as being as "easy to use" as we would have hoped. They also illustrate why many students believed it would be difficult to predict which change would require more effort simply based on its

classification. These comments reflect improvements that will be made to the classification scheme to improve its use in future studies.

6.2.2 H1 Results

Observation 4: Change #2 (Feed Display) required more effort than Change #1 (Conformance Monitor)

Although most students were not able to completely implement both changes, a majority (12/18) stated that Change #2 (Feed Display) required more effort and was more difficult to implement than Change #1 (Conformance Monitor). Out of the three students that did successfully implement both changes, all indicated that Change #2 was more difficult to implement.

This qualitative response is explained in part by the difference in classification of the two changes. Change #2 was classified as having a larger effect on the source code, and required a larger amount of code to be added to the system.

In addition to the qualitative responses, for each change that was completed, we used a tool to compute the number of modules changed and the number of LOC changed. The data came from three students who completed both changes, one student who completed only Change #1, and one student who completed only Change #2. Of the five students that completed a change request, three received Change #1 first, the other two received Change #2 first. Due to the small number of data points, we provide the statistical results for both the parametric t-test and the non-parametric Mann-Whitney U test.

Table 3

Change Implementation Results

Subject	Change	LOC Changed	Modules Changed
S5	C1	19	2
	C2	25	7
S10	C1	19	2
	C2	N/A	N/A
S13	C1	N/A	N/A
	C2	13	3
S14	C1	16	2
	C2	36	8
S17	C1	11	2
	C2	37	7

The results of this analysis showed that that the subjects changed more modules when implementing Change #2 versus Change #1. This result was statistically significant (t_4 = -16.00, p = .004 [t-test]; Z = 2.121, p = .034 [Mann-Whitney]). Furthermore, Change #2 also required more LOC to implement than Change #1. This result was again significant (t_4 = -3.854, p = .018 [t-test]; Z = -1.964, p = .04 [Mann-Whitney]). The number of modules and LOC changed are shown in Table 3.

The statistical results coupled with the qualitative responses from the subjects provide some evidence that if used correctly, the classification scheme provides some help in comparing the difficulty of changes.

6.3 Study Implications

Only three of the eighteen students that participated in the study completed both change requests, with two students completing one change request each. To better understand this occurrence, we attempted to identify the difficulties that the students had

while making the changes by analyzing the report each student provided about each change. There were several factors that may account for the low completion percentage. Here we list four prevailing factors from the student reports that may account for the low completion percentage.

TSAFE is too complex to understand and change in such a short time: The students had a difficult time understanding the TSAFE code. This was the biggest obstacle noted in their reports. The TSAFE source code contains several packages and over 100 source and configuration files. The average analysis time, time spent understanding the architecture and source code, for students making their first change was 5 hours. Most students struggled to understand the system and could not sufficiently perform the required tasks.

No design document to show how the architecture maps to code: We only provided the students with the TSAFE requirements, architecture, and source code. Since we did not provide the students with a low level design document for the system, the students had trouble figuring out exactly how to map the architecture to the source code in some cases.

Java development experience low: Several students stated that they had to "relearn" Java because they had not used the programming language in "quite some time."

Did not provide enough motivation to implement challenging change requests: The students had a total of 3 weeks to complete both changes late in the semester. The assignment only accounted for ~5% of the students overall grade, but was more difficult

than previous homework assignments. The students did make attempts to implement each change, but with finals approaching and other semester long projects coming to completion, the architecture change assignment may not have been the highest on their priority lists.

6.4 Threats to validity

We have identified a set of validity threats both with the design and the execution of the study. In some cases we were able to address potential threats prior to the study, while in other cases the threats remain. In this section we discuss those threats and their potential impact on our results.

6.4.1 Threats Addressed

Maturation and Testing: We randomly gave half of the class Change #1 first and the other half Change #2 first. We did this to negate any effects of maturation where students would naturally perform better when implementing the second change after having the experience of implementing the first. Variables such as analysis time and implementation time, given equal changes, would also have been less for the second change because the student would have a better understanding of the code the second time through.

6.4.2 Threats Not Addressed

Small Sample Size: The biggest threat to the validity of this study was the small sample of students that completed both change requests. We took this issue into account when running statistical tests by using both the parametric and nonparametric tests, which

showed similar results. But it is still problematic to draw any solid conclusions from this data.

Effort in Man-Hours not Collected: We were not able to measure the amount of time required to make each of the changes. While the students were asked to report this figure, they were not given a time sheet to use to track their effort to analyze the code, modify the architecture, and modify the source code. Therefore, most students (including the three that completed both changes) did not report this information and our comparisons between the changes were based on an estimate of effort (number of modules and LOC changed).

Change Differences from Other Factors: The different amounts of effort required to implement the two changes could result from other factors besides the difference in classification. Because we only had 2 changes, we were unable to investigate other possible differences that could cause the dissimilarity in the amount of effort required to implement each change. In Chapter 7, we describe a future study that will address this issue.

CHAPTER VII

CONCLUSION

Our initial goal for this research was to use the architecture change classification scheme to test its usefulness in allowing developers to conceptualize the affects a change would have on a software system. We hypothesized that different classifications would require differing amounts of effort to implement allowing the change classification scheme to be used for effort prediction. Predicting software change effort is a difficult task even for experienced developers [19]. Our purpose for creating this scheme is to provide an input to a decision support model that will incorporate change classification, impact analysis, and risk assessment to aid developers in making go/no-go decisions for changes based on how the system will be affected.

The results we were able to obtain did, in general, support our initial hypotheses. That is, Change #2 required significantly more modules and LOC to be changed than Change #1; and the subjects qualitatively agreed that Change #2 was more difficult. Furthermore, the students responded that the in most cases the classification scheme was useful, although they did provide some helpful suggestions for improvement.

We will continue to refine the classification scheme by making several changes based on the feedback from the students. We will continue to use the modified scheme to classify change requests from historical datasets that provide sufficient change

implementation detail and correlate those changes to the implementation data to gather more data about the relationships of the change attributes. We hypothesize that differing classifications of changes for a single system will exhibit distinguishing trends from other change classes. We expect to identify certain attributes of a change that will require more implantation effort than others. We have identified another open-source software repository, the Apache Software Foundation, which contains funded open-source software projects that include detailed change data. We also plan to investigate other data sources to continue the historical data analysis efforts.

We intend to replicate the study reported on in this thesis, improving it based on our lessons learned. In future experiments, we will use multiple change requests that include a set of similar change classes and a set of different change classes. We can then compare both within classes and across classes to better understand the source of variation in effort. We can also perform further statistical analyses to show if the classification scheme does a reasonable job of predicting effort. We intend to perform further analysis on the data to examine an information-theory based metric approach to measure consensus when multiple developers classify a change and different classifications exist [12]. Our end goal for this study is to replicate it in industry. We understand the difficulties associated with running experiments using students as subjects, and running this study in an actual development environment would provide stronger evidence in support or against our hypotheses [4].

Change classification can be a useful tool in determining the impact the change will have on the system being changed. Upon further research, we envision that this

classification tool could be incorporated in to an organization's current change implementation process. An additional step could be added after receiving a change request where the system developers would classify the change request. The developers would meet to discuss any differences in classification, and use this meeting to come to a consensus on the change's attributes. After some level of agreement is reached, the developers would analyze baseline change data of similar classes of change and begin to assess the impact the current change will have on the system. A decision could then be made on whether to implement the change in the system based on the importance of the change and its classification. If the change must be made, the proper precautions could be used to prevent architectural degradation.

Being able to accurately classify changes that affect software architecture will aid developers in understanding the change impact and help them make architecture changes without degrading the quality of the system.

7.1 Publication Plan

I plan to publish this research using three different approaches. I plan to submit the information pertaining to the literature review on architecture change classification to either ACM Computing Surveys (CSUR) or Information and Software Technology, two technology journals. The Background and Motivation and Change Classification Literature Search sections of this thesis will be the bulk of the survey paper. I expect to submit this paper in May 2006. I have submitted a paper about the empirical study and its results to The 2006 International Symposium on Empirical Software Engineering (ISESE). I plan to combine the literature survey, historical data analysis, change

classification scheme, and the empirical study into a paper that will be submitted to IEEE Transactions on Software Engineering or The Journal of Systems and Software.

REFERENCES

[1]. V. Basili and D. Weiss, "Evaluation of a Software Requirements Document by Analysis of Change Data", *Proceedings of the 5th international conference on Software engineering*, San Diego, CA, 1981, IEEE Press, pp. 314-323.

[2]. V. Basili, G. Caldiera, and H.D. Rombach, *The Goal Question Metric Paradigm*, in *Encyclopedia of Software Engineering*, J.J. Marciniak, Editor. 1994, John Wiley & Sons, Inc.: New York. p. 528-532.

[3]. L.C. Briand and V.R. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process", *Proceeding of the Conference on Software Maintenance*, Orlando, FL, 1992, pp. 328-336.

[4]. J. Carver, L. Jaccheri, S. Morasca, and F. Shull, "Issues in Using Students in Empirical Studies in Software Engineering Education", *Proceedings of the Ninth International Software Metrics Symposium*, Sydney, Australia, 2003, pp. 239-249.

[5]. M.A. Chaumun, H. Kabaili, R.K. Keller, and F. Lustman, "A Change Impact Model for Changeability Assessment in Object-Oriented Software Systems", *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, 1999, pp. 130-138.

[6]. P. Clarke, B. Malloy, and P. Gibson, "Using a Taxonomy Tool to Identify Changes in Oo Software", *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, 2003, pp. 213-222.

[7]. G. Dennis, *Tsafe: Building a Trusted Computing Base for Air Traffic Control Software*, masters thesis, Massachusetts Institue of Technology 2003.

[8]. S.G. Eick, T.L. Graves, A.F. Karr, A. Mockus, and P. Schuster, "Visualizing Software Changes," *Software Engineering, IEEE Transactions on*, vol. 28, no. 4, 2002, pp. 396-412.

[9]. M. Fredericks and V. Basili, *Using Defect Tracking and Analysis to Improve Software Quality*, D.D.A.C.f.S. (DACS), Editor. 1998.

[10]. L. Hochstein and M. Lindvall, "Combating Architectural Degeneration: A Survey," *Information and Software Technology*, vol. 47, no. 10, 2005, pp. 643-656.[11]. P. Hsia, A. Gupta, C. Kung, J. Peng, and S. Liu, "A Study on the Effect of Architecture on Maintainability of Object-Oriented Systems", *Proceedings of the International Conference on Software Maintenance*, Opio, 1995, pp. 4-11.

[12]. U. Kudikyala, E. Allen, and R. Vaughn, "Measuring Consensus During Verification and Validation of Requirements", *Proceedings Supplement: 10th IEEE International Software Metrics Symposium*, Chicago, IL, 2004.

[13]. D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, "Change Impact Identification in Object Oriented Software Maintenance", *Proceedings of the International Conference onSoftware Maintenance*, Victoria, BC, 1994, pp. 202-211.

[14]. W. Lam and V. Shankararaman, "Managing Change in Software Development Using a Process Improvement Approach", *Proceedings of the 24th Annual Euromicro Conference*, Vasteras, 1998, pp. 779-786 vol.2.

[15]. M.M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," *Proceedings of the IEEE*, vol. 68, no. 9, 1980, pp. 1060-1076.

[16]. M.M. Lehman and L. Belady, *Software Evolution - Processes of Software Change*, Academic Press, London, 1985.

[17]. M.M. Lehman, "Feedback, Evolution and Software Technology", *Proceedings of the 10th International Process Support of Software Product Lines Software Process Workshop*, 1996, pp. 101-103.

[18]. M.M. Lehman and J.F. Ramil, "Towards a Theory of Software Evolution - and Its Practical Impact", *Proceedings of the International Symposium on Principles of Software Evolution*, 2000, pp. 2-11.

[19]. M. Lindvall and K. Sandahl, "How Well Do Experienced Software Developers Predict Software Change?," *Journal of Systems and Software*, vol. 43, no. 1, 1998, pp. 19-27.

[20]. N.H. Madhavji, "The Prism Model of Changes", *Proceedings of the 13th International Conference on Software Engineering* Austin, TX, 1991, pp. 166-177.

[21]. J.F. Maranzano, S.A. Rozsypal, G.H. Zimmerman, G.W. Warnken, P.E. Wirth, and D.M. Weiss, "Architecture Reviews: Practice and Experience," *IEEE Software*, vol. 22, no. 2, 2005, pp. 34-43.

[22]. A. Mockus and L.G. Votta, "Identifying Reasons for Software Changes Using Historic Databases", *Proceedings of the International Conference on Software Maintenance*, San Jose, CA, 2000, pp. 120-130.

[23]. P. Mohagheghi and R. Conradi, "An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality Vs. Quality Attributes", *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE '04)*, 2004, pp. 7-16.

[24]. J. Nedstam, E.A. Karlsson, and M. Host, "The Architectural Change Process", *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE '04)*, 2004, pp. 27-36.

[25]. N. Nurmuliani, D. Zowghi, and S.P. Williams, "Using Card Sorting Technique to Classify Requirements Change", *Proceedings of the 12th IEEE International Requirements Engineering Conference*, 2004, pp. 240-248.

[26]. J.F. Ramil, "Laws of Software Evolution and Their Empirical Support", *Proceedings of the International Conference on Software Maintenance*, 2002, pp. 71-71.

[27]. X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley, "Chianti: A Tool for Change Impact Analysis of Java Programs", *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications* Vancouver, BC, Canada, 2004, ACM Press, pp. 432-448.

[28]. R.T. Tvedt, M. Lindvall, and P. Costa, "A Process for Software Architecture Evaluation Using Metrics", *Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, 2002, pp. 191-196.

APPENDIX A

CLASSROOM CLASSIFICATION CHARTS

Appendix A includes charts that represent the classification accuracy (student's classification compared to the researcher) and classification consistency (number of students that selected a particular impact value) of the change requests classified during the training exercises. It also includes the sample change requests used in the training sessions. The students were given 10 change requests, which are provided following the charts, and asked to classify each change request using the architecture change classification scheme. The change accuracy charts show the median impact value selected by the students and the value of the researcher's classification. The change consistency charts show the number of students that selected each impact value. These charts support Observation 1, described in Section 6.2.1.

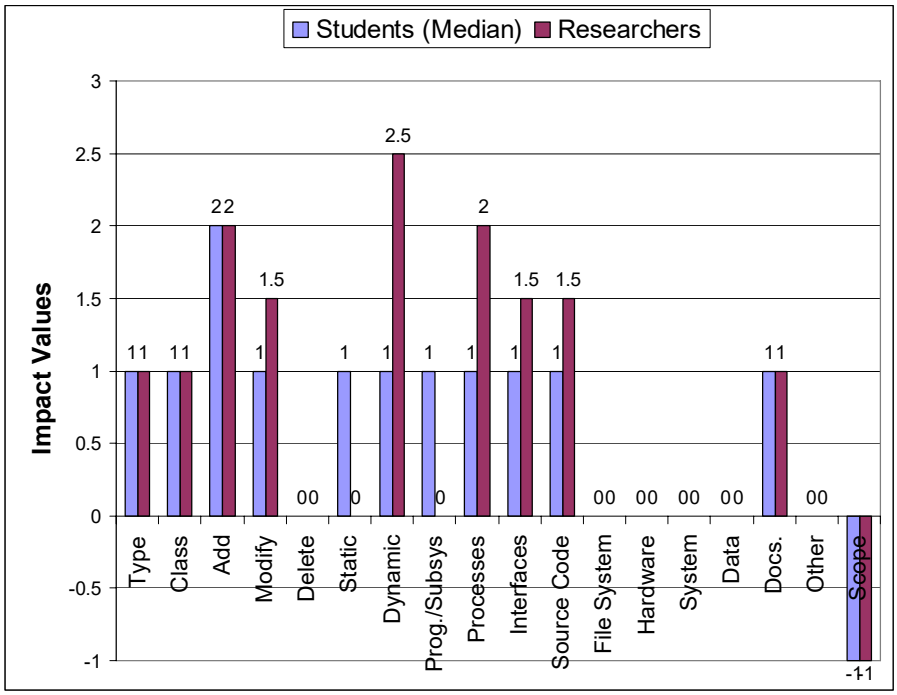

Figure 12 Sample Change #1 Accuracy

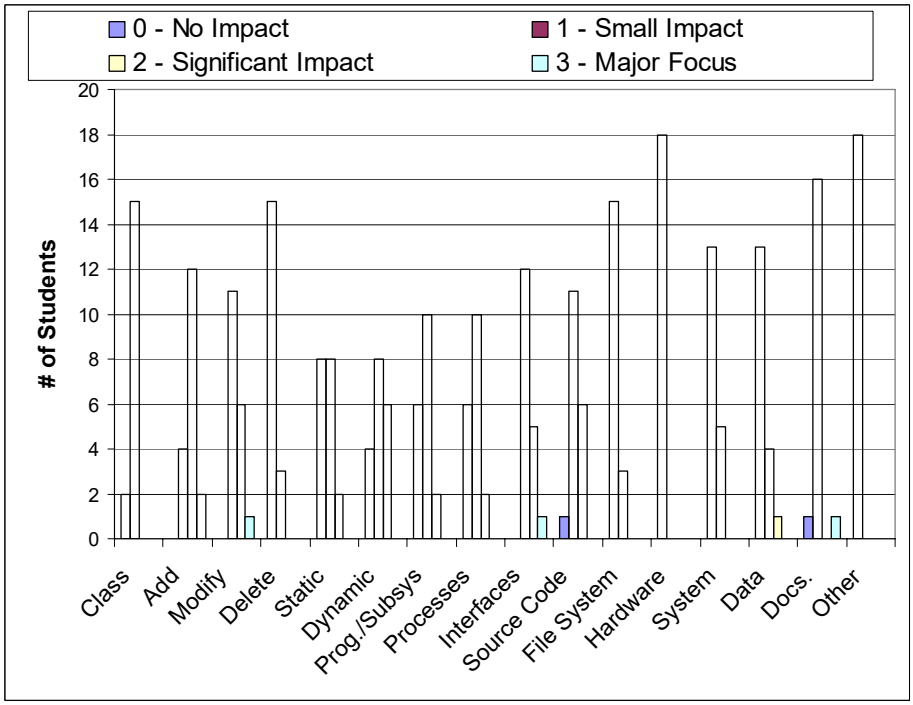

Figure 13 Sample Change #1 Consistency

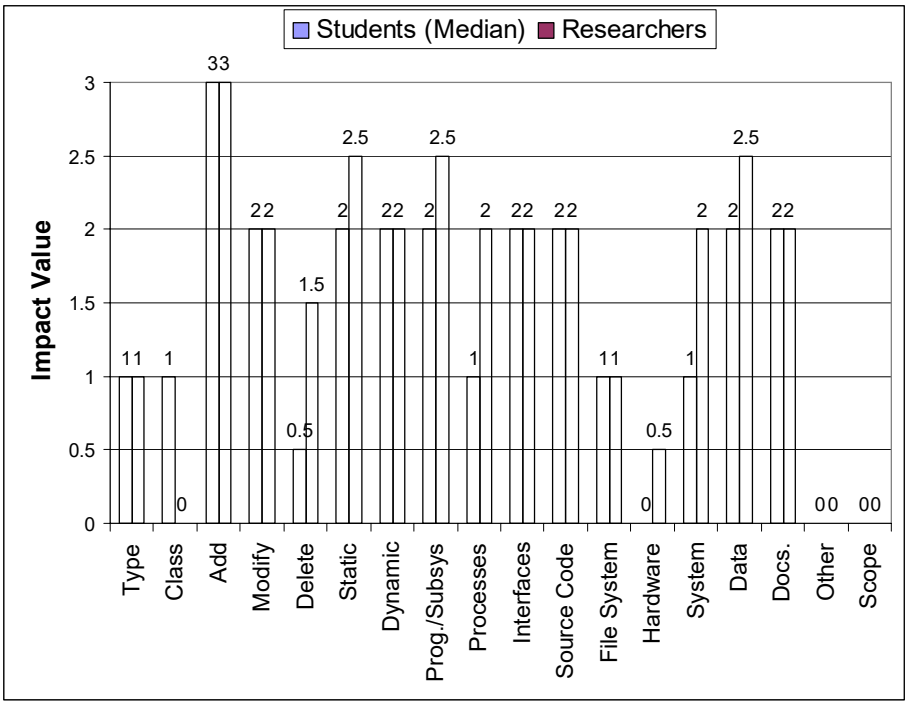

Figure 14 Sample Change #2 Accuracy

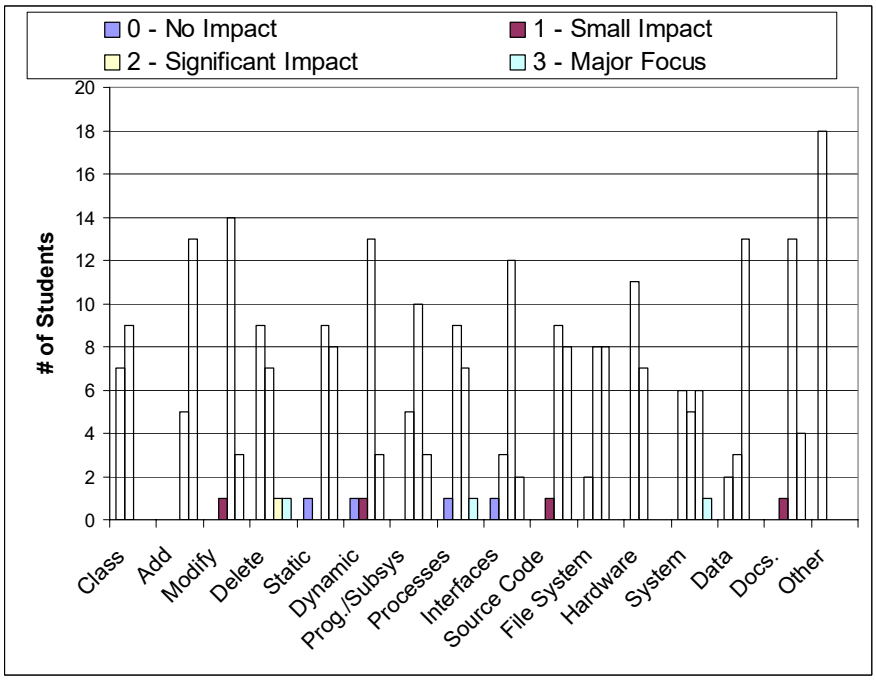

Figure 15 Sample Change #2 Consistency

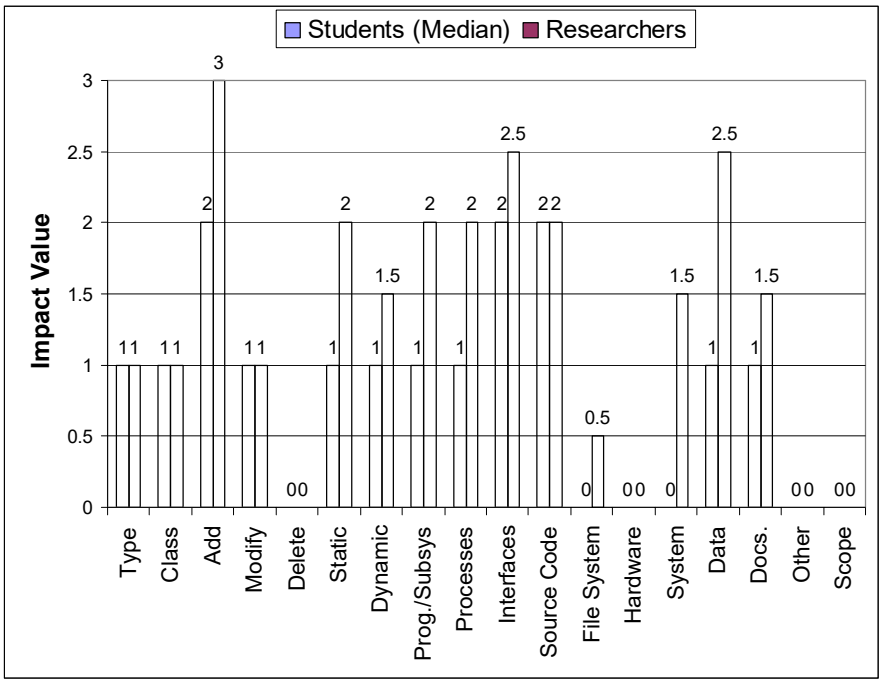

Figure 16 Sample Change #3 Accuracy

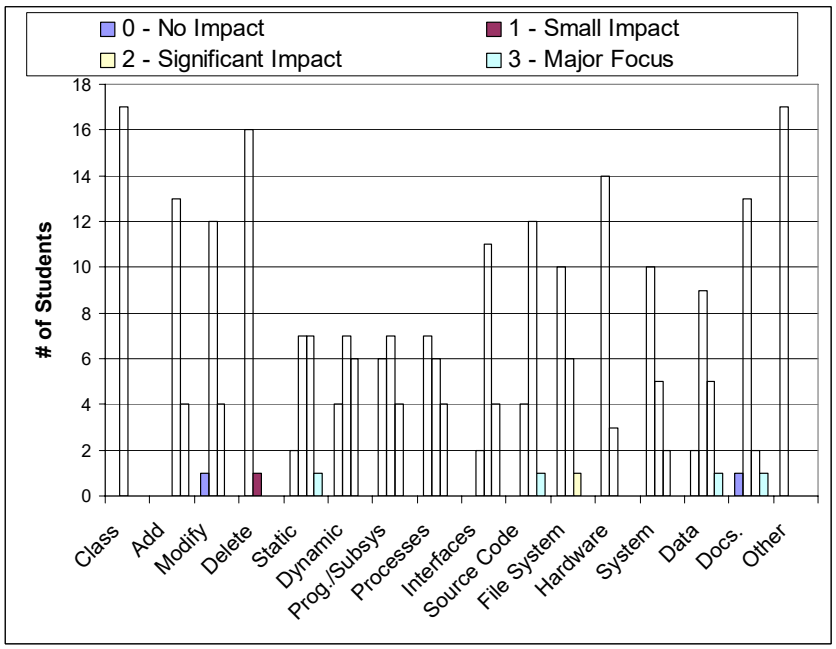

Figure 17 Sample Change #3 Consistency

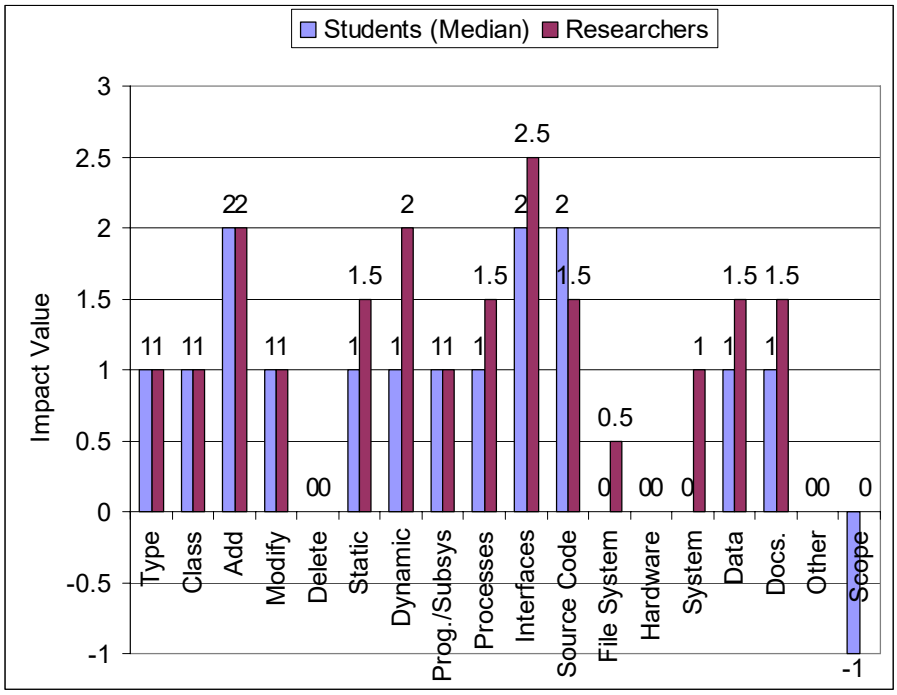

Figure 18 Sample Change #4 Accuracy

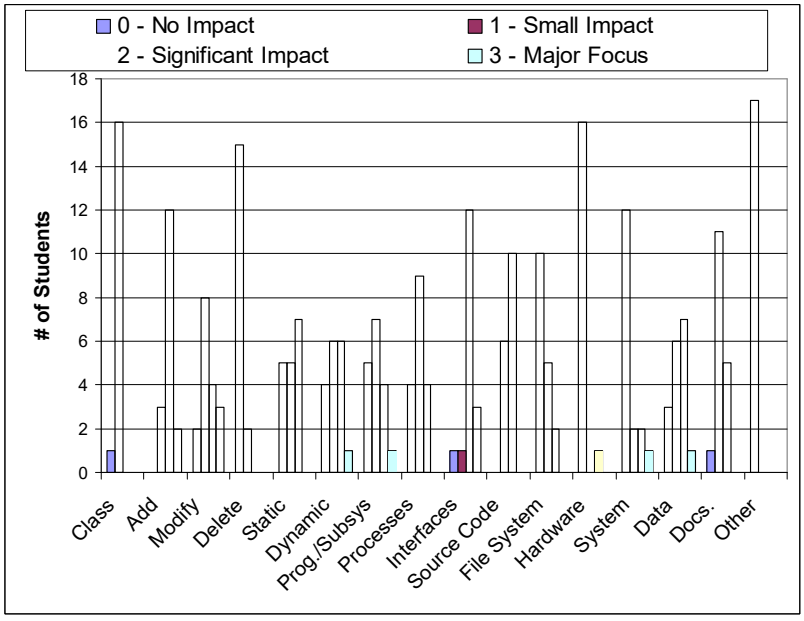

Figure 19 Sample Change #4 Consistency

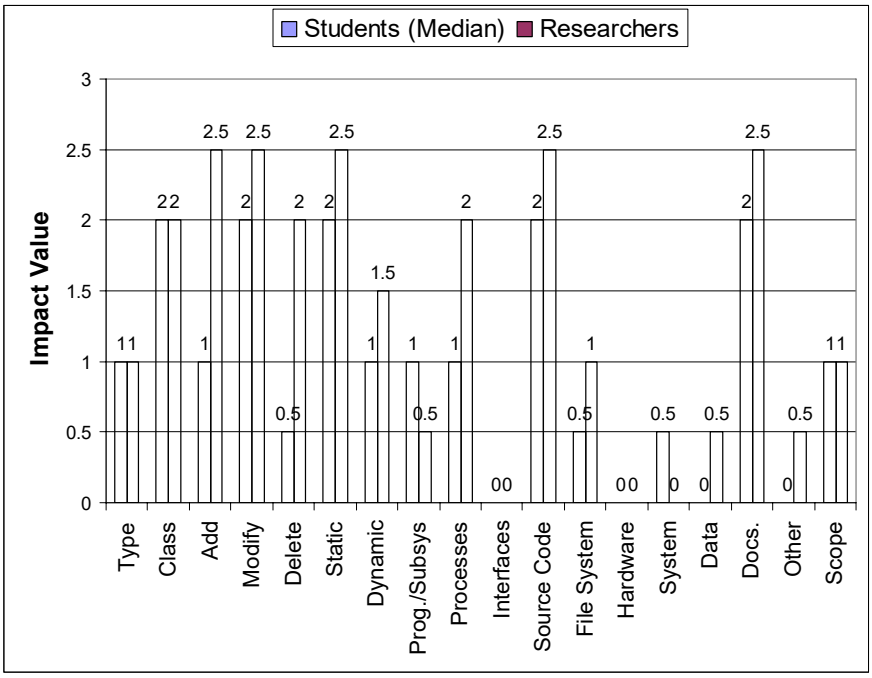

Figure 20 Sample Change #5 Accuracy

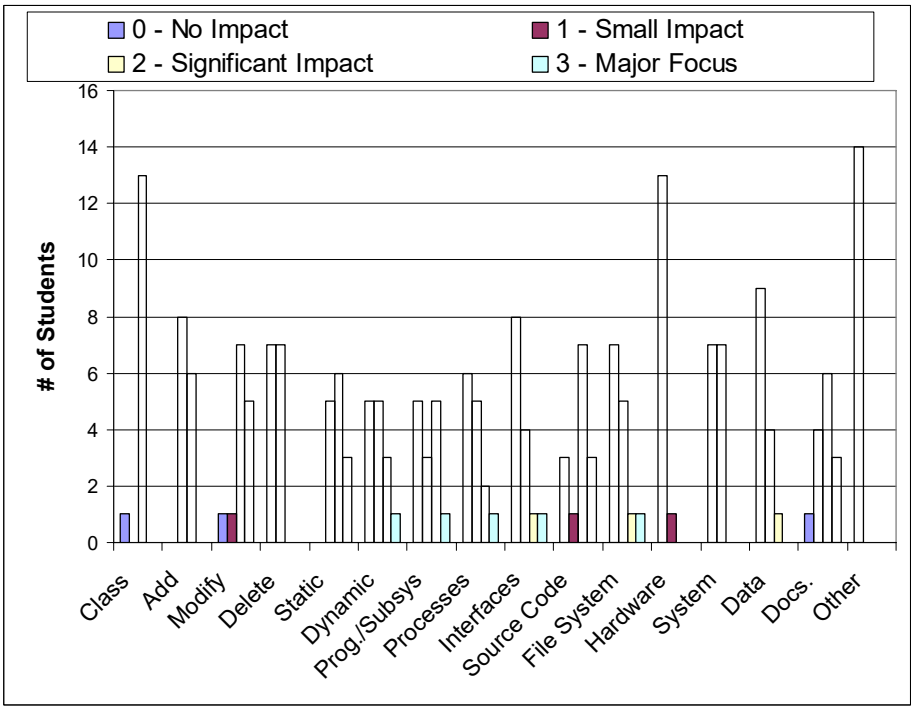

Figure 21 Sample Change #5 Consistency

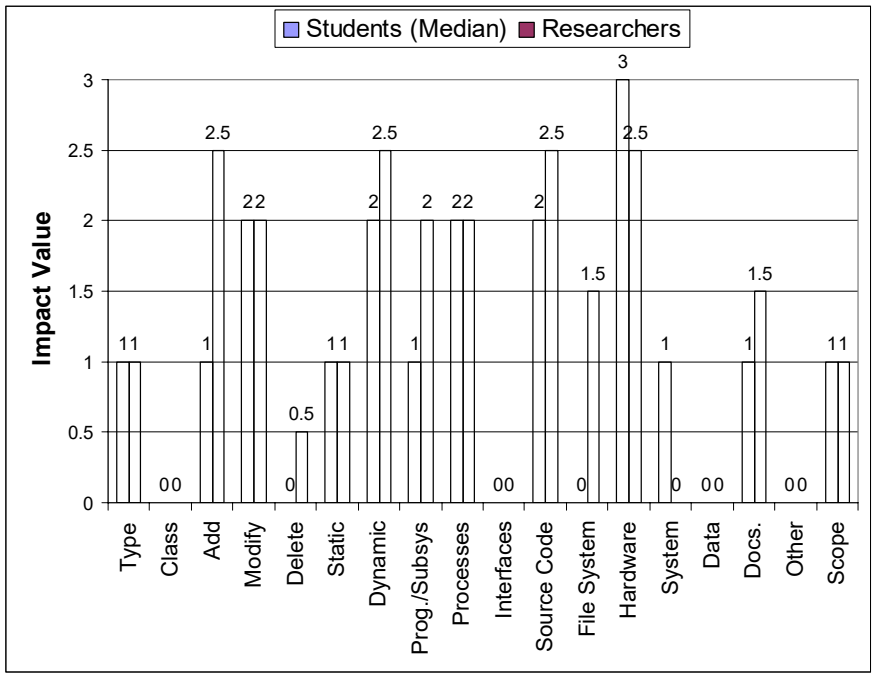

Figure 22 Sample Change #6 Accuracy

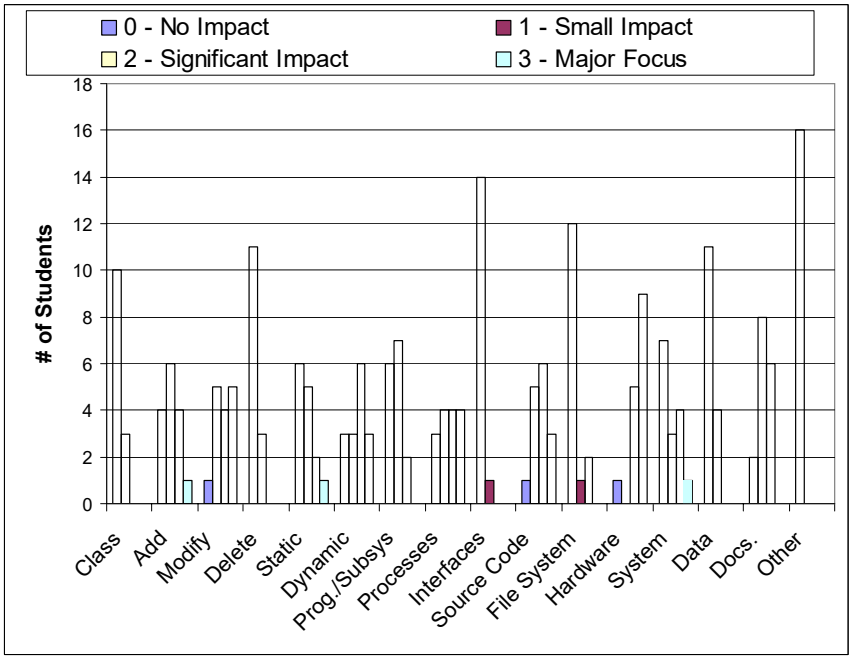

Figure 23 Sample Change #6 Consistency

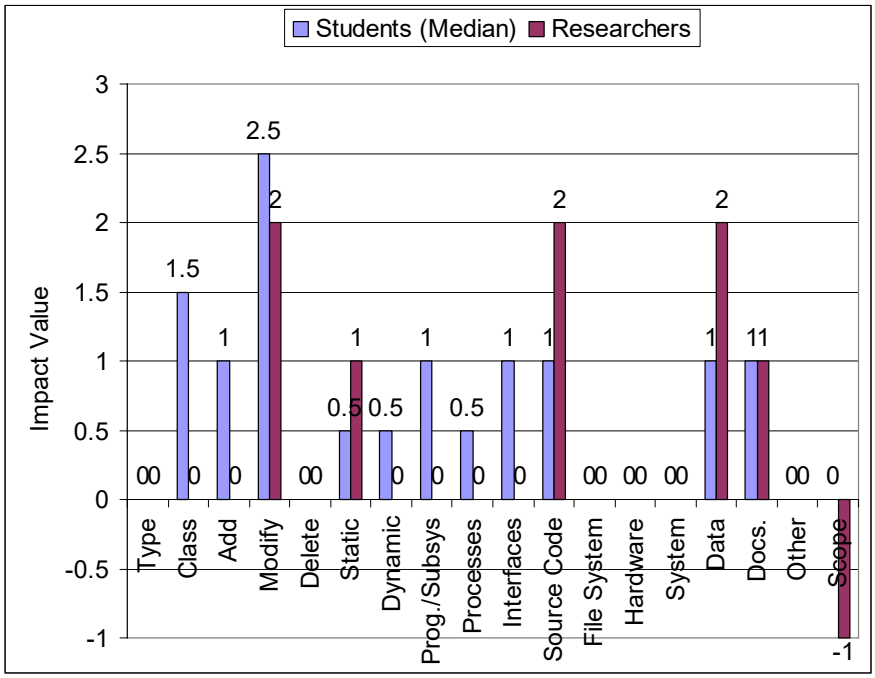

Figure 24 Sample Change #7 Accuracy

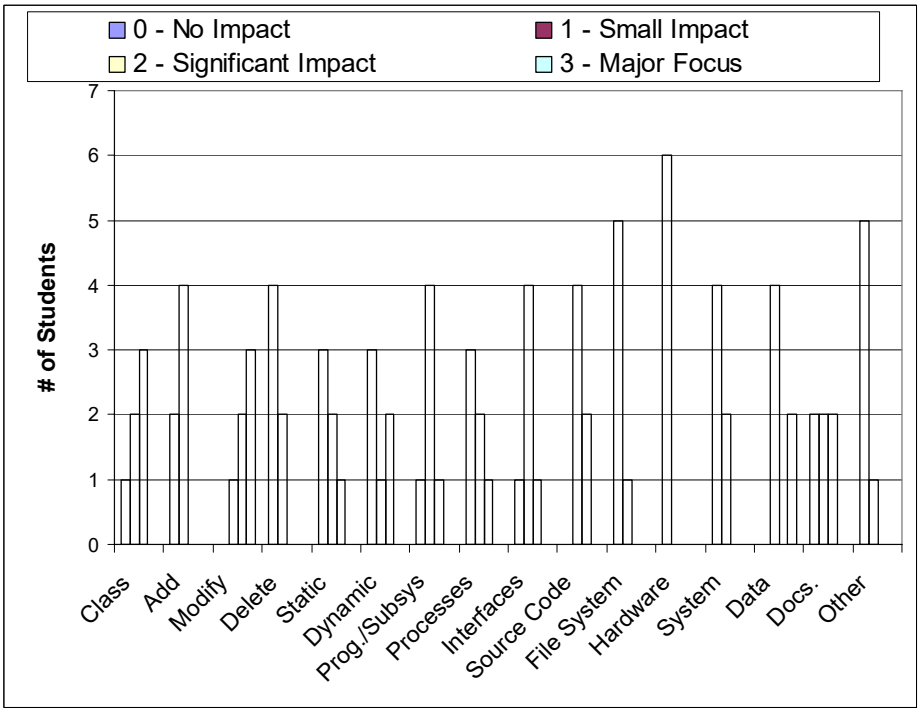

Figure 25 Sample Change #7 Consistency

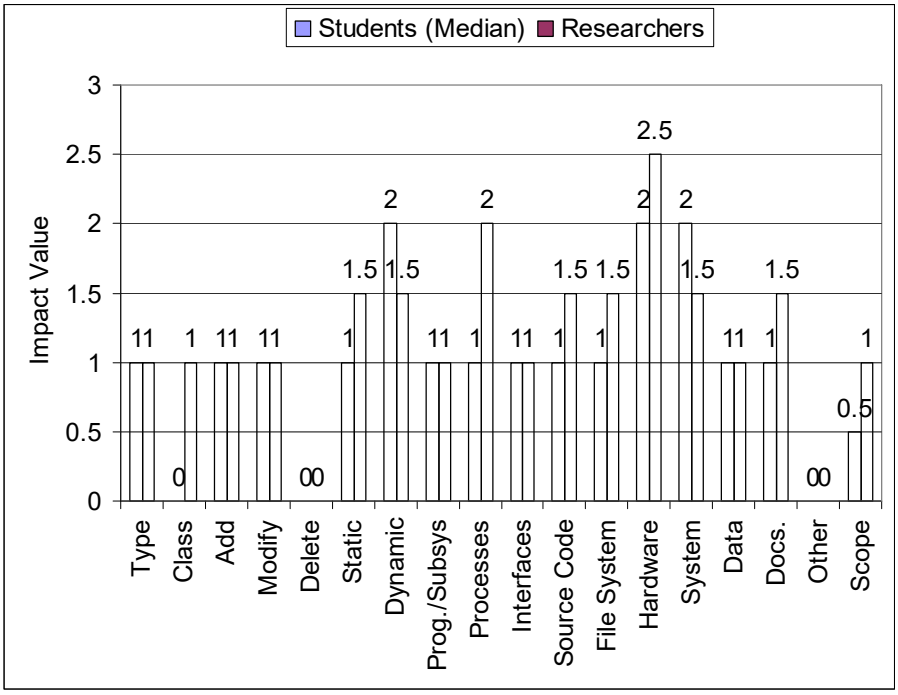

Figure 26 Sample Change #8 Accuracy

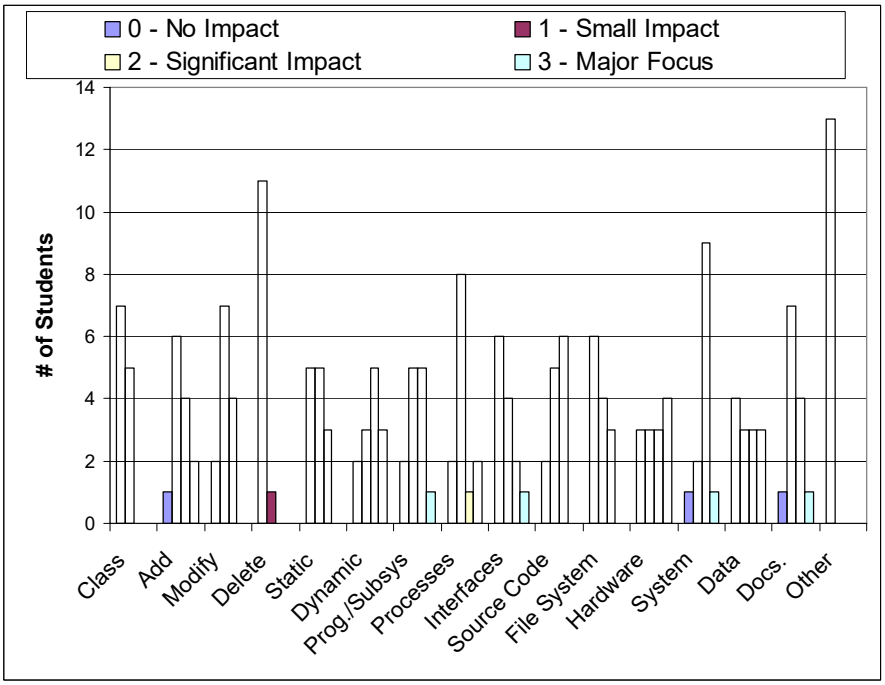

Figure 27 Sample Change #8 Consistency

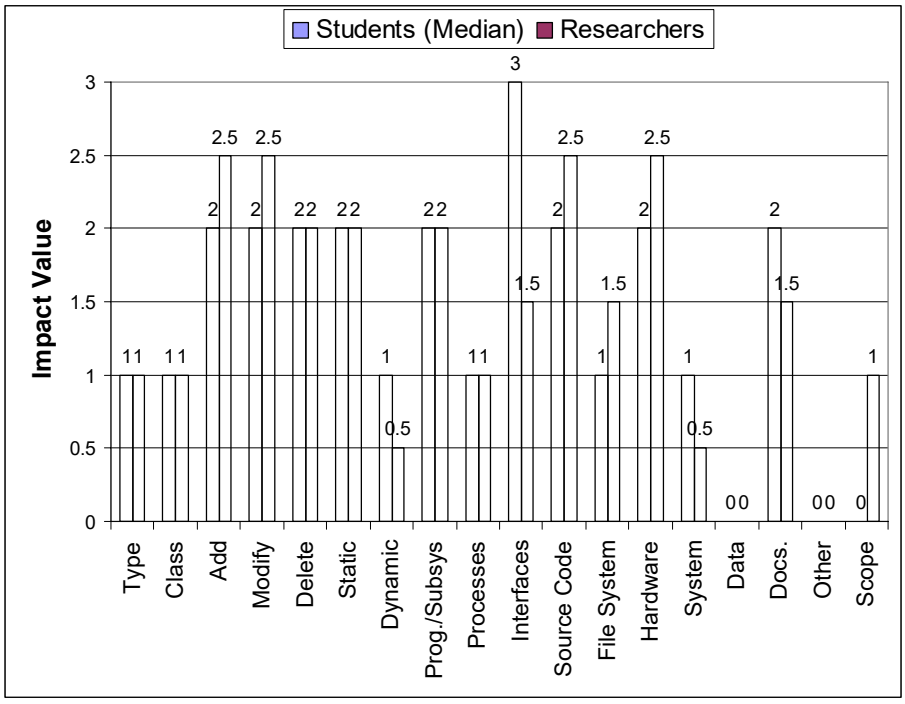

Figure 28 Sample Change #9 Accuracy

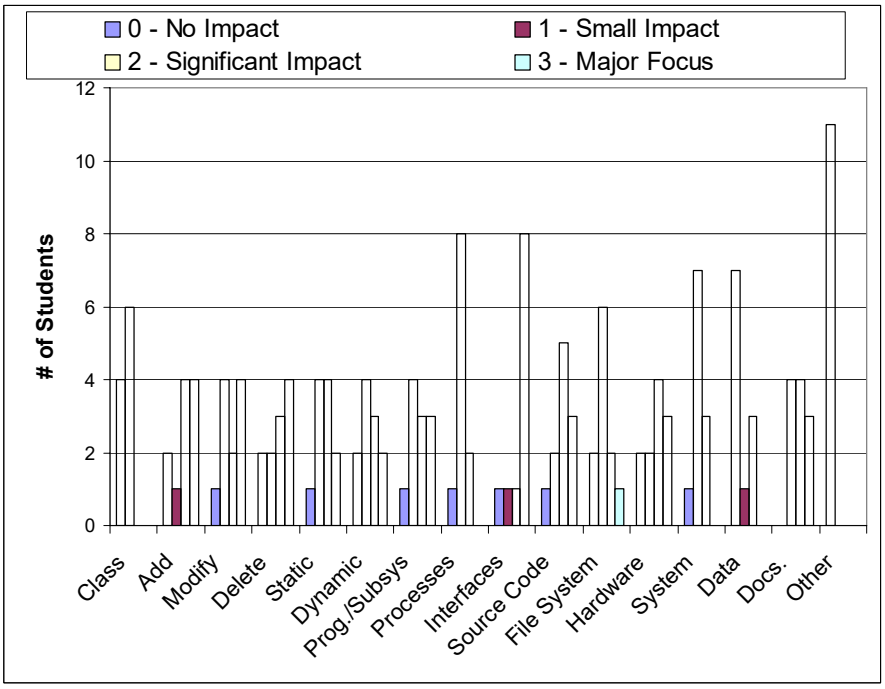

Figure 29 Sample Change #9 Consistency

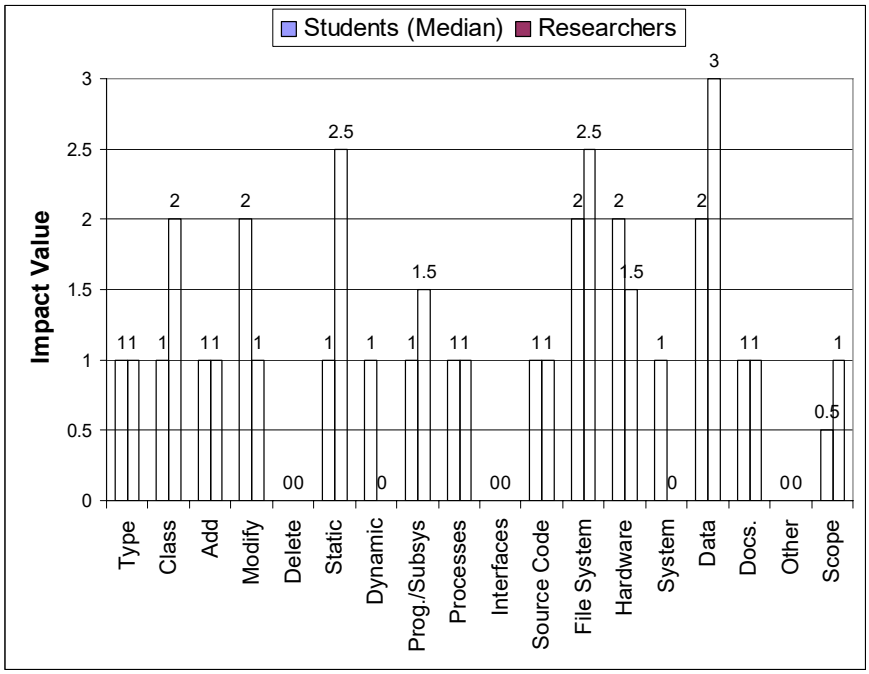

Figure 30 Sample Change #10 Accuracy

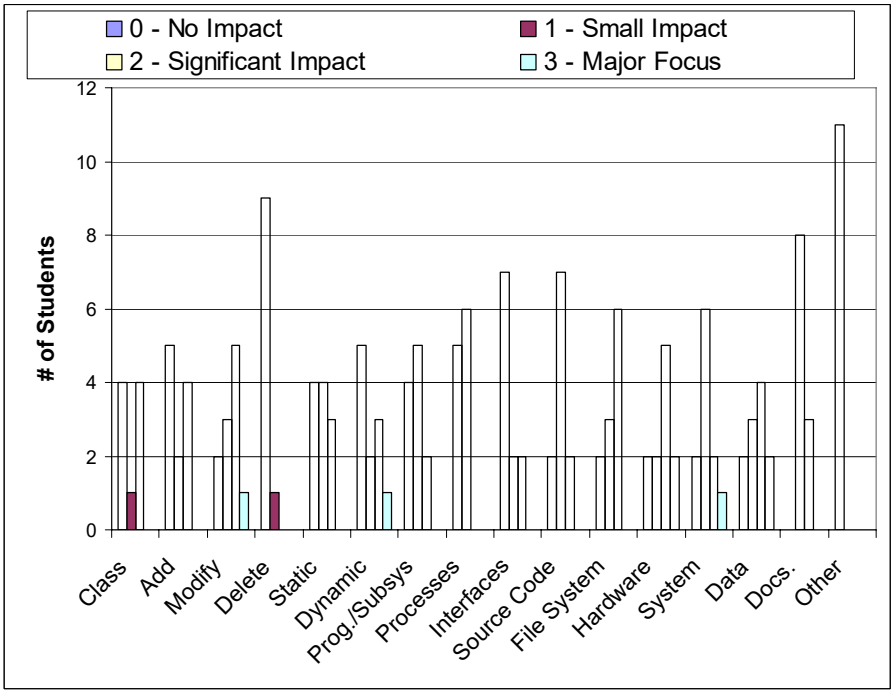

Figure 31 Sample Change #10 Consistency

London Ambulance Service CAD Sample Change Requests

Please classify the following change requests using the Architecture Change Classification Scheme. NOTE: All changes are assumed to be made to the original system.

1. **System Override**
   - Change CAD system to allow the Supervisor to override the assignment of the ambulances by the CAD System after an assignment is made.

2. **Fire Department Use**
   - Want to extend the system to work at a fire department to manage fire trucks. Human resources along with equipment must be accounted for. The system must also know the qualifications that each human has to ride on the fire truck and perform a certain function in order to dispatch the appropriate individuals.

3. **Medical Records**
   - System needs to add an interface to a medical records database that the Lot Supervisor, Ambulance Driver, and Dispatcher must be able to interface upon the receipt of a call and/or assignment of an incident.

4. **Google Maps**
   - Change the Dispatcher display to include a map and satellite image of the managed area. System will include display data from Google Maps online and will interface its API in order to integrate the maps display and functionality on the client machine.

5. **Increased Flexibility**
   - System maintainers notice a potential problem in system flexibility. They request a change to the system design that will integrate design patterns into the design of the software.

6. **Response Time**
   - Customer wants an increase in the response time of the system. The system engineer requires that the system be able to run in a multi-processor environment and the system must be updated to make use of the additional processors.

7. **Address Processing**
   - Tests found that the system is unable to decode the address information from a specific area code of callers provided by a telecom carrier. The required data format used to interpret this information and make it usable by the CAD system was not properly understood by developers.

8. **Increased Reach**
   - State government officials have required that all hospitals in this major metro and surrounding area use this system and interact with its central database. This change requires that additional servers be placed at 5 other hospitals that duplicate/mirror the data and processing capability of the original central server, and that all servers and clients communicate.

9. **Web Client Access**
   - The customer wants to change how the server is accessed by remote clients in order to better supervise and manage assignment processes from additional locations. They want to remove all PCs from remote locations and replace them with web terminals. This will require that all client access be done via the web. The GUI

application software that previously ran on the each PC to access the central data will be removed. The interface must now be displayed in a Java based web browser.

10. Update Databases

- The Hospital Resource and Ambulance Resource Databases are getting too large to remain on the central server. They need to be moved to separate severs that the central server can access.

APPENDIX B

CLASSROOM STUDY DOCUMENTS

Appendix B contains several forms used to collect data from the students in the classroom study. The first two forms are the change requests. Change Request #1 is the Conformance Monitor Change Request, and Change Request #2 is the Feed Display Change Request. The details of what the students were required to do for each change request are provided in the forms.

The Architecture Change Classification Form is the form the students used to classify each change requests. This form was used to classify the sample change requests shown in Appendix A and the change requests used in the classroom study. After reading the change requests, the students classified each change by selecting attributes and attribute impact values displayed on the form. They also used the form to provide rationale for the selections made.

The Architecture Change Definitions Form was provided to the students when completing the sample change requests and also included in the study material that we gave them. This form contains brief definitions for all of the attributes in the classification scheme.

The final form is the Post-Study Questionnaire that was used to obtain feedback from the students on how the scheme could be improved. The questionnaire also includes questions inquiring about the difficulty the students had in implementing the two change requests.

Change Request #1 – Conformance Monitor

The TSAFE system needs to be extended to indicate conformance problems that may arise with the monitored aircraft. It must be able to determine if a flight is conforming to the planned route or blundering. The function of determining the degree to which a flight is conforming, or not conforming (blundering), to its assigned flight plan is know as “conformance monitoring.”

Conformance monitoring is the detection mechanism that tracks a flight to determine whether it is conforming to its recorded flight plan or whether it is blundering. The Conformance Monitor performs a comparison of a flight’s position and heading against its flight plan. It does not make use of any trajectories, and therefore has no dependencies on the Trajectory Synthesizer. The Conformance Monitor returns the state of the flight, which is a Boolean: either the flight is conforming or it is blundering.

The first step to detect a flight’s conformance to its flight plan is known as residual generation. The residual is a measure of the dissimilarity between the observed state of the real world system and the expected state. Residual generation is the process of calculating the residual. The expected state of a flight is essentially the entirety of instructions given to the flights, including the filed flight plans (flight routes, assigned altitudes, and assigned speeds), flight plan amendments, and any clearances and directives given to the flights by air traffic control, as well as any reasonable extrapolations thereof. A non-conformance is deemed to have occurred, therefore, when a flight’s actual state deviates significantly from the expected state.

The Conformance Monitor uses the values input from the Conformance Data screen to determine whether or not a selected flight is conforming or blundering. The algorithms used to calculate this information will be provided. It is up to you to determine what changes need to be made to the architecture in order to implement this change. Finally, you must modify the code to reflect the change made to the architecture and ensure that the system functions as required. The basic functionality that must be added is as follows:

For each flight:
1) If it has no flight plan, assign it a dead reckoning trajectory and continue
2) If it has a flight plan, determine if it is blundering
3) If it is, assign its dead reckoning trajectory as its predicted trajectory
4) If it isn't, assign its route trajectory as its predicted trajectory

You must decide where the ConformanceMonitor class should be added to the system, and modify the appropriate classes to use the CM algorithm. The ConformanceMonitor.java file has been provided, but certain declarations have removed. You must figure out where the CM class belongs and include the declarations and parameters that are currently missing (marked by comments and a ‘?’).

In order to check to see if a flight is conforming or blundering, view the flights on the air traffic screen. Flights that are blundering will be shown using a red dot and text, and conforming flights will be displayed using a white dot and text.

In order to test this feature, for the demo-file.txt, Flight 1 conforms for its entire route, Flight 2 conforms at some points and blunders at others, and Flight 3 generally blunders for its entire route.

Change Request #2 – Feed Data Display

The air traffic control operator needs to be able to access the data that is provided by the Feed source server from within the TSAFE system. The operator needs to see the feeds to verify that a constant flow of information is being received from the feed source and that the correct type of information is being displayed. This feature will provide another check on the accuracy of the data by the ATC in the event the data is not read correctly by the system. The TSAFE system must be extended to display this information. The TSAFE System relies on the information from the feed source to show air traffic to its users. The system must now make an additional connection to the Feed source in order to display the information output by the Feed Source Server in a separate window from within the TSAFE system.

The TSAFE system should be extended by adding this data to the TSAFE map display aircraft flight data. This window will display the text feeds from the FigFileGenerator Server. It can either be displayed from within the TSAFE map window, or displayed in a window of its own. The ATC must be able to view this data when launching TSAFE.

The Feed Source that TSAFE reads the flight data from outputs this information to the system. This information should be displayed in the additional window that is added to the TSAFE GUI.

The FMS Feed Generator v1.0 source file that executes the server has been included with this change request.

Architecture Change Classification Form

Change Request Name & Number:

High-level change detail (Please circle one value for each listing)

- **Phase** — Requirements Design Code Test Maintenance

- **Type** — Defect Enhancement

Please define the level of impact that each change class will have using following 3-point scale when needed:
0 = No impact; **1** = small impact; **2** = significant impact; **3** = major focus of change;

Enhancement Detail (Please circle one value for each listing)

Class — Adaptive Perfective Preventative

Type

• Add	0	1	2	3
• Modify	0	1	2	3
• Delete	0	1	2	3

Properties

• Static	0	1	2	3
• Dynamic	0	1	2	3

Impacts

• Program/Sub-system	0	1	2	3
• Processes	0	1	2	3
• Interface	0	1	2	3
• Source Code	0	1	2	3
• File System	0	1	2	3
• Hardware	0	1	2	3
• System	0	1	2	3
• Data	0	1	2	3
• Documentation	0	1	2	3
• Other: _____	0	1	2	3

Please define the effect on the system scope using the scale below:
-1 = purely functional change; **0** = affect both functions & architecture; **1** = purely architectural change;

Scope

• Functional vs. Architectural	-1	0	1

Defect Detail (Please circle one value for each listing)

Class — Corrective

Found — Inspection Testing User Reported

Origin — Requirements Design Code

Issues

• Architecture/Design	0	1	2	3
• Environment	0	1	2	3
• Problem Definition	0	1	2	3
• Domain Knowledge	0	1	2	3
• Technology	0	1	2	3

- Interface (System/User) 0 1 2 3
- Data Transmission 0 1 2 3
- Data Accessibility 0 1 2 3
- Other: _____ 0 1 2 3

(Please include any relevant comments below, stating why you chose the selected classes and their respective levels of impact for any classes that will significantly impact the system)
Rationale:

Architecture Change Classification Definitions

Enhancement change due to a requested improvement of the system

Defect change is due to an error, fault, or failure detected in the system

Phase location in development lifecycle where the change request is encountered – e.g. requirements, architecture, design, code, test, maintenance

Architectural ←→ Functional
Scale used to determine type of change – **architectural change**: affecting the structure of a system rather than user-observable attributes **functional change**: change to a user function

Adaptive change due to new hardware and software interfaces – change in the standards or protocols used in system communication (adapting to a new environment)

Perfective adding new features or changing the system to better suit its users (changes not included in adaptive class)

Preventative restructuring to prevent future problems – reengineering

Add, Delete, Modify Adding, deleting, or modifying code, modules, and/or components (affect of change on source code)

Impacts

- Hardware – level of impact to system hardware (i.e. modification to improve data storage or increase system memory)
- System – level of impact change will have on related systems that rely on the software
- Data – affect on data access (storage, retrieval, transmission) - any data used by system or data that is provided by system to other systems
- Documentation – level of impact to documents affected by the change – including requirements, design, user guides, manuals….etc
- Interface – level of impact to system interfaces and user interfaces - user to system component or component to component
- Source code– level of impact change will have on the source code
- Program/Sub-system – impact to subsystems of larger system
- Processes – impact to software processes and C&C run-time structure
- File system – impact to the implementation structure – how the class, library, and resource files will be affected by change
- Other – impact to additional aspect of system not in classification list

Properties

- static – change that will affect static design properties (i.e. class diagram, component diagram, deployment diagram)
- dynamic – change that will affect dynamic design properties (i.e. activity diagram, statechart diagram, collaboration diagram)

Corrective change due to defect (error, fault, failure)

Inspection change requested due to a defect found during an inspection

Testing change requested due to a defect found during testing

User Reported change requested due to a defect found by a user of the system

Location

- location of the source of the defect (requirements, design, code)

Areas

- Architecture/Design –design solution does not adequately solve the problem
- Data accessibility – unable to access required data stores – format of data not correct (incoming & outgoing)
- Environment – problems related to system's operating environment(i.e. operating system, hardware, relationship to other systems)
- Problem Definition – problem isn't completely or clearly defined
- Domain Knowledge – team doesn't have adequate knowledge or experience to solve problem
- Technology – the languages, tools, and components being used to build/maintain the system are inadequate
- Interface System/User – problems in with an external interface
- Data Transmission – problem transmitting data to a requesting system

Post-Study Questionnaire

Architecture Change Classification Survey

These questions all refer to the change classification that was covered by Byron Williams in class. Please answer these questions based on your experience in the classroom exercise and on homework 4.

1. List any additional attributes or sub-attributes that you recommend adding to the scheme for either *defects* or *enhancements*. Explain.

   _____
   _____
   _____
   _____

2. How would you improve the classification scheme?

   _____
   _____
   _____
   _____

3. Are there any inconsistencies in the attributes of the scheme and your experiences in using the scheme during development? Please describe.

   _____
   _____
   _____
   _____

4. Were there any characteristics of the scheme that were confusing? Please explain.

   _____
   _____
   _____
   _____

5. Please list and describe any attributes that were unnecessary in the scheme for either *defects* or *enhancements*. Please explain.

   _____
   _____
   _____
   _____

6. Please rate your level of agreement in this section with respect to the following 5-point scale:
   (Please include any relevant comments below)
   1 = Totally disagree; **2** = somewhat disagree; **3** = neither agree or disagree;
   4 = Somewhat agree; **5** = Totally agree

Architecture Classification Scheme Practicality

- The attributes are logical and easily understood 1 2 3 4 5
- The scheme is beneficial for a developer making a change 1 2 3 4 5
- The scheme has practical application in industry 1 2 3 4 5
- The scheme is easy to use 1 2 3 4 5
- After classifying both changes, I had an idea of which would would be the most difficult to implement 1 2 3 4 5

Additional Comments:

_____
_____
_____

These questions now refer to the changes for homework 4.

7. Which change request (please give the name or short description) was more difficult to implement? What aspects of that change made it more difficult?

   _____
   _____
   _____
   _____

8. For each change, were the changes you made to the architecture easy to make to the code?
   Conformance Monitor:

   _____
   _____
   ATC Feed Display:

   _____
   _____

9. Which architecture change was more difficult to make to the code and why?

   _____
   _____
   _____
   _____