Analytical Design of Evolvable Software for High-Assurance Computing

Carol L. Hoover

14 February 2001
CMU-CS-01-111

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering.

Thesis Committee:
Daniel P. Siewiorek, Chair
Donald E. Thomas, Jr.
Philip J. Koopman
Rick Kazman, Software Engineering Institute
Susan Finger

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Material Command, USAF, under agreement number F30602-96-2-0240 as well as by an Intel fellowship. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Laboratory, the U.S. Government, or the Intel Corporation.

Keywords: Evolvable Software, High-Assurance Computing, Software Architecture, Software Design.

Abstract

Software is a ubiquitous feature of today's world. The goodness of products and services is frequently dependent on the goodness of the related software. Optimal software performs correctly and requires minimal effort and cost to develop and maintain. The development of optimal software is an admirable goal but is difficult to achieve. In particular, software maintenance and evolution is costly and error-prone. The significance of the problem is magnified for high-assurance applications that require the certainty that the software will behave reliably despite budget constraints and product evolution. Though automated software development is the ideal solution, design for evolution is the practical solution. For most applications, analysis of the required behavior (behavioral analysis) and translation into a blueprint for building the software (software design) are necessary. High-level design involves the organization of the required behavior into building blocks or components. Design for evolution is the generation of a software architecture that can be changed with minimal human effort to produce a class of similar applications. Design for evolution makes feasible the cost-effective development of high-assurance applications.

This dissertation presents a semi-automatable research approach for designing an evolvable software architecture. The research approach focuses on the partition of basic elements of a software solution into reusable components that localize the effects of change. The input to the partitioning process is a set of software requirements along with an analysis of the required behavior and planned or feasible evolution of the product line. The output is a partition of the required behavior into components that reduce the effort associated with developing a software product line. The dissertation provides an analytical verification of the research approach through proof and constructive examples. Empirical results validate the effectiveness of the research approach in comparison to human intuition, experience, or other training. The research approach is novel and fills a gap in the systematic generation of software architectures that minimize the effort associated with product-line evolution. The dissertation describes in detail the degree to which the research approach is automatable and specifies, more generally, future research needed to achieve full automation of software architecture generation.

Acknowledgments

First, I would like to thank my advisor, Dan Siewiorek, for his guidance and support. His wisdom and thoughtful suggestions inspired me to think more critically and to continually try to improve my research. Most importantly, his kindness to students helps to make the doctoral process less stressful. Thank you also to Laura Forsyth and Marian D’Amico who helped with scheduling and communications.

My thesis committee members offered sound advice. Their expertise and research helped me to better understand how my work fits into a broader scope of engineering design. For their contributions and service as committee members, I thank Don Thomas, Phil Koopman, Rick Kazman, and Susan Finger.

On a personal note, I sincerely thank my husband Marco for his encouragement and input. He provided useful feedback about the application of my research to software engineering practice and patiently endured the geographical distance between us during most of my thesis years.

Lastly, I am grateful to my parents for their part in my education. They have always done what they could to encourage and support my desire to learn.

Table of Contents

Appendices:

List of Figures

Figures in Appendices:

List of Tables

Tables in Appendices:

Dedicated to my family.

1 Introduction

A fundamental problem of software development is uncertainty. There is no guarantee that the resulting software will be optimal or the development costs within budget. Optimally, the software performs correctly and satisfies design objectives, such as maintainability and reusability, as well as economic constraints. Uncertainty introduced by change is not acceptable for high-assurance computing which requires a high degree of confidence that the software will perform correctly. The current approach to reducing uncertainty involves software testing and repair, but extensive error detection and correction is time-consuming and expensive.

Designing software for change can reduce the error, time, cost, and ultimately the uncertainty associated with maintenance. Popular design approaches such as object-oriented and structured design help to promote software maintainability and reuse but require the human to determine how to precisely apply them to a particular problem and solution. Existing design tools assist the designer in documenting a particular design, in checking interfaces between parts of the solution, and in some cases verifying a formal model of the system behavior. For some well-understood and functionally stable domains, they may automatically generate code from requirements. Current design approaches and tools do not for any arbitrary set of requirements systematically or automatically determine a functionally appropriate solution or organize this solution into a software design that minimizes the complexity of change.

The thesis addresses the general problem of how to *consistently* design software solutions that are evolvable as well as reusable. In the context of this dissertation, evolvable means that the software can be statically (off-line) changed with minimal effort and error to satisfy changing requirements. Likewise, reusability is the property of being useful in more than one different software solution or application. Software components that are reusable have the potential to make it easier to develop systems related by common functionality. Reusability can enhance evolvability and is therefore a complementary design objective. Evolvability is critical for high-assurance computing which requires a high degree of confidence that static changes to software preserve functional correctness.

This chapter provides an overview of the thesis research and contains the following sections.

- Section 1.1 discusses the motivation for the research.
- Section 1.2 is a discussion of the problem area.
- Section 1.3 contains the detailed problem statement.

- Section 1.4 is an overview of the research solution.
- Section 1.5 presents the general hypothesis about the effectiveness of the research solution.
- Section 1.6 overviews the organization of the dissertation.

1.1 Motivation for the Research

Software is a ubiquitous feature of today's world. Products and services are as dependent upon software as upon metals, plastics, electricity, light, and human effort. The difference between "good" and "bad" software can make the difference between "good" and "bad" products and services. Incomplete, defective, or suboptimal software can result in unreliable, useless, or even dangerous systems [105]. Similarly, the difference between "good" and "bad" software development practices can determine whether or not the resulting product or service meets its economic goals [31]. There is a need to increase the probability that software projects follow a process that leads to the development of good products or services within economic constraints.

Researchers and practitioners in the areas of computer engineering and science, software engineering, management of information sciences, and other related disciplines, can identify the characteristics of good software products and processes. For instance, there are algorithms for optimizing CPU performance and memory usage as well as theories concerning the organization of data [42,91], and practitioners as well as researchers have documented lessons learned from project management [31,71]. However, software developers still find it difficult to consistently apply the best ideas in these fields. Unfortunately, people continue to develop or use suboptimal algorithms, to develop haphazard information models, and to mismanage projects. The state of software engineering practice is particularly critical for the development of software deployed in applications such as air traffic control, avionics, and nuclear power plant control [32,105]. These applications require software that is highly reliable and safe as well as useful.

There are currently four basic approaches to address the lack of consistency in the application of proven or at least promising ideas for software development.

1. *Education and Training* - Software engineering education has a 30-year history in which academe has struggled to fulfill industry needs via single courses, Master's curricula, and now undergraduate degree programs [150]. Some programs have incorporated specialization in particular technology or application domains [64].
2. *Process Development and Institutionalization* - Organizations formalize their development and management processes to make the organization work with a common focus and to apply best practices consistently [70,118].
3. *Reuse of Software Artifacts* - Instead of repeating the decision-making process for the same ques-

tions, organizations use the results of a previous decision process. The term *software artifact* refers to any tangible result of the software development process such as requirements and design specifications, feasibility studies, project plans, test procedures, source code, etc.

4. *Automation* - When feasible, tool designers embed the best ideas and practices into tools that enable their consistent application. For instance, tools to support software architectural analysis are progressing [79].

Increased automation of software development offers the best hope for the consistent development of good software. To make a positive impact on software development, automation must go beyond the recording or management of design artifacts that many software engineering tools offer today. In addition to programming language translation, automation should address higher level design: current automation is most prevalent at the level of program compilers. Ideally, these new tools would enable us to precisely specify the requirements for an application and to automatically generate the software solution. Unfortunately, software designers currently do not have techniques that enable them to precisely express all types of requirements; nor do they have techniques to automatically (without human intervention) convert all types of requirements into software solutions.

Reuse of good software artifacts can implicitly lead to the reuse of best practices and products. A good software artifact is one that has been shown, validated, or certified to perform a useful function correctly. The concept of component-based software development originated with idea of building applications from reusable pieces of software called components. The idea is to design standard components whose functionality is useful across multiple solutions. A refinement of this idea is to develop standard solutions defined as software architectures which are useful across applications. The specification of a software architecture includes a description of the following structural features of the solution [132].

- Type and number of each component
- Name and interface for each component
- Functionality of each component
- Interactions between the components

A standard software architecture is the high level design for a class of software applications with common functions and contexts for use: it is a solution to be used across a family of software products composed of reusable components. The next section further describes the problem area for the thesis research.

1.2 Problem Area

Figure 1.1 details a process for building a software application from new and/or reusable software components. The input and output of each process box (a step in the overall development process) is a software artifact. For instance, the application requirements specification and detailed behavioral models are input artifacts to the software architecture specification process. The output artifact is naturally the specification of a software architecture for the application to be built. Appendix A shows a more general version of this process for generic component-based product development.

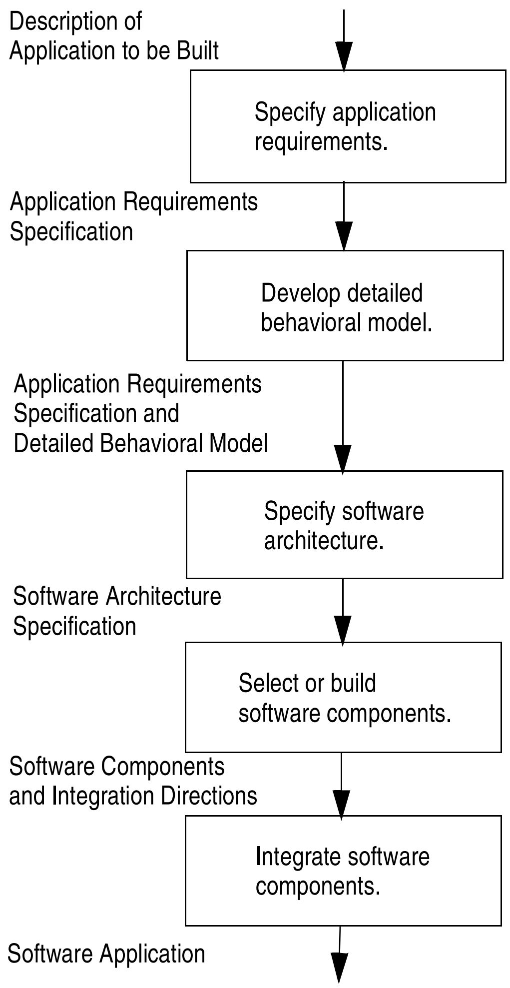


Figure 1.1 Component-based software application development.

Each box in Figure 1.1 is decomposable to support reuse of the type of artifacts generated by the specific box. Figure 1.2 shows the decomposition of a generic software artifact generation box. The decomposition includes process components to determine if an appropriate software artifact already exists in an artifact database and to generate an appropriate artifact if one does not exist. The names of the input and output to the software artifact generation box are in straight text, while the names of artifacts passed between the interior process components are in italics.

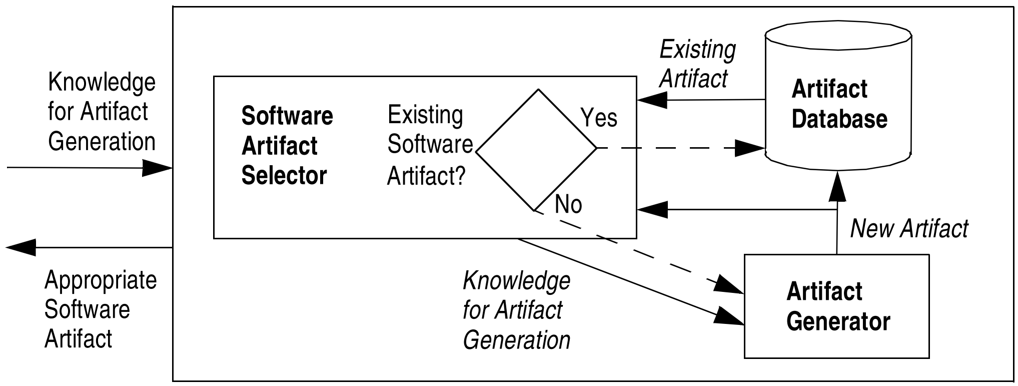


Figure 1.2 Decomposition of a software artifact generation box.

Figure 1.3 shows the decomposition of the software architecture specification process box to support reuse of existing software architecture specifications. The idea is simple: determine if an appropriate specification exists and generate a new one if needed. The implementation of this idea is complex: for any input set of software requirements and behavioral model, determine if an appropriate software solution is known and if the specification for the corresponding software architecture exists in the database. If one is not found, then generate an appropriate software architecture and save its specification in the database so that it can be easily reused in the future.

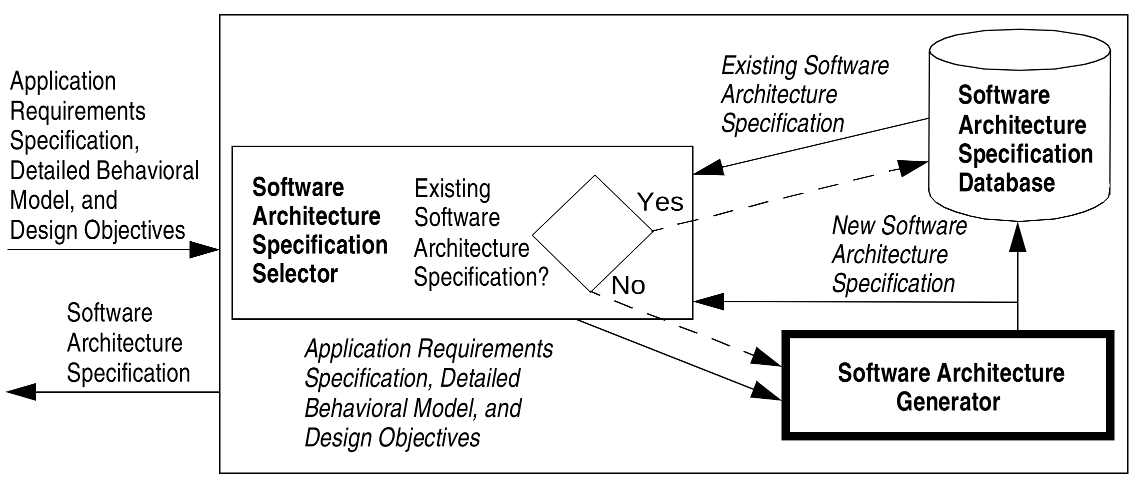


Figure 1.3 Decomposition of software architecture specification.

The interior process components involve the following related but separate areas of research which will be discussed in Chapter 3.

- Specification of software architectures and evaluation of the correctness of software architectural specifications.
- Generation of software architectures that satisfy input requirements and design objectives.
- Evaluation of candidate software architectures.
- Mapping requirements to existing software architecture specifications and finding appropriate specifications in a database.

The area of research for the thesis is the generation of a software architecture as indicated by the bold-lined box that encloses the Software Architecture Generator process component. The research does not address the mechanism for specifying a software architecture.

As shown in Figure 1.4, the generation of a software architecture includes two major processes:

1. Determination of the solution elements that will satisfy the software requirements and behavioral model.
2. Partition of the solution elements into components of the architecture that meet the design objectives.

This dissertation focuses on these two processes for the generation of new architectures. The next section further refines the problem and presents the problem statement for the research.

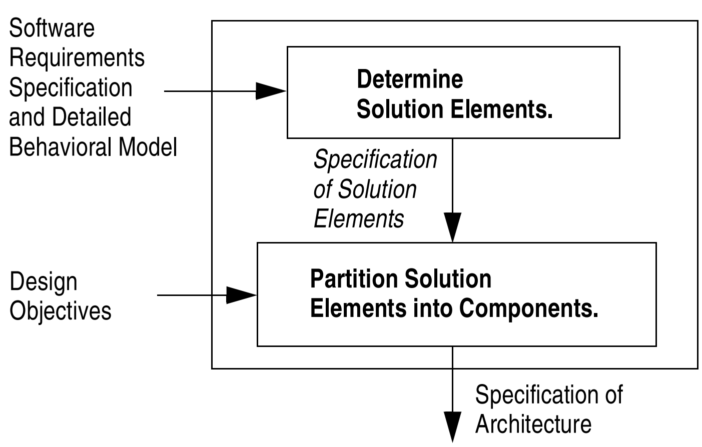


Figure 1.4 Software architecture generator.

1.3 Problem Statement

The problem area, as discussed in the previous section, addressed by the thesis is the generation of software architectures that satisfy not only requirements but also design constraints. The global problem (Figure 1.5) is the determination of software components that simplify and make more reliable the evolution of a product family of high-assurance applications. The thesis problem (Figure 1.6) is how to partition basic solution elements into reusable software components that minimize the effort and error associated with changing the solution. Chapter 4 discusses the target measures for reusability and change complexity that will be used to evaluate the research solution. The next section of this chapter briefly discusses the research solution.

Global problem:

How to determine the software components needed to build a product line of high-assurance applications.

Subject to the constraints:

- Applications evolve, so software changes.
- Modifying software is largely a manual process that is time consuming, expensive, and error prone.
- High-assurance computing requires the software to be reliable.

Figure 1.5 Problem of software architecture generation.

Given:

- A set of solution elements such as the high level data and operations.
- Knowledge about how the:
  - Problem and solution may need to change.
  - Solution will be reused.

Partition:

Solution elements into components that satisfy design objectives.

Design objectives:

- Minimize change complexity measure.
- Maximize reuse measure.

Figure 1.6 Thesis problem.

1.4 Research Solution

Software design involves the repeated partition of a high level solution into components. The partition process starts with the identification of basic system and subsystem components (high level components) and concludes with the identification of modules (low-level components) to be implemented. For this research, the term partition has two related but different definitions: (1) the process of dividing into parts, and (2) the process of grouping elements into disjoint sets. We specifically apply the first definition to the decomposition of a software solution and the second definition to the logical grouping of solution elements such as data and operations.

Some researchers draw a distinction between high level and low level components. In their view, high level components compose the system architecture and low-level components are design elements [132]. For the purposes of this dissertation, design encompasses the entire process of translating a problem (software requirements) into a specification for a solution (design specification); and the term architectural specification is synonymous with design specification. The equivalence is appropriate because for this work the focus is on the partition of components at various levels (high or low) of abstraction and because the thesis does not address aspects of a design specification (e.g., directions for implementing components) that may not be included in an architectural specification.

The research solution consists of the following processes.

1. Identification of basic data and operational solution elements as determined from operational behaviors described in behavior analysis.
2. Decomposition of solution elements based on reuse.
3. Analysis of the relationship between the design objectives such as evolvability and the solution elements.
4. Composition (grouping) of solution elements based on these relationships.

These processes embody a rationale for thinking about design objectives during the transformation from requirements to design and include techniques for analytically partitioning data, operations, and control flow into components that satisfy the design objectives of evolvability and reusability. Chapter 5 presents the details of the research approach to the analytical partition of software components. The next section briefly states the general hypotheses about the research solution that were tested. The specific hypotheses for the two human subject experiments that were conducted to evaluate the research solution appear in Chapter 7.

1.5 Statement of Hypothesis

In general, the research goal is to enable human designers to more consistently generate good software architectures. As discussed earlier, the overall approach is to make the design process more precise, systematic, and analytical. The research solution systematically guides the human to make decisions that can result in designs that achieve the design objectives of evolvability. It is possible that competent designers can make decisions that result in designs that also achieve the design objectives and that competence may correlate to experience and training. But as will be discussed later, current design methods do not necessary guide the designer to systematically make decisions that achieve the design objectives.

Therefore, the general hypothesis concerns the comparison of software designs prepared by designers who apply the research solution with software designs prepared by designers who do not apply the research solution. The phrase "do not use or apply" refers both to designers not trained to use the research approach as well as to designers trained to use the research approach but who do not apply it for some reason. Figure 1.7 contains the statement of the general hypothesis.

> Software designs developed by designers who apply the research solution will be as or more likely to demonstrate design objectives than those developed by designers who do not use or apply the research solution.

Figure 1.7 General hypothesis about the effectiveness of the research solution.

Likewise because the research solution is precise and systematic with corresponding mathematical models, one can hypothesize that this solution is at least semi-automatable.

The next section shows the layout of the dissertation and serves as a navigational guide to the reader.

1.6 Organization of the Dissertation

The organization of the dissertation, as shown in Figure 1.8, facilitates navigation for a variety of readers. The bold-lined boxes indicate those chapters that can logically be read separately or in succession. People who are primarily interested in background on software design and high-assurance computing, research in software architecture generation or evaluation, software design measures, or the author's analytical approach to partitioning software components, can turn directly to any of chapters two through five, respectively.

Those readers solely interested in the validation of a software design approach should at the minimum read Chapter 6 followed by Chapter 7, Chapter 8, and Chapter 9. Chapter 5 describes the design approach that is being validated by the empirical experiments discussed in Chapter 6 and Chapter 7 and is therefore helpful though not necessary for a full understanding of the experimental design. Chapter 9 presents an overview of the research needed to more fully automate the generation of evolvable and reusable software architectures as well as suggestions for advancing the reuse of software artifacts via automation of the software development process. The dissertation concludes with a list of references related to software design and measurement as well as supporting appendices.

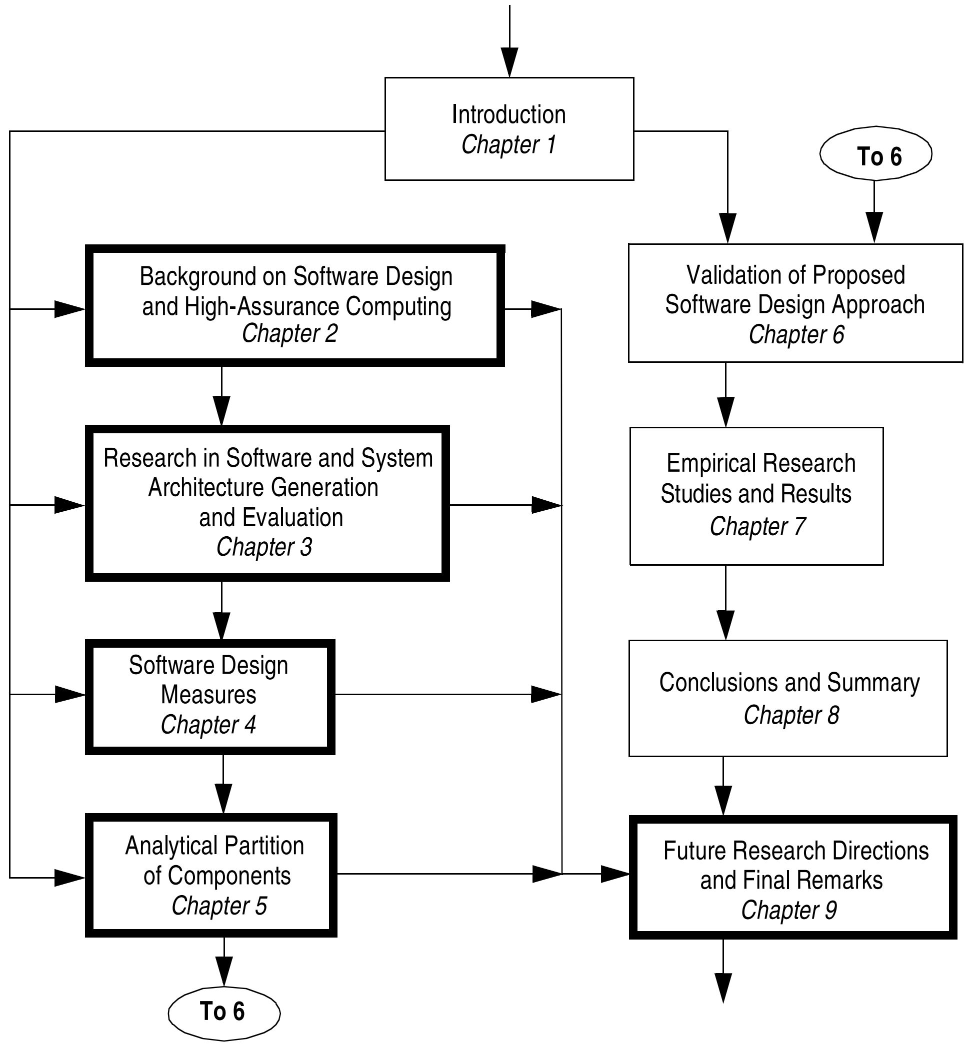


Figure 1.8 Organization of the dissertation.

2 Background on Software Design and High-Assurance Computing

For most applications, software design is a necessary part of developing the solution; and software design decisions are primarily a human-intensive activity. This chapter characterizes the nature of software design and illustrates the types of requirements that make high-assurance computing systems a challenge to design. The organization of this chapter consists of the following sections.

- Section 2.1 details the process of software design.
- Section 2.2 illuminates the role of partitioning in the software design process.
- Section 2.3 defines the concept of a “good” design.
- Section 2.4 explains why the consistent generation of good designs is difficult for humans.
- Section 2.5 illustrates the types of decisions made by a designer using an object-oriented design approach.
- Section 2.6 describes the characteristics of a high-assurance computing system.
- Section 2.7 explains why software changeability is important for high-assurance systems.
- Section 2.8 further discusses why the design of these types of systems is complex.

2.1 What is software design?

Design is the transformation of requirements analysis to a description of how system should be built. The content of this description depends upon the type of design approach. With a structured design approach such as those defined by Ward and Mellor [153] or Yourdon and Constantine [160], the description includes a definition of the modules and tasks in which the modules will execute as well as a mapping of the tasks to processors. With an object-oriented approach such as those defined by Booch [27] or Rumbaugh [126], the description includes solution-oriented class and object diagrams. Module and class definitions are the physical representations of the basic components for the structured or object-oriented types of design, respectively.

A design specification describes in detail how the software will perform the functions outlined in the requirements and behavioral specifications as well as how the software should be constructed. More specifically, it details the following features of the software solution.

- Architecture of the software system
- Execution platform and operating environment (if not already specified in the requirements specification)
- Integration and packaging of the software components or system
- Other implementation requirements such as the use of existing software components, language(s), and compiler(s)

The design is the official specification for verifying the correctness of a software implementation. The next section explains the role of partitioning in determining basic components of the software solution.

2.2 What is the role of partitioning in the design process?

Partitioning (decomposition and grouping) is an important part of the design process. Figure 2.1 and Figure 2.2 show a macro-level view of the transformation from requirements to design for structured and object-oriented types of design. With both types of design, the designer must determine the appropriate level of decomposition as well as the grouping of solution elements to best satisfy design objectives. The terms decomposition, allocation, and grouping in these figures indicate the extensive and iterative role of partitioning in the transformation from requirements to design. The object-oriented design process consists of several micro steps, each of which might be performed repeatedly in an iterative development process.

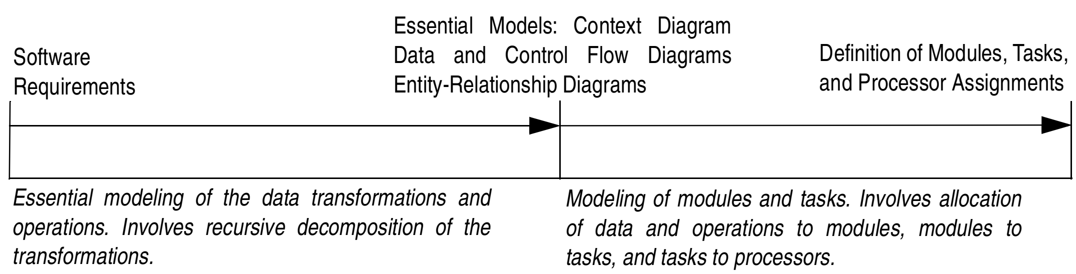


Figure 2.1 Transformation from requirements to design via a structured analysis and design approach.

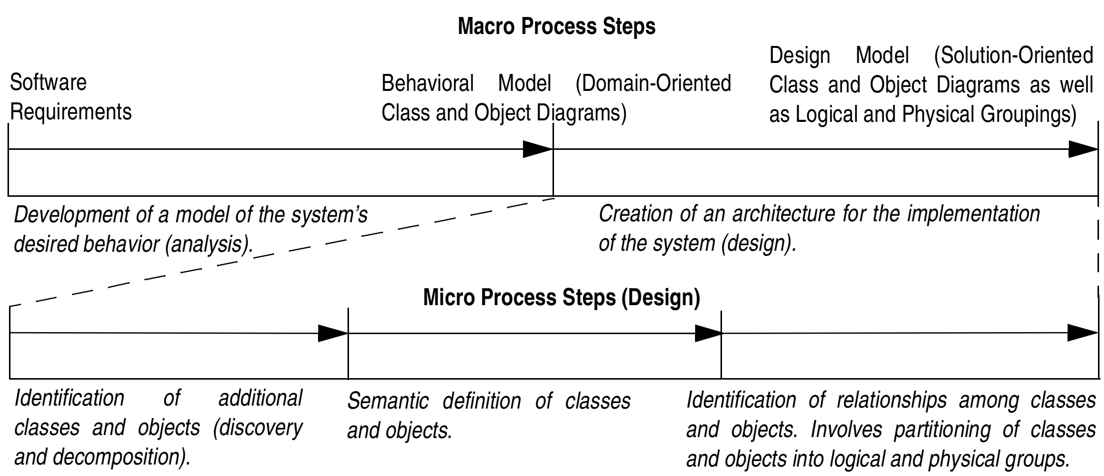


Figure 2.2 Transformation from requirements to design via an object-oriented approach.

The identification of user-oriented objects (classes) and solution-oriented objects (classes) involves abstraction as well as partitioning. Figure 2.3 show the transformation from a user-oriented to a solution-oriented view of a generic control system. In the solution space, objects (classes) communicate with the external sensor and actuator objects shown in the user view. The designer may decide to decompose the user-oriented sensor, actuator, and control objects into generic super-objects and into more specialized objects which hide the details of the specific sensors, actuators, and controllers. The designer may also decide to logically group the sensor, actuator, and controller objects that will collaborate in the real system as shown by the hatched boxes in Figure 2.3. For an in-depth report on an analysis using the Booch method of the air traffic control domain, another type of high-assurance application, the reader should see [159]. This report outlines the results of a research project which focused on "Data Modeling for Advanced Flight Plan Processing Systems" and which was performed jointly by the Computer Information Systems group at the Technical University of Berlin and the EUROCONTROL Experimental Center.

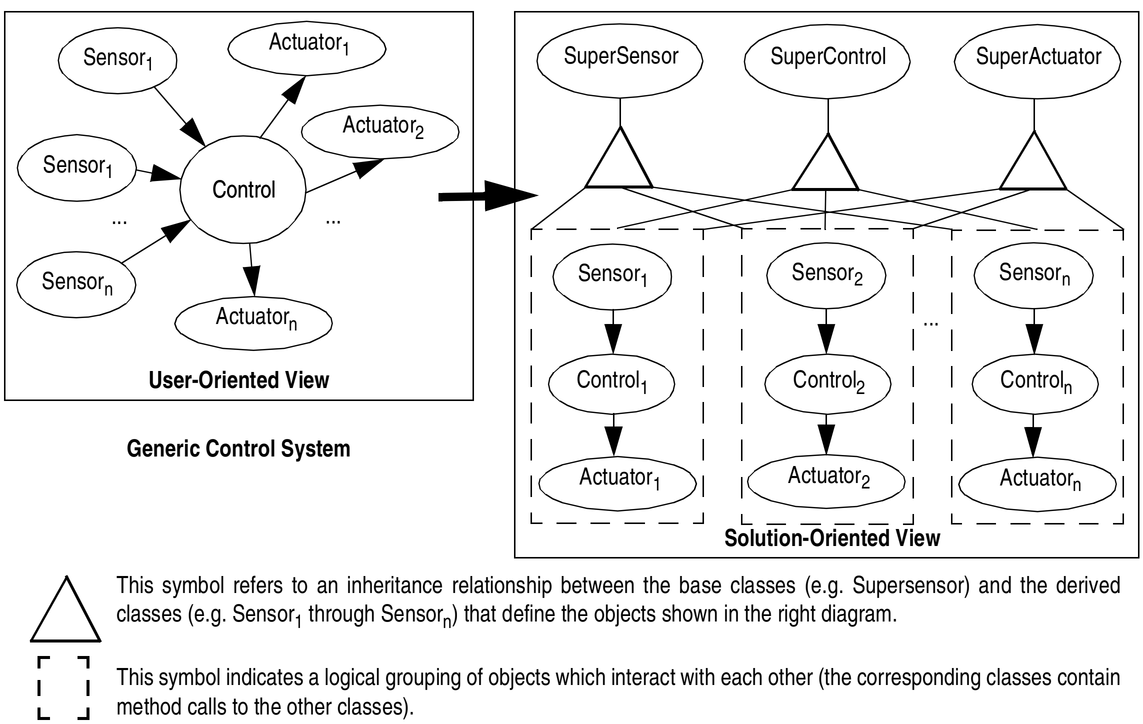


Figure 2.3 Transformation from user-oriented to solution-oriented objects and classes.

Throughout the design process, the designer must repeatedly partition the design space and decide between alternative designs. As shown in Figure 2.4, Part (a), a designer may alternatively decompose a communication component into subcomponents defined by performance characteristics or by functional use. A partition according to performance may simplify the mapping to execution-time tasks, whereas a partition according to functionality may best localize logic that differs for each type of device.

Figure 2.4, Part (b), demonstrates three alternative levels of decomposition. A motion control system consists of three basic types of control: (1) program control (background of dots) to download, store, decode, and execute user programs that direct the motion of a particular device; (2) motion control (background of slanted lines) to plan the path and perform the low-level servo-control of the motor that controls the movement of the target equipment; and (3) system control (plain background) to handle system functions such as monitoring the proper operation of the system (watchdogs), start-up/shutdown, and task management. The designer must determine the appropriate level of decomposition into subsystem components. The designer then maps these components into modules to be implemented and possibly into packages of modules for compilation. Components such as these may also represent building blocks to be reused across similar applications [120]. The next section discusses software qualities that determine whether or not a design is good.

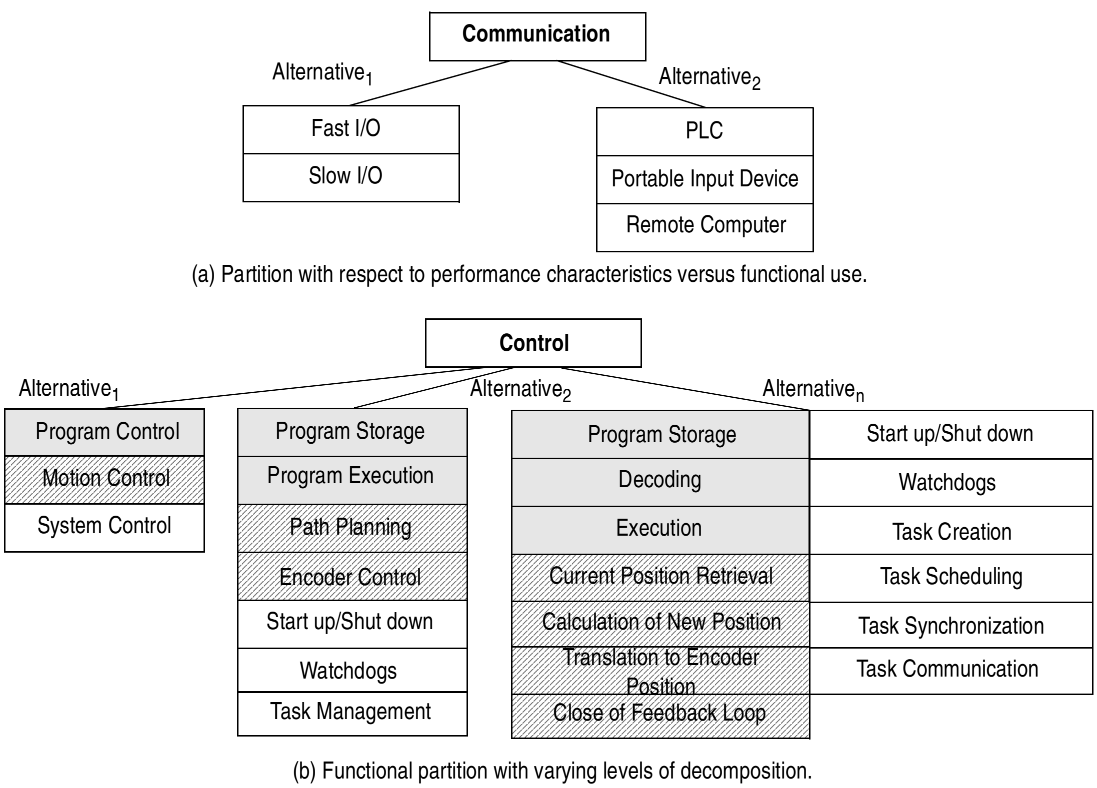


Figure 2.4 High-level design alternatives in a motion control system.

2.3 What are "good" software designs?

As was stated in Chapter 1, "good" software designs result in software that correctly performs the required behavior while satisfying design objectives. The designer's goal is to generate designs that result in software with maximum quality and minimum cost of implementation and maintenance. There has long been a debate about what software quality means with general definitions such as fitness for purpose and more specific definitions in terms of *quality attributes* such as reliability, maintainability, and usability.

This research targets the software qualities of *changeability* and *reusability,* both of which contribute to *evolvability*. Software evolution involves changes in software structure and meaning over time to satisfy changes in application requirements. The problem is that the process of changing software often involves extensive impact of change (non-localized change) and substantial manual effort. The process can be costly and error-prone [26]. "Reduce the impact of change" means that a software solution can be modified in a reliable, timely, and cost-efficient manner. Software reuse is broadly defined as the use of engineering knowledge or artifacts from existing systems to build new ones. The importance of this technology increases as designers develop reusable artifacts that provably improve software quality as well as reduce development costs and time-to-market. The goal is to design reusable software components that reduce the impact of making changes to software solutions by localizing change.

It may seem easy to design software components that localize change. There are numerous language features and design strategies for localizing the detailed logic related to ways in which the software is expected to evolve. Some common design strategies to facilitate change are those listed below.

- Hide the details of how something works and what data is manipulated inside of components such as modules or classes.
- Design standard component interfaces whose parameters and types do not change.
- Use generic data types or user-defined types whose actual types are resolved by the compiler.

Likewise, the design of reusable software artifacts is a popular development objective that is easy to understand. For instance, the concept of *product-line software architectures* embodies the reuse of software artifacts across similar types of applications. The designer partitions the solution into components that are useful for a class of applications.

Current technologies attempt to simplify the creation of reusable software components. For instance, a designer following an object-oriented design approach can define base classes whose generic method logic and data is reused via inheritance across the derived classes. Alternatively, the base class may contain the declaration of prototypes for methods which are defined in the derived classes. This enables the reuse of logic in objects that activate the methods even though the implementation of the methods may vary (e.g., to allow for differing levels of performance).

In practice, the determination of reusable components that localize change is not automatic, is dependent on human judgement, and is therefore difficult to achieve. The next section will make more clear why the myriad software design decisions make the process difficult for humans.

2.4 Why is software design difficult for humans?

For most applications, there is no automated way to generate a design that satisfies requirements and design objectives; so the designer must imagine a good software solution and produce the directions for building it. This creative process includes both decomposition and synthesis. Separation into parts (decomposition) enables the separation of concerns among parts of the software solution. Separation of concerns allows the software designer to focus on fewer details at one time and helps the designer to avoid coupling between unrelated solution elements. Decomposition of the solution into parts supports the specification of software components that can be implemented concurrently by different programmers. The process of decomposition requires the human to determine the constituent the parts and to decide when the separation into parts should stop.

Synthesis is the composition of parts into a whole. As discussed in Section 2.2, partitioning or grouping of solution elements into components is a type of software synthesis. In the generation of new software architectures, the designer must group solution elements into components that will result in software with qualities such as reusability and maintainability. Following a component-based software development process such as the one outlined in Section 1.2, the designer composes the specifications of both existing and new components to generate a design. With a process based on reuse, the designer critically selects existing components that perform the desired functions with the desired level of performance. The designer must also decide if the interfaces to these components are compatible. The decomposition and synthesis of software designs require human creativity and judgement.

The ability to make decisions that will result in software with the desired qualities is a primary skill of a good software designer. Skills which help the designer to make good decisions are those listed below.

- Abstraction (hiding information that is irrelevant from a particular view)
- Analysis (studying the software behavioral model to identify the primary behaviors; determining the constituent parts of a software solution)
- Attention to detail
- Complexity management

The difficulty lies in consistently making good design decisions without the aid of systematic, precise, and proven steps for generating good designs: software design is still primarily an “art” rather than an “engineering discipline.” Current software has too many surprises due to immature or poorly integrated software domain sciences, construction principles, and engineering processes [17]. Another problem is the difficulty of determining whether or not a particular design can (if implemented correctly) result in software with the desired qualities, a research topic that will be discussed in Section 3.3.

Guidelines rather than automated design assistants are the designer’s tools. For instance, suppose the designer is following an object-oriented design approach. The designer must determine the following elements of the software architecture, which are a refinement of the generic architectural decisions listed in Section 1.1.

- Types and number of objects (components)
- Interactions between objects via method calls (component interactions)
- Names of classes, names and types of methods and parameters (component names and interfaces)
- Description of logic for methods (component behavior)

Figure 2.5 and Figure 2.6 present guidelines for determining classes (definitions for software objects) that demonstrate high cohesion and low coupling concepts. Though applied intuitively by expert designers, these guidelines do not show the designer when and how to precisely apply them.

Principles:
- Classes should relate to something in the real world.
- Classes should have a well defined purpose.
- Classes should have distinct purposes.
- Objects of the same class should behave identically.

Reason: Clarity of definition promotes ease of understanding and thereby simplifies development and maintenance.

Figure 2.5 Object-oriented design principles to define classes that exhibit high cohesion.

Principles:
- Classes should contain objects of as few classes as possible.
- Classes should call methods of as few classes as possible.

Reason: Classes with minimum knowledge of each other and therefore low coupling reduce the complexity of development and maintenance.

Figure 2.6 Object-oriented design principles to define classes that exhibit low coupling.

The next section demonstrates the types of decisions made by designers following an object-oriented design approach. The process for defining and refining classes presented in Section 2.5 makes the “when” and “how” of designing classes more precise; but the process still requires the designer to determine the elements of the design space (solution elements and feasible organizations of them) and to select a good or optimal design in this space. The primary theme in this dissertation is that making the choice of solution elements and their organization more precise, objective, and systematic would more consistently result in good designs. Likewise, the type of precision and objective criteria for determining good designs would enable semi-automation of the design process. The examples in the dissertation follow an object-oriented approach because this style of architecture is widely used today in industrial as well as research software development. The theme and ideas presented in this dissertation apply to design in general.

2.5 Examples of Software Design Decisions

Refinement, the iterative restructuring of classes to achieve design objectives such as high cohesion and low coupling, is an essential part of an object-oriented design process. Figure 2.7 lists ways to reorganize classes to obtain different groupings. One type of refinement involves the technique of inheritance in which derived classes inherit the properties of one or more base classes. The actions to take (how to restructure classes) are clear, and the type of results to be achieved from these refinement steps are apparent. But the decision of when and how to precisely apply them is not obvious and requires human judgement.

Types of refinement:
- Change a class into an object or vice versa.
- Assign methods to classes differently.
- Combine or split classes or methods.
- Rename classes or methods.
- Make classes into base classes and create derived classes, or remove base classes and make derived classes as new potential base classes.

Figure 2.7 Object-oriented refinement of classes.

For instance, the designer must answer questions such as those which follow to refine the classes in a way that satisfies the design objectives.

- The refinement of which classes will help to achieve the design objectives and why?
- Which type of refinement should be applied to which classes and why?
- Does the application of some types of refinement help achieve some design objectives while hindering the achievement of others?

The steps shown in Figure 2.8 represent a typical process for transforming a set of requirements and behavioral specifications into an object-oriented design. Steps 3 and 5 specifically direct the designer to refine the classes as discussed in Figure 2.7. The goal is to achieve high cohesion and low coupling.

1. Identify nouns and verbs in the requirements and behavioral specifications that describe the real world entities as well as the functions to be performed or the actions to be taken by the software.
2. Make the nouns into classes or objects and the verbs into methods.
3. Think about the object-oriented principles related to cohesion and refine the classes.
4. Think of scenarios by which collections of the classes can accomplish the functions outlined in the specification. Identify the sequence of method calls and selection of classes that contain or use objects instantiated from other classes.
5. Think about the object-oriented principles related to coupling and refine the classes.
6. Repeat steps 3-6 until done.

Figure 2.8 Object-oriented design process.

The problem is that the directives "think about", "think of", and "refine" based upon established principles do not tell the designer precisely what to think and how to act appropriately on any thoughts. Likewise, how does the designer decide when the refinement process is done, a primary decision required for step 6? By default, the stopping condition is when no additional improvements via refinement are apparent. Obviously, the process outlined in Figure 2.8 is neither precise nor guaranteed to generate an optimal design.

The design of software to control a simple microwave serves as the example to illustrate concepts about design for parts of this dissertation. This is a convenient example because the function of a simple microwave is generally well understood. Likewise, the scope of the problem is appropriate not only for discussion in a dissertation but also for the design of an experiment to test the effectiveness of the experimental design approach presented in Chapter 5. Figure 2.9 shows the mock-up of the user interface to the example microwave.

There are six function buttons (start, stop, defrost, power, reheat, and timer), a display consisting of five characters, and ten keys to input the digits 0 through 9.

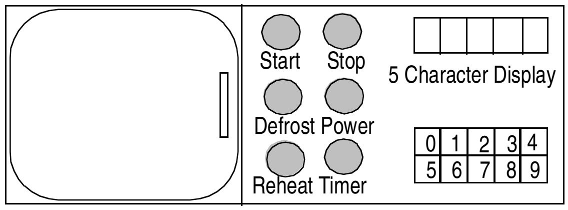


Figure 2.9 Mock-up of the user interface to a simple microwave.

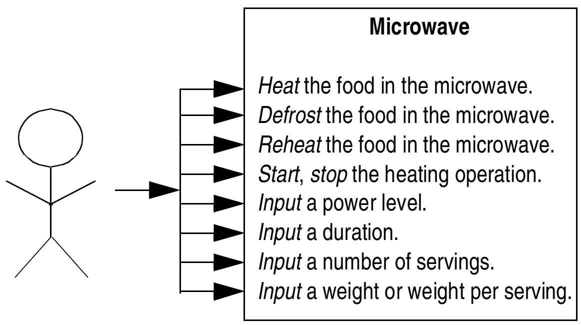


Figure 2.10 User interactions with the simple microwave.

Via the buttons, keys, and display described above, the user interacts with the microwave as shown in Figure 2.10. There are three basic types of heating operations.

1. **Straight Heating**: The user inputs a power level and a heating time or duration.
2. **Defrosting**: The user presses the DEFROST button and inputs a weight. The microwave software determines a power level and a duration to defrost the food of a specified weight.
3. **Reheating**: The user presses the REHEAT button and inputs a number of servings as well as the weight for each serving. The microwave software determines a power level and a duration to reheat the specified servings of a particular weight.

The user presses the start or stop buttons to start or stop any of the three heating operations. The digit keys enable the user to specify the power level, duration, number of servings, weight, or weight per serving. The display prompts the user for the value to be input and provides the user with status information such as error messages if an improper value is entered. For instance, the available levels of power for the microwave may be specified as one through nine. The entry of a value less than one or greater than nine for the power level would result in an error message on the display.

In addition to the user interaction with the microwave, the software designer needs to know about the hardware devices that control the microwave. Microwave ovens consist of a container in which food can be heated by penetrating it with microwave radiation. In this example, the radiation comes from four identical power sources. There are two primary controls for the oven: (1) a timer, which limits the duration of the radiation; and (2) a power sensor, which checks the level of radiation actually generated. Since radiation is harmful to people, a door sensor monitors whether the oven door is closed. If it is not closed, the microwave oven must shut off the radiation immediately. In essence, the software for controlling this microwave must direct the following devices: four identical power sources, a power sensor, a door sensor, and a timer.

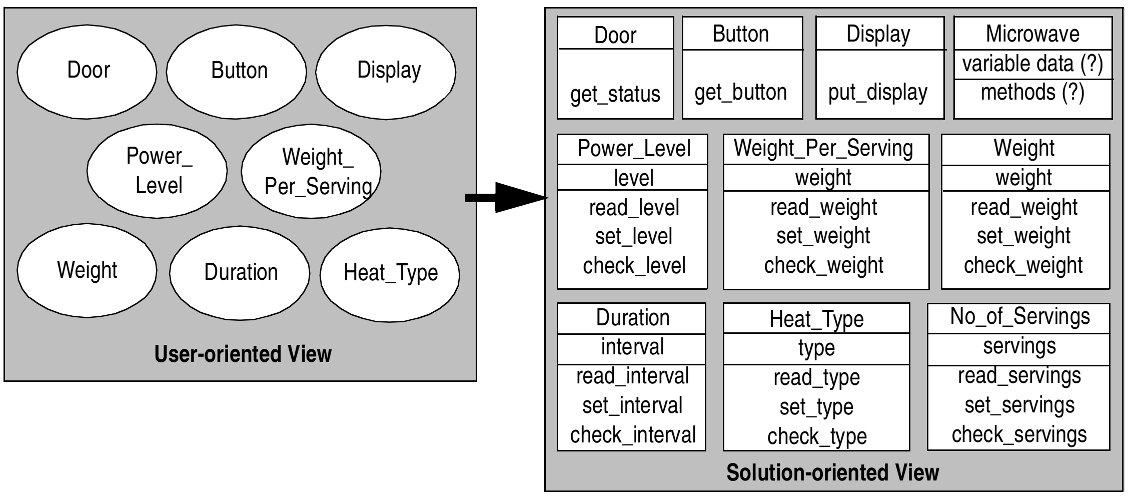
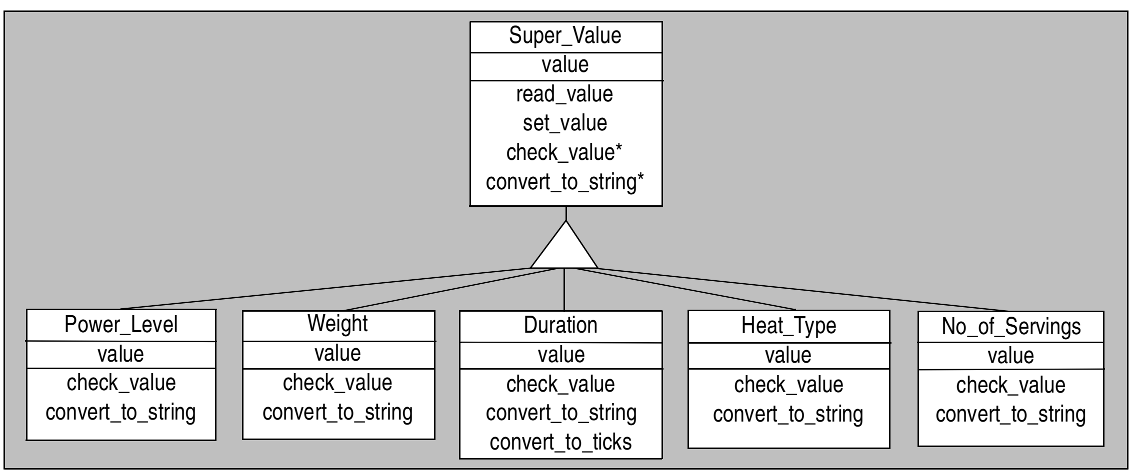


Figure 2.11 Transformation from user-oriented to solution-oriented objects and classes.

The first step performed by the designer using an object-oriented process as outlined in Figure 2.8 is to identify the nouns and verbs in the requirements and behavioral specifications that represent “real-world” entities. From the diagrams of the microwave in Figure 2.9 and of the user interaction with the microwave in Figure 2.10, the designer could identify the user-oriented entities as shown in the left diagram of Figure 2.11. In this case the designer chose to create the abstraction “button” to represent each button or key that can be pressed. The designer could have chosen to represent each button or key as a separate object. The reason for this choice may be that the designer learned from the requirements specification that the interface to the actual keyboard is a call to a software routine called GET_KEYBOARD which is supplied by the hardware manufacturer. Likewise, the designer chose to create one object called Heat_Type to represent the type of heating operation that could be input by the users.

The second step in the object-oriented design process is to convert the nouns (user objects) to classes and the verbs to methods as shown in the solution-oriented diagram of Figure 2.11. Here again the designer may or may not think extensively about the required behaviors. The designer could have created only an "input" method for each type of value to be input by the microwave. Instead, the designer understood that the value must not only be input or read but must also be checked for correctness before being set to control the microwave. At this point in the design process, the designer had not thought extensively about how the software should actually perform a heating operation; therefore, the details of inputting and performing a heating operation are hidden in the Microwave class. The data and methods for this class are not defined completely.

Steps 3-6 involve refining the classes to provide sufficient detail so that the solution can be implemented. While doing this, the designer should try to create classes with high cohesion (high degree of relatedness between the purpose of the data and methods) and with low coupling (minimal interaction between classes). The creation of base and derived classes generally occurs during the refinement process. For instance, the Power_Level, Weight, Duration, Heat_Type, and No_of_Servings are derived classes that could inherit the properties of a base Super_Value class as shown in Figure 2.12. The burden is on the designer to identify this abstraction and to determine if it will help to improve the design.



The derived Power_Level, Weight, Duration, Heat_Type, and No_of_Servings classes inherit the attributes of the base class, Super_Value. *With the implementation language C++, the check_value and convert_to_string methods could be virtual functions that are defined in the classes derived from the abstract class Super_Value.

Figure 2.12 Refinement via inheritance.

Some classes are better composed of two or more classes for ease of understanding and implementation. For example, the Microwave class currently encapsulates most of the methods needed to coordinate and perform a heating operation. In this example, the requirements and behavioral specifications explain that the software should perform the following four basic functions, each of which will build upon each other.

- **Driving the electronics:** This part of the software communicates directly with the hardware. It gives instructions to the hardware and obtains status in return. From a high level viewpoint, the drivers use electronic connections between the processing unit that runs the software and the rest of the hardware.
- **Controlling the electronics:** This part of the software puts together the instructions to the hardware that the drivers then pass on to the hardware.
- **Scheduling of heating operation:** This part of the software translates the button pushes of the users into instructions that match the hardware's capabilities.
- **Managing the user interface:** This part of the software directly receives user input through the keyboard and posts output messages on the displays.

Directions such as these may guide the designer to select a layered software architecture in which each layer (defined as a class) communicates only with the layers (classes) above and below [133] as shown in Figure 2.13.

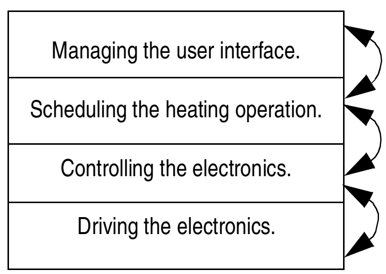


Figure 2.13 The four basic functions of the microwave software organized in a layered architectural style.

The problem is that the requirements and behavioral specifications may not organize the description of the functionality in a way that guides the designer towards a known style of architecture; and the designer must still decide if the design should encapsulate each function as a separate layer(s) or class(es). The designer for our example chose to combine the functions of managing the user interface and of scheduling the heating operation into one class called the User_Interface class. The resulting decomposition of the Microwave class yielded the User_Interface, Control_Electronics, and Drive_Electronics classes shown in Figure 2.14.

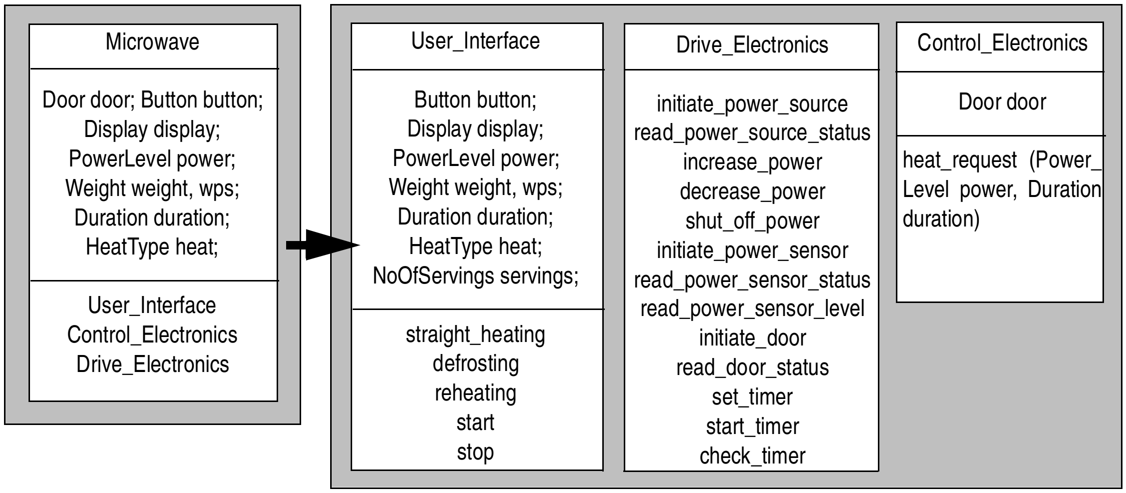


Figure 2.14 Refinement via decomposition of the Microwave class.

Another possible refinement would be the decomposition of the Drive_Electronics class into separate classes for the hardware devices that it directs as shown in Figure 2.15. If this decomposition is made, then the designer must decide if the Control_Electronics class should communicate directly with the new device classes or should funnel heating directions through a Drive_Electronics class. Another question is whether or not an abstract device class or an abstract class for each type of device would be useful. In our example, the designer may choose not to create separate device classes because the lowest-level logic for controlling the devices is provided by the manufacturer of the devices. The interface to any of the manufacturer device drivers is through a standard routine called CALL_HARDWARE.

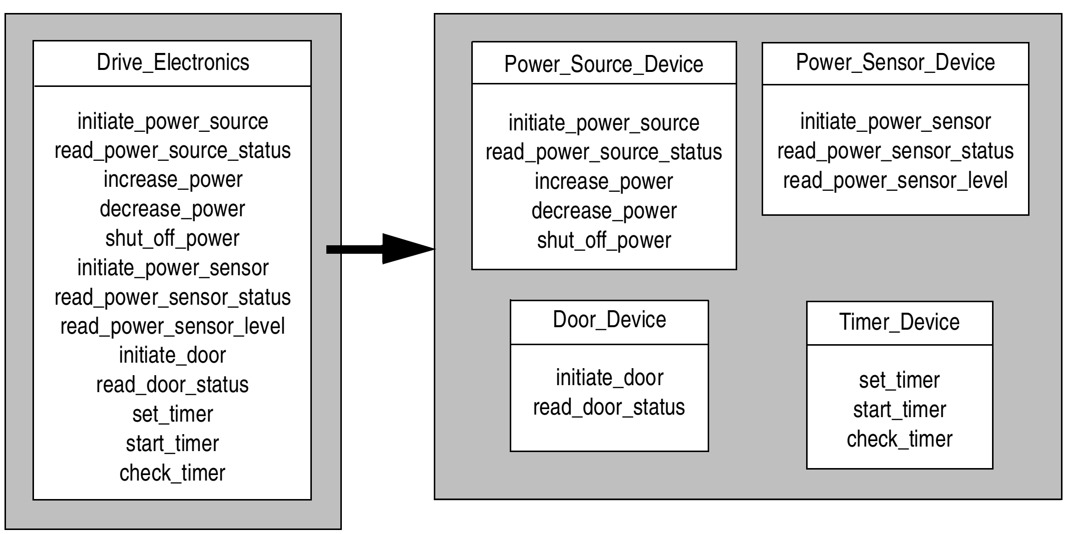


Figure 2.15 Further decomposition of the Drive_Electronics class.

Object instantiation enables the creation of multiple objects of the same type. Classes encapsulate data and methods that are appropriate for the particular type of object. For example, the Weight class includes the weight variable and the methods *read_weight*, s*et_weight*, *check_weight*, and *convert_to_string*. Here there is no assumption of an abstract class Super_Value. The defrost operation requires a total weight to be input, whereas the reheat operation uses a weight per serving. The designer could represent these two values by a Total_Weight object and a Weight_Per_Serving object, both of which are instances of a Weight class as shown in Figure 2.16.

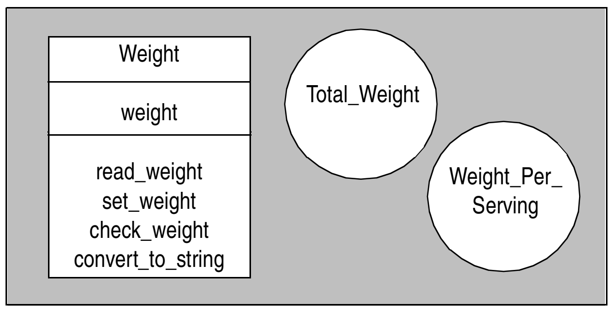


Figure 2.16 Multiple objects of the same type.

Steps four and five in the object-oriented design process involve the development of scenarios in which the defined objects interact to perform the required functions as well as the refinement of classes to de-couple objects (classes) which should or should not communicate. The designer must decide which objects will communicate (which classes will include calls to methods defined in other classes). In the microwave example diagrammed in Figure 2.17, the Control_Electronics object may communicate with the User_Interface object as well as the device objects. Discussed previously, a refinement would be to create a Drive_Electronics object that hides the interface to device drivers from the higher-level control of a heat operation that is encapsulated in the Control_Electronics object

.

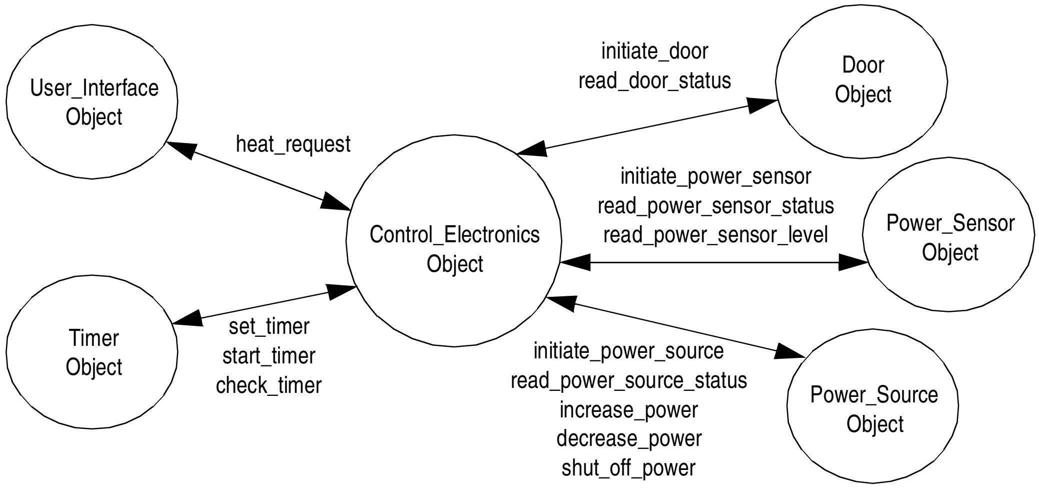


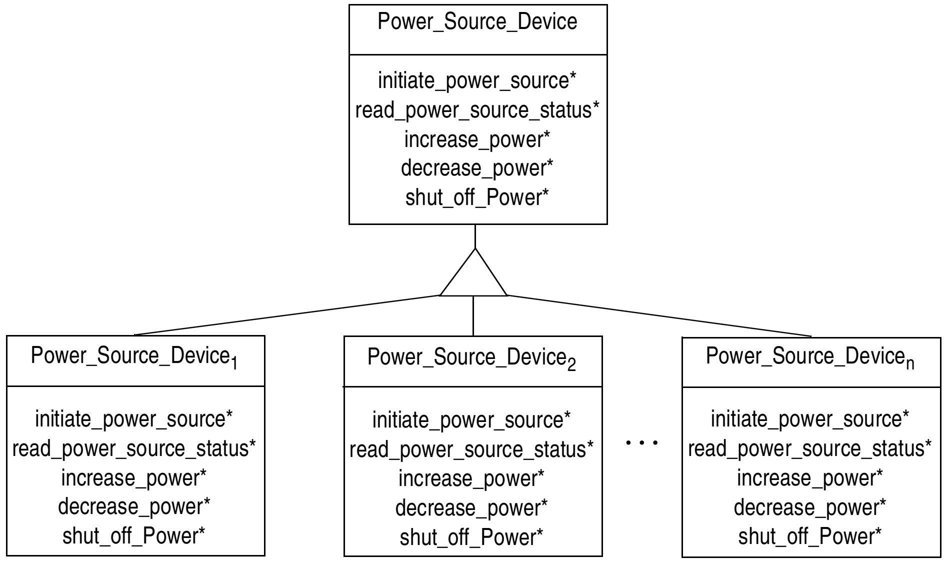
Figure 2.17 Object interactions.

When partitioning a software solution into components, the designer should consider design objectives such as those that follow.

- Product-line evolution (change via modification of existing similar solutions)
- Reuse of generic components (even across unrelated software solutions)
- Flexible performance (adaptability via alternative performance levels, concurrency, or distribution)

The extent to which the maintainer must examine and modify different parts of the software depends upon the type of change and the organization of the software. Therefore, the designer would like to localize the parts of a software system affected by a change because it is easier to modify software when the related changes are close together. The maintainer is less likely to introduce errors into unrelated parts of the system if the unrelated part is logically or physically separate from the part affected by the change.

Designing for change can be done by creating generic base classes which define the types of methods and data whose implementations occur in the derived classes. For instance, a generic Power_Source_Device, as shown in Figure 2.18, enables manufacturers to create device drivers for different power sources that have the same interface and basic functions. Software that interfaces with a power source device need not change even though the power source device and its driver may change.



The calls to the generic methods are reused across different types of power source devices. The power source device drivers (defined in the derived classes) may be easily exchanged for migration to different processors. *virtual functions in C++

Figure 2.18 Design for reuse and change.

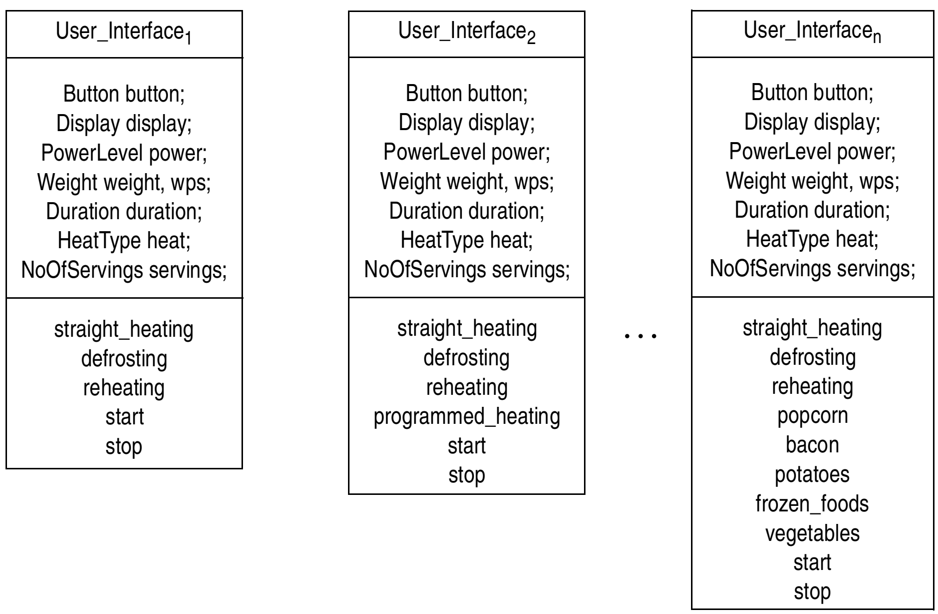


Figure 2.19 Design for functional change.

Likewise, the designer may want to create a design that can be easily changed to include new heating operations. The User_Interface class encapsulates the logic to convert a key sequence for a particular heating operation to a power level and duration. New microwaves would contain different versions of the User_Interface class, as displayed in Figure 2.19, and would reuse the original Control_Electronics class. Designing for product evolution requires the designer to think ahead of the ways in which the product would change over time and to localize those parts that would need to change or which would not change.

The designer can partition the solution into components, such as those listed below, that can be used to build solutions with flexible and reliable run-time performance.

- Parameterized components that each perform a particular function with differing levels of performance.
- Separate components that perform similar functions, each with a different level of performance.
- Components that encapsulate logic that can execute concurrently and be distributed across multiple processes and processors.

In future microwaves, the designer may want to allow the user to program the microwave for the next heat operation while the current heat operation is in progress. This use scenario would require the User_Interface and Control_Electronics to execute concurrently as shown in Figure 2.20. Likewise, the user may prefer a single interface that can control multiple ovens. Figure 2.21 shows a distributed design to achieve this behavior. It might be convenient to program from a remote interface a set of ovens to concurrently cook different courses of a meal at predetermined times.

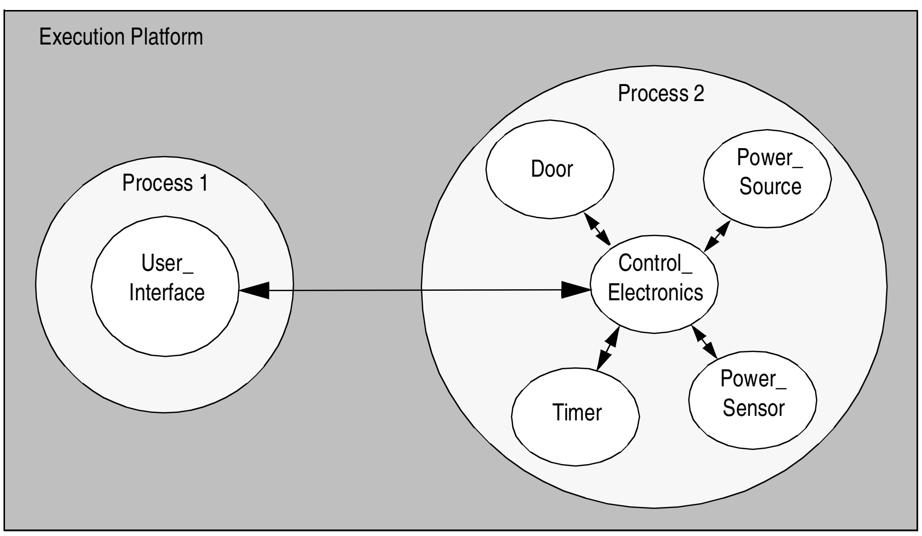


Figure 2.20 Design for concurrency.

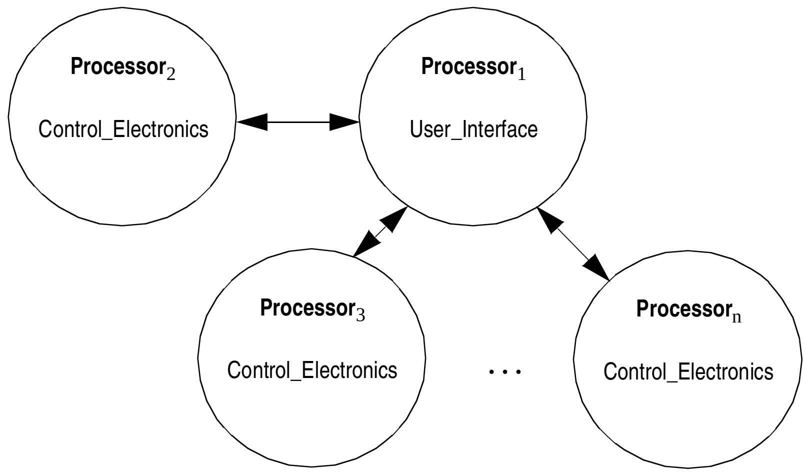


Figure 2.21 Design for distribution.

As has been demonstrated in this section, design guidelines help the designer to determine how to partition the solution into components (classes in object-oriented design). But they are not precise steps; nor are they measures for determining the goodness of a design. Ultimately, most architectural decisions are based on the judgement of the designer. These decisions are critical to the design of correct and reliable software systems that can be easily maintained and adapted for new uses. The next section describes the nature of high-assurance systems for which design quality is essential.

2.6 What are high-assurance computing systems?

High-assurance computing means that there is a high level of confidence that the following statements about a system are true:

1. The system specifications are complete, consistent, and relevant.
2. The implemented system conforms to the system specifications.

The term system includes not only the system hardware and software but also the human interaction with the physical system, especially in the context of security. A quality of service (QoS) model for high assurance computing designed to satisfy these two aspects of high assurance encapsulates the attributes of *timeliness*, *precision*, and *accuracy*. To achieve a practical engineering and metric-oriented approach to system development, all inputs/output of a system are defined by these three attributes. For an input/output to be completely specified or measured, there must be a value as well as a range of permissible values for each attribute [102].

Along with a high degree of conformance to valid system specifications is the need for availability and the lack of tolerance for error. A high-assurance system must perform reliably and be highly available, fault tolerant, and secure. Castaneda adds that high-assurance systems for the U.S. Navy must also be operationally suitable. When operated and maintained by the expected number of properly experienced fleet personnel, an operationally suitable system is reliable, maintainable, operationally available, logistically supportable when deployed, compatible, interoperable, and safe. Safety means that the system will do what it is supposed to do and NOT do what it is not supposed to do: in other words, the weapon will fire when properly commanded to do so and will not fire otherwise [36].

Example applications which require high-assurance computing systems are command and control for military defense systems, nuclear power plant management, air traffic control, aircraft control, on-line investment reports and transactions, and patient monitoring. As users increasingly rely on computing systems to live healthy, happy, and productive lives, many other applications will increasingly require high-assurance computing. High-assurance computing may become the expected mode of operation and maintenance for most if not all computer systems.

2.7 Why is software changeability important for high-assurance software systems?

The ad hoc way in which software engineers develop and maintain high-assurance software systems is often difficult, time-consuming, and costly [114]. As early as 1973, Boehm commented that organizations should concentrate more on reducing software life cycle costs rather than on reducing software development costs. He noted that the cost of maintaining Air Force avionics software was approximately 50 times higher than the cost of its development ($4000 per instruction for maintenance versus $75 per instruction for development in 1973 dollars). The high cost of software maintenance is still a problem; and to make matters worse, the maintenance process is frequently error-prone [26]. The delay of the first NASA Space Shuttle orbital flight due to a software problem exemplified the difficulty of changing software. In his discussion of the Shuttle incident, Garman emphasized a key idea: though software is readily changed, it is via change that software is the easiest part of a system to compromise [52].

Software reuse is a technology intended to improve software quality as well as to reduce the cost of development and time-to-market [48]. Some researchers argue that reuse does not guarantee the safety of a software solution. For instance, Leveson states that safety is not just a property of the software itself but also a property of the software design and environment in which it is used [105]. It appears to be an open question as to whether or not it is easier to construct a safe software system totally from scratch or from parts of a "safe" existing software solution. Software reuse alone may not solve the costly problem of maintaining high-assurance software systems.

Design for changeability has its roots in the principle of modularity, a concept described by the currently popular term "component-based development." The basic idea is to group the required data and logic for a solution into modules or components that can be easily reused as is, modified, or replaced. The question is how to decide which solution elements should be put together into the same modules. The objective is a grouping which simplifies and reduces the error associated with maintaining the target software system. The research approach described in Chapter 5 strives to achieve this objective.

2.8 Why is software design for high-assurance software systems complex?

Two primary factors make the design of high-assurance software systems a complex process. One factor encompasses the specification and verification of designs that satisfy the types of stringent expectations for high-assurance system discussed in Section 2.6. The software designer must carefully specify the organization and content of a software system that will behave precisely and accurately as specified in the software requirements. The designer must also determine a way to establish a high degree of confidence that the design specifies a software system that can satisfy the requirements. In particular designing high-assurance systems that satisfy stringent timing constraints is difficult [33].

The other factor which contributes generally to design complexity is the "integration factor." Integration involves the connection of separate components, hardware and software, to create a system of interacting parts. The connections may be physical or logical. The difficulty lies in the fact that connections in combination with components must together satisfy the stringent expectations for the target high-assurance system. Some interconnects as well as components may have differing levels of criticalness, a feature which compounds the difficulty of verifying the system. A panel of researchers from industry and academe identified the following types of inconsistencies that can arise in the integration of high-assurance systems [58].

- Dependability requirements

(e.g., The maximum allowable frequencies for certain types of failures defined as a product requirement may differ from the allowable frequencies specified as a Capability Maturity Model or CMM process requirement for a component [118]).

- Exception handling

  (e.g., The actions of an operating system or scheduler may override specific exception handling provided within a software component.)

- Assumptions about “safe states”

  (e.g., A system-level requirement such as “maintain prior value” may not propagate into a component-level requirement such as “maintain prior valve position.”)

- Verification practices

  (e.g., Verifying an entire system may be difficult due to variances in testing practices at the component level.)

Assuming that real-time software systems are more likely to be predictable and dependable if they are simple [56], there is a need for engineering techniques that reduce the software complexity of high-assurance systems.

The next chapter discusses more specifically research in software architecture generation and evaluation that relates to the research problem.

3 Research in Software and System Architecture Generation and Evaluation

The motivation for this thesis is the search for an analytical way to generate a software architecture that localizes change. The input would be a requirements analysis, known design constraints, and potential changes to the problem or solution. The output would be a specification of software components that would either be selected for reuse from a database of existing components or built from scratch. The ideal solution would be a way to automatically apply Parnas' guidelines for modularizing complex software systems [116].

Though fully automatic generation of a software architecture that can be easily changed to satisfy product evolution is an ongoing goal, research in several areas sets the foundation for its accomplishment. This chapter will briefly overview key work in the following areas which have the potential for advancing the fields of software architecture generation and evaluation as well as validation of software design approaches.

1. Modularization to achieve software design objectives such as modifiability (Section 3.1)
2. Reuse of software design knowledge via styles or patterns (Section 3.2)
3. Evaluation of candidate software architectures (Section 3.3)
4. Generation of good designs via search of a design space (Section 3.4)
5. Reuse of solutions through automatic system generation (Section 3.5)
6. Design and composition of software systems from reusable components (Section 3.6)
7. Empirical research in software engineering (Section 3.7)

Table 3.1 explains the concerns and issues that each area listed above shares with the thesis research.

Table 3.1 Concerns and issues shared by each related research area with the thesis research.

Related Research Area	Concern or Issue Shared by the Related Research Area and the Thesis Research
Modularization to Achieve Design Objectives	Shares principles of decomposition and information hiding to achieve evolvability.
Reuse of Software Design Knowledge via Styles or Patterns	Discovery and reuse of design knowledge (e.g. the research approach demonstrates a heuristically good and reusable approach to partitioning control flow).
Evaluation of Candidate Software Architectures	Seeks the evaluation of a design with respect to quality attributes that can be measured.
Generation of Good Designs via Search of a Design Space	Uses techniques for representing solution elements of a software design space and for finding a good design based on the relationship between those elements.
Reuse of Solutions through Automatic System Generation	Shares goal of automation.
Design and Composition of Software Systems from Reusable Components	Shares goal of reusable design and code components.

Related Research Area	Concern or Issue Shared by the Related Research Area and the Thesis Research
Empirical Research in Software Engineering	Shares goal of validating software design approaches with respect to their use by software developers.

Note 3.1 The reader should note that in some of the related research areas, the author draws analogies to the design of hardware systems. This is to indicate cases in which ideas researched in hardware design were later explored in software design.

3.1 Changeability Via Modularity

Changeability or modifiability depends primarily on a software system's modularization and is a reflection of the system's encapsulation strategies. The selection or generation of a particular architecture partitions changes into the following three categories:

1. *Local changes* accomplished by modifying a single component.
2. *Non-local changes* accomplished by modifying multiple components while leaving the underlying architecture intact.
3. *Architectural changes* which affect the ways in which components interact with each other and which most likely require changes throughout the system.

Ideally, most changes are local. Reasoning about change demands an assessment of the relationships, dependencies, performance, and behavior of the software components composing the system. Change assessment is the job of the software architect [18]. Parnas and others advocated the use of information hiding via modularization to facilitate ease of change in two well-known papers [116,117].

A seminal study of software modularity was the A-7 Project, conducted by the U.S. Navy to test the effectiveness of "new" design strategies in the development of a software system with demanding requirements such as timing and memory constraints. The idea for the A-7 Project was to publish a complete engineering model with documentation, design, code, methodology, and principles [30]. The target system requirements were those for the avionics software embedded in the A-7E Corsair II, an attack aircraft used by the U.S. Navy throughout the 1960s, 1970s and 1980s. The result of the project was the development of a standard reusable design or *reference architecture* as well as a *domain model* describing the expected behavior and anticipated changes for the A-7E. Two primary lessons from the project are those which follow [19].

1. Information hiding is a viable and prudent design discipline.
2. Careful engineering of different structures of an architecture significantly helps to achieve target

software qualities such as changeability.

The process of determining the modular structure of a software system was primarily an art until researchers and expert practitioners developed more precise guidelines for modularization to achieve specific design objectives. At first, researchers focused on the partition of algorithms to achieve reusable and easily replaceable modules. Some researchers studied ways to decompose mathematical software solutions into software components. For instance, Westerberg's work reflects the common approach of intuitively decomposing algorithms to functional modules. Westerberg's modules were used in various permutations to solve systems of linear equations [158].

Similarly, research in recasting large-effect algorithms into small-effect or partial algorithms inspired systematic decomposition for reuse. The idea is that software modules that encapsulate whole algorithms are more likely to require internal modification in order to be reused. On the other hand, components that contain parts of an algorithm can be alternated to achieve variations in the algorithm's logic or implementation features [155]. Note 3.2 explains what this rationale means for the decomposition of larger-scale software solutions.

Note 3.2 At the system and subsystem levels, software solutions implemented as large-effect components are not likely to be as reusable as those implemented as small-effect components. The resulting research question is how to recast a monolithic software solution into parts whose implementations will facilitate changes to the software solution via replaceable and reusable components.

Today, object-oriented design and programming are popular methods for creating reusable modules (class definitions) that facilitate ease of change. VanHilst and Notkin applied the language features of C++ to structure class definitions in a way that localizes design decisions and changes to the software solution [151]. The current problem is that object-oriented techniques currently guide but do not automate the transformation of requirements to design. Analysis objects do not necessarily map to design objects, and the designer must determine which and how much logic should be encapsulated by each class definition. The resulting decisions may or may not optimize design constraints.

A related thread of research concerns the restructuring of software. For the purposes of maintenance, software restructuring is the modification of software to make the software:

1. Easier to understand and to change.
2. Less susceptible to error when future changes are made.

Software restructuring may also improve the software performance. Restructuring at the module level affects the architecture of the system and is done to preserve or maintain software value. External software value is the cost savings the software provides to the user community; while internal software value is the cost savings due to reduced maintenance costs, increased potential for reuse in other applications, and extended lifetime of use. Software developers can perform software restructuring throughout the development process to iteratively improve the quality of the software [11]. For a collection of earlier papers concerning software restructuring the author refers the reader to [10], and for more recent but related work in software maintenance and re-engineering to [72] and [74].

The research approach proposed in this dissertation more precisely and systematically guides the designer in the process of partitioning logic into modules than the approaches discussed above which are based on intuition and guidelines provided by researchers and experienced designers. The research approach directs the decomposition of a software solution into small-effect data and operations that are perceived or known to be reusable. More importantly, the approach provides the designer with an analytical way to recombine the small-effect data and operations into software components that localize the expected changes to the software solution. The goal is a systematic and semi-automatable way to design reusable software components that localize change to as few software components as possible. The larger goal is the development of an engineering approach to software architecting through the application of science and mathematics.

3.2 Reuse of Design Knowledge Via Styles or Patterns

This section begins with a few notes about the field of system architecture and the psychology of system thinking. This field has its roots in the broader field of systems architecture. A system is in general a collection of different things related in such a way as to produce a result greater than what its parts, separately, could produce. An architecture is the underlying structure of something, whether of buildings, communication net-works, computers, or software (all of which are systems) [121]. As Rechtin discussed the architectural essence of a system is therefore twofold:

1. All systems have subsystems, and all systems are parts of larger systems: the systems world is inherently unbounded.
2. The value added by a system must come from the relationships between the parts, not from the parts alone.

In thinking about a system architecture, the software designer is likely to use previously acquired knowledge about other systems which have similar requirements or constraints. The variety and complexity of the problem (software requirements as well as operating and design constraints) may also require the designer to creatively use problem solving techniques from various fields in science, engineering, and mathematics. The "reuse of knowledge about solutions" to solve other problems with similar features is common to the human experience. Likewise, the process by which a designer's brain searches and explores its database of knowledge to find a suitable structure for a system depends not only on the knowledge base but also on the interconnects between the information. These interconnects may be dependent on the designer's experiences with other software systems. The author suggests that system architecting is an art when it is dependent on the way a particular designer solves problems or thinks. For additional information on ways to think about system structure, the author refers the reader to [9,22,63,122,124,125,156].

Alexander was one of the first modern architects to systematically observe commonalities between the architectures of physical structures such as buildings. His observations included notes about properties or attributes along which physical architectures could be compared and classified. Such properties enable the development of a framework for defining patterns of architecture, where each pattern is the definition of a generic structure with certain basic physical properties and uses [3,4,5]. For instance, an inverted cone is the shape associated with a tepee; whereas a cylinder with a dome-shaped top is associated with an igloo. Notes about the "tepee" and "igloo" patterns would include a description of when, where, how, and why the *architectural style* is most effective for the design of a home.

Early on, the computer design community formulated and applied the concepts of abstraction and styles for the design of digital systems. Bell and Newell defined the different levels of design for digital systems from the most abstract level consisting of a black box description of the high-level hardware components in the system (processors, memories, switches, etc.) to the electronics level, the lowest level that a digital designer usually considers [24].

In his thesis work, Thomas defined a *digital design style* as an abstraction which represents a group (class) of module sets, where each set in the group is more or less the same. A module set is a collection of digital devices, at the register-transfer level, that provide the necessary arithmetic, logic, and memory functions to implement a general digital system. Differences between the module sets in one style with respect to the module sets in another style are due to the different rules for implementing a digital design using the module set and the features inherent in the underlying hardware. Thomas presented an algorithm to automatically select the proper style for a particular hardware design [148].

Likewise, the research and development of styles or patterns of software architecture seek to codify the knowledge of the expert software designer. Novice designers would use a handbook of styles or patterns to guide them in the selection of a type of design that has been used successfully to solve a similar problem (an example is the use of styles to teach software engineers about software architecture [135]). Historically, the concept of style applied more generally to the types of components and component interactions at the system and sub-system levels [133], while a pattern defined the interaction of lower-level design components [50]. The level of abstraction for defining patterns has broadened so that some researchers now distinguish between software design patterns and software architectural patterns [35]. The distinction depends on one's definition of software design and software architecture. Note 3.3 expresses the author's view of software design and software architecture for the purposes of this dissertation.

Note 3.3 In the context of this dissertation, the author views design to encompass all of the steps taken to transform requirements into a description of how to build a software system. In her view, design encompasses the specification of a software architecture for the software solution. For the purposes of partitioning data and logic into components, the author uses the term software design and software architecture synonymously with the understanding that designs can be viewed at multiple levels of abstraction.

The description of a style or pattern includes a definition of the problem, the forces which guide or constrain the solution to the problem, and the solution. The solution or software design consists of components and their interactions. Some researchers are developing languages for describing software architectural styles and patterns. For instance, the ROOM approach defines a model and language for documenting architectural patterns for real-time systems [128]. Shaw and Garlan explored and documented the necessary characteristics of high level languages for describing software architecture [131]. Two representative architectural definition languages for formally describing a software architecture are WRIGHT [7,6] and Rapide [107]. UniCon, an architectural description language, provides a selection of abstractions for the connectors that mediate interactions among components [130,134].

Reuse of design knowledge defined as styles or patterns depends upon the expert software designer or researcher to discover, investigate, and codify good designs. Criteria for determining the goodness of a particular design provide the foundations for evaluating a software architecture, a research area to be discussed in the next section.

3.3 Evaluation of Candidate Software Architectures

An engineering or scientific approach to the selection of a good software structure depends on precise criteria for evaluating a candidate architecture and a systematic method for applying the criteria. Earlier work focused on the comparison of alternative system structures with respect to properties such as system complexity and performance. An example was an approach developed by Stankovic to iteratively restructure the contents of the software modules in a system to achieve the desired levels of structural complexity and run-time behavior [143]. The approach included:

- Vertical migration, a partially automatable process of moving functions to lower levels in the software and firmware hierarchy to improve performance by reducing CPU overhead.
- Analysis of the structural complexity of the resulting system via measurement of the coupling and cohesiveness of the software modules in the system.
- Iterative restructuring of the functions and modules to achieve desirable performance and structural complexity levels. The result was sometimes a trade-off between structural complexity and performance.

Developed more recently, SAAM (Software Architecture Analysis Method) is a technique for manually evaluating a candidate software architecture with respect to quality attributes such as maintainability, security, performance, and reliability. The evaluator defines the target quality attributes within the context of scenarios that describe how the system would behave under given conditions. The evaluator measures an architectural description with respect to an agreed upon set of scenarios that describe how the system will be used. In effect, the candidate architecture is given a qualitative rating based on its perceived ability to support each scenario. Candidate architectures are then compared to each other with respect to how they "perform" for similar scenarios [20].

ATAM (Architecture Tradeoff Analysis Method) is a scenario-based method like SAAM to analyze software architectures. Unlike SAAM, ATAM focuses on multiple quality attributes (currently modifiability, availability, security, and performance). The purpose of ATAM is to locate and analyze trade-offs early during the development of a software architecture. The method guides the user in the formulation of questions during the requirements and design stages that will help to detect and resolve potential risks within a target architecture for a complex software system [80]. Experiences in applying ATAM are reported in [81], and a discussion of tool support for architecture analysis and design appears in [79].

Evaluation of software architecture and reuse of software design knowledge (styles and patterns) complement each other: an attribute-based architectural style or ABAS defines a class of designs along target quality attributes that enable evaluation of the appropriateness of the design for a particular use [90]. The author notes that formal specification of the attributes along with the architectural description of the style may enable the automatic selection and evaluation of candidate styles from a software style database.

Architectural evaluation and design reuse both require the designer to discover or generate candidate software architectures for evaluation or codification. The discovery process, though useful over time, may not be expedient for a particular project. As discussed in Chapter 2, the generation of a software design is primarily a manual and difficult process. A systematic method for determining the design space of requisite solution elements and criteria for partitioning them into components would simplify the generation of candidate designs. The next section discusses the generation of good designs via the construction and search/analysis of a design space.

3.4 Generation of “Good” Designs Via Search of a Design Space

A design space identifies key architectural choices, classifies the available alternatives, and provides a framework for the selection/generation of an appropriate software architecture. The design space may describe architectural properties relevant to a particular class of systems, e.g. user interface systems, and is, in general, useful as a shared vocabulary for describing and understanding systems. Each dimension of a design space describes variation in one system characteristic or design choice. Values along a dimension correspond to alternative requirements or design choices. A specific system design corresponds to a point in the design space that is identified by the dimensional values that correspond to its characteristics and structure [94].

A design space is the composition of a *functional design space* and a *structural design space*. Some dimensions reflect requirements or evaluation criteria (function and/or performance), and others reflect structure (or other available design choices). A design dimension may consist of discrete values rather than a continuous scale, and the dimensions may not be independent. Correlations between the dimensions motivate the definition of design rules that describe appropriate and inappropriate combinations of design choices. In an experiment to validate the effectiveness of a design space and associated rules for determining appropriate designs for six user interface systems, Lane found that the experimental designs determined by applying the design space rules were moderately to substantially representative of already existing designs for these systems. With respect to most system characteristics, the design space and rules adequately reflected the design choices and constraints considered by designers of the six test systems [93,95].

In a related project, graduate students in software engineering formulated a process to apply Quality Function Deployment (QFD) to the analysis of design spaces. QFD is a quality assurance technique that helps translate customer needs into technical requirements needed at each stage of product development. While the design space organizes the choices available for a particular design into a hierarchy of dimensions and alternatives, the QFD process represents and weights (quantifies) a particular set of these choices, one from each dimension of the design space. By explicitly including all alternatives of each dimension in the QFD framework, each alternative can be analyzed for its ability to meet the needs of the customer and for its correlation to alternatives in other dimensions [12,13].

Automatic generation of a good design from a design space requires the following formulation of a search problem and solution environment.

- Definition of a finite design space
- Determination of rules for identifying viable and non-viable choices in the design space
- Definition of a cost function for comparing the goodness of different points (alternative designs) in the design space
- Development of an algorithm for traversing the design space in search of good designs
- Representation of the design space, rules, and search algorithm in a programmable form

Definition of the finite design space is difficult for an arbitrary set of software requirements, hence current work in design spaces has focused on the specific concerns for classes of applications, such as user interface systems as described in Lane's work. Likewise, the design rules inherent in the design spaces help the designer to make high level decisions about basic structural and performance properties of the solution; but they do not guide the designer in finding the best partition of the required data and logic into components.

Though the goal of automatic synthesis of a good software architecture is similar, the research described in this dissertation takes a more generic approach to defining the design space. The research approach guides the designer in defining a design space of basic solution elements that map directly to the required behavior for the software system. The designer then refines these elements based on their potential for reuse in other software systems with similar requirements. The result is a set of solution elements/sub-elements that are then partitioned to localize change. Similarly the hardware-software codesign community partitions basic functional units for implementation in hardware or software and for allocation to different processors [1]. But unlike the codesign approach, the research approach does not start with a set of predefined functional units. Instead, we transform a high level definition of a software solution (corresponding to behavior described in the requirements) to design components that can be mapped to implementation modules. An interesting result of the analytical approach is the discovery of design heuristics or generic patterns for localizing change in the solution elements. More about this will be said later.

3.5 Reuse of Solutions Through Automatic System Generation

For those types of applications in which the design space is well defined and directly mappable to a solution, automatic generation of the system implementation is feasible. In general, automatic system generation requires the specification of parameters over which the solution space may vary. Another requirement is a “compiler-like” algorithm for interpreting the values of the parameters and generating the system implementation. One way to do this for software is to determine a suitable software architecture for a class of related applications and to embed or codify the knowledge about how to generate a solution with the particular architecture into a program or tool. The next paragraphs provide some examples of tools for automatically generating hardware and software solutions.

MICON is a collection of tools for the rapid prototyping of small computer systems. M1, the synthesis tool, uses components in a library to build a design that satisfies high level specifications. The input to M1 are functional requirements (e.g. processor name and speed, amount of memory, number of serial ports) and constraints (e.g. maximum board area, power dissipation). The output from M1 is a complete list of parts and a net list indicating the necessary interconnections between the parts. Researchers used M1 to generate designs based on four microprocessor families. An automated knowledge acquisition tool aided the collection of knowledge for the specifications of the designs [25,55].

MOOSE (Model-based Object-Oriented Software generation Environment) is a framework for software development based on formal models and code generation principles. Within MOOSE, a model consisting of a set of component models describes the required behavior for the application. Code generators create code components from the models. Reuse within MOOSE occurs at the modeling level through the creation of new application models from base models which describe common characteristics of the application domain. MOOSE supports component reuse via the code generators as well as from libraries of code components [8].

GenVoca generators synthesize software systems by composing components from reuse libraries. GenVoca components export and import standardized interfaces and are therefore interchangeable and interoperable with other components. The generators include application-independent algorithms for validating compositions of GenVoca components. The interfaces and bodies of GenVoca components are generic descriptions that enlarge upon instantiation. As a result, GenVoca generators enable software systems with customized interfaces to be composed from components with standardized interfaces [21].

The next section discusses in more detail research in the design and composition of software systems from reusable components.

3.6 Design and Composition of Software Systems from Reusable Components

In addition to the codification and reuse of design knowledge, researchers and practitioners have studied the design and composition of software systems from reusable components. There are many areas of interest in the field of reusable software components and component-based development. Current areas with ongoing research include those listed below.

- Specification of software components and interconnections [7,130,138,139]
- Software component composition and architectural concerns [14,38,51,111,112,147,149]
- Commercial Off-The-Shelf (COTS) software robustness testing [92,15]
- Design of software components for use in product-line development [141]
- Design of reconfigurable or upgradable software components [145,129]

Some of the research areas listed above are composed of subareas based on the class of applications to which the research applies. For a more comprehensive review of research involving software reuse in the 1990's and earlier, the author refers the reader to [113], [73], [75], and [77]. The reader might refer to [140] and [28] for extensive lists of web links to software architecture information and resources.

Though ease of change over time is a common goal, there is an important distinction between design for reconfigurability/upgradability and design for maintenance/product evolution. The difference between the two approaches lies in the definition of time. Design for reconfigurability/upgradability usually involves on-line change in the context of the dynamic behavior of the system. Within the run-time environment of the system, a component is defined by its execution state and space (e.g. state behavior within a process space) as well as its logical composition. Design for maintenance typically concerns the static or off-line organization of the components in a system. The designer may design reconfigurable components using tunable parameters, or the designer may define interchangeable components with similar functional behavior but differing levels of performance [137]. Though both design choices require the designer to consider the static organization of the software, the focus is on the execution behavior of the components.

Two research groups at Carnegie Mellon University have studied the design of reconfigurable and evolvable, real-time software components. Chimera components (objects) are reusable software packages that enable the rapid deployment of application processes activated by a real-time kernel [144,145]. The Simplex Architecture is a design approach for building software modules, called cells, that can be safely interchanged during run-time to enable the dynamic upgrade of system control [129,49,142].

The Chimera software infrastructure prescribes a standard format for building reusable software objects. The port-based objects automatically map to executable processes. The format includes operations for initializing, executing, and deactivating the dynamic objects as well as for error handling and inter-object communications [145]. The Chimera infrastructure includes a real-time kernel designed to ensure a performance-efficient and reliable real-time operating environment for robotics and factory applications [146].

In contrast to both Chimera and Simplex, the focus of this dissertation is on the static organization of software and its impact on the software engineer's ability to easily change the software design and resulting implementation. Unlike the Chimera and Simplex emphasis on the rapid deployment of application modules by a real-time kernel, the research approach does not address low-level design mechanisms such as verification of module interfaces and of process state to support dynamic reconfiguration/replacement of processes running in real-time. The research approach applies to any application domain in which high level solution elements such as basic data and operations can be determined from a given set of requirements.

3.7 Empirical Research in Software Engineering

Participants of a workshop sponsored by the National Science Foundation (NSF) define empirical research as an analysis based on the observation of actual practice for the purpose of discovering the unknown or testing a hypothesis. Impediments to empirical research in software engineering often involve differences in attitudes and motivation between software developers and researchers as well as between industry and academe. Cultural differences as well as practical considerations such as the following explain why empirical research in software engineering is seldom done.

- Experimental work is generally complex and expensive to perform.
- Experimental work requires collaboration with practitioners.
- The difference in time scales between research and development inhibits collaboration between researchers and developers.
- Reward structures for development and research personnel are not congruent.
- Students are often not available long enough to make a meaningful contribution to empirical research.
- Many problems to be solved involve human factors such as management and may therefore not be as attractive to researchers.

NSF workshop reports provide guidelines and suggestions for increasing the collaboration between researchers and practitioners and between industry and academe [29]. They also advocate increased funding for validation of research concepts as well as meaningful empirical investigation [17].

Despite the difficulties listed above, there have been empirical studies that have successfully analyzed the effects of particular software practices. These tend to be of three types:

1. Industrial or government case studies that facilitate the collection of data about software practices.
2. Collaborative efforts between industry and academe to test research ideas.
3. Smaller, controlled experiments that are conducted by academic researchers with the help of student subjects.

In the area of software reusability, there have been several case studies documenting the effects of reuse programs or activities instituted by industrial and governmental organizations interested in improving the quality of their software products and processes. As might be expected, organizations willing to invest time in such programs frequently do so because of the need for quality in high-assurance systems that they develop. Leach overviews reuse activities at organizations such as NASA, AT&T, Battelle's Pacific Northwest Laboratory, and Hewlett-Packard. Particularly relevant is his case study of a software reuse program put in place for the Control Center Systems Branch of NASA in Greenbelt, Maryland. This center is responsible for the ground systems that control the initial interface between a spacecraft and ground-based computer control centers. One main feature of the program was the development of a generic core of software that is common to multiple missions as well as a general description of the control center software with a differentiation between generic and mission-specific software. The study also outlines the metrics and methodology used for measuring the quality of the resulting software as well as the extent of software reuse and the cost savings associated with reuse [103].

Collaborative, empirical studies between industry and academe are less common. Two such collaborations are the A-7E Project discussed earlier [30] and a study by Khoshgoftaar and Allen. The latter study involved the use of structural metrics and hysteresis information to predict faults that may occur when the target system is developed and maintained by human subjects. They tested their method with software embedded in a complex, telecommunications system developed by industry. This study was important because it helped to substantiate the assumed relationship between structural features of modules and their resulting quality and because it demonstrated the effectiveness and practicality of a theoretical predictive model [87].

Empirical studies by Hudak, et al., and by Maxion and Olszewski belong to the third category. In the Hudak study, the researchers developed an experimental procedure for testing the effects of different techniques for developing fault-tolerant software. The participants, graduate engineering students, developed fault-tolerant software based on the requirements specification for the Launch Interceptor Program. The subjects in each of four groups applied one of the following software development techniques: *n*-version programming, recovery block, concurrent error-detection, and algorithm-based fault tolerance [69].

In their paper, Maxion and Olszewski describe controlled experiments for testing the hypothesis that robustness for exception failures can be improved through the use of various coverage-enhancing techniques such as N-version programming, group collaboration, and dependability cases. Dependability cases, derived from safety cases, comprise a new methodology based on structured taxonomies and memory aids for helping software designers think about and improve exception handling coverage. They showed that all three methods showed improvements over control conditions in increasing robustness to exception failures and that dependability cases proved the most efficient in terms of balancing cost and effectiveness [108].

The next chapter presents background in the measurement of software design quality.

4 Software Design Measures

As stated repeatedly in previous chapters, the purpose of the thesis is the development of a method for partitioning data, operations, and control flow into software components that simplify product evolution. Chapter 5 details the steps of the method, verifies that the method does what it is intended to do, and explains the mathematical foundations for semi-automation of the method. Chapter 6 discusses the process by which the method was validated to determine its usefulness to software designers. In preparation for these discussions, Chapter 4 briefly reviews concepts about software product quality and techniques for measuring software quality with respect to quality attributes such as maintainability, structural complexity, and reusability. In particular, the chapter outlines ways of measuring software design quality and previews those to be used in validating the research method.

Chapter 4 is divided into the following sections.

- Section 4.1 briefly discusses software product quality and quality attributes.
- Section 4.2 outlines relevant ways to measure structural complexity.
- Section 4.3 overviews the measurement of software reuse.
- Section 4.4 discusses the measurement of changeability.
- Section 4.5 previews the evaluation measures used in validating the research method.

4.1 Software Product Quality and Quality Attributes

As mentioned in 2.3, software engineers frequently define software quality with specific *quality attributes* such as reliability, maintainability, and usability. The target quality attribute for the thesis is **maintainability or evolvability**, with **changeability** and **reusability** as its manifestations. Quality attributes are measurable when associated with observable properties of the software product and the environment in which it is used.

Fenton distinguished between *internal* and *external* product attributes, properties of the software that affect the quality of the software. Internal attributes, such as size, control structure, and modularity, are dependent only on the product itself. In contrast, external attributes depend upon other entities in addition to the product itself. For instance, reliability depends not only upon the software code but also upon the machine environment and the user. External attributes are defined with respect to the perception of the user and are synonymous with particular views of quality or with quality attributes [46].

Often, external quality attributes are too high level to be measured directly. In this case, designers measure lower-level attributes called *quality criteria* that are related to the higher-level concept. For example, the criteria of consistency, accuracy, error-tolerance, and simplicity compose the reliability quality in McCall's model of quality attributes [110]. The importance of internal attributes is the extent to which their measures indicate or predict the external quality of the software. There is an assumption in software engineering that good internal structure implies good external quality. In software engineering, the term *complexity* embodies the totality of all internal attributes. One quality criterion relevant to the thesis is **structural complexity**, because of the commonly shared assumption that it is related to maintainability. The thesis does not address the broad issue of cognitive complexity associated with organization of a software system.

The author refers the reader to [44,78,104] for additional information about software quality and its measurement.

4.2 Measurement of Structural Complexity

Because of the difficulty of measuring design quality, evaluators of software design methods frequently measure the structural complexity of the resulting software to determine the effectiveness of a particular method [46]. As Shepperd notes in an earlier study, design measures encounter problems with the lack of sufficiently formal notation for documenting designs and with validation. Ideally, automatic extraction of design measurements from the artifacts should be possible; but the lack of formal design languages makes it impossible for a machine to translate and evaluate a design. Therefore, a common approach is to infer design or structural properties from the resultant code; but this precludes the evaluation of complexity during the design phase when alterations of structure are most easily made [136].

Another general problem is the validation of software measures. Pfleeger, et al., defines a measure as a mapping from the real, empirical world to a mathematical world in which one can more easily understand an entity's attributes and relationship to other entities. The problem is that validation is necessary to determine if a particular measure is really measuring what it claims to measure: the measure is valid if it captures in the mathematical world the behavior perceived in the empirical world. Frameworks for validating software measures based on measurement theory and statistical rules are relatively new [119]. The thesis uses or builds upon accepted design measures.

Henderson-Sellers developed a taxonomy of widely accepted structural complexity measures. In the Henderson-Sellers model, measures of internal attributes contribute in whole to the measurement of the structural complexity of a software product. Intermodule and intramodule measures compose the measurement of structural complexity [59]. Many of the example intermodule and intramodule measures presented in the following paragraphs relate to object-oriented systems. Since the subjects for the research experiments were learning or applying an object-oriented programming language in the classes from which they were recruited, the experimenter chose evaluation measures applicable to object-oriented software systems. Likewise, the experimenter adapted the research method to partition the required logic into modular structures used in object-oriented software development.

Intermodule measures involve the coupling or interaction between modules. Examples of intermodule measures for object-oriented systems are those which follow.

- For each class *c* of a software system:
  - Total number of activations of methods which are defined in classes other than *c* by methods defined within *c* (coupling between classes at the method level)
  - The number of classes (other than *c*) which contain a method activated by a method contained in *c* (coupling between classes at the class level)
- For each method *m* of a class *c*:
  - Total number of activations of methods which are defined in *c* (coupling between methods within a class)
  - Total number of activations of methods which are not defined in *c* (coupling between methods of different classes)

Traditional intramodule measures involve features such as size (e.g. lines of code, Halstead's software science [57], and function point counting [2]) and control flow (e.g. McCabe's cyclomatic complexity [109]). Examples of related size measures for object-oriented systems are those which follow [60].

- Method size (including the mean and the standard deviation across all methods)
- Number of methods per class (including the mean and the standard deviation across all classes)
- Number of data stores per class (including the mean and the standard deviation across all classes)
- Class size (including the mean and the standard deviation across all classes)
- Total number of classes in the system

Class size is dependent not only upon the number and size of the methods encapsulated by a class but also upon the number of attributes or "data stores" in a class. In contrast to variables defined within methods, class attributes are variables defined within a class but outside of the methods declared in the same class. Li and Henry define class size S_C as shown in Figure 4.1.

$$\text{class size} = S_C = NOA + NOM$$

where *NOA* is the number of attributes per class and *NOM* is the number of methods per class.

Figure 4.1 Mathematical expression for Li and Henry's definition of class size [106].

Henderson-Sellers present an extension of class size that includes programming language-dependent weights W_A and W_M (mean values) for the sizes of attributes and methods, respectively [61]. This extended definition of class size, s_i, is shown in Figure 4.2. The related size of a system S for N classes, each of size s_i, is as shown in Figure 4.3.

$$\text{class size} = s_i = (AW_A + MW_M)_i$$

where A is the number of attributes or data stores defined in class i, M is the number of methods defined in class i, and W_A and W_M are mean values for the size of the attributes and methods, respectively.

Figure 4.2 Mathematical expression for Henderson-Sellers' extended definition of class size [61].

$$\text{system size} = S = \sum_{i=1}^{N} s_i = \sum_{i=1}^{N} (AW_A + MW_M)_i$$

where N is the number classes in the system, s_i is the size of class i, A is the number of attributes or data stores defined in class i, M is the number of methods defined in class i, and W_A and W_M are mean values for the size of the attributes and methods, respectively.

Figure 4.3 Mathematical expression for Henderson-Sellers' definition of system size [58].

Several researchers have organized suites of metrics for measuring structural features of object-oriented (OO) systems. Six product design measures proposed by Chidamber and Kemerer focus on size, coupling, and cohesion [37]. Basili, et al., conducted an experiment to empirically validate the OO metric suite proposed by Chidamber and Kemerer with regard to the ability of each measure to predict fault-prone classes. Except for the number of children of a class (*NOC*), they found a positive correlation between all of the measures and the probability of fault detection [16]. Readers should refer to [62] for a comprehensive review of OO metric suites. Typically these suites include metrics that can be evaluated at design time as well as those which are most easily extracted from code (e.g. the number of methods per class versus the size of a method).

Many suites also distinguish between product and process metrics (e.g. the number of classes in the system versus the number of classes reused when modifying a system). In Section 4.5, we present an extension that makes the weighting of attribute and method sizes less dependent on programming language features and more dependent on functional features that can be expressed in pseudo-code at design time.

The author refers the reader to [115] for the application of software metrics and to [127] for a study in the validation of metrics.

4.3 Measurement of Software Reuse

As stated in Section 2.4, software reuse involves the use of one or more software artifacts in multiple software solutions. In earlier work leading to the development of the research approach, the focus was on the development of reusable software components that are executable or compilable modules carefully designed to be useful in several programs, including unanticipated ones [65]. As it became more apparent that partitioning logic to achieve evolvable software solutions is primarily a software design activity, the focus centered on architectural or design components [66,67,68]. Similarly, many software developers think the reuse of software code embodies the whole of software reuse. The research community investigating the usage of software patterns and styles have shown that the potential for reducing effort and error by reusing existing software designs is substantial (discussed in Section 3.1). Hence, there is certainly a need for ways to measure a design's potential for reuse.

Measures of reuse tend to be "after the fact." In other words, a software artifact's reusability is a measure of the degree to which it is actually reused across multiple software solutions. Typically such measurements involve the reuse of software components implemented as code, but it is feasible to measure the reuse of design components. For instance, components such as class and method definitions are apparent at both the design and code levels. During the process of creating a new software design, the designer can classify the components into the following three categories.

- Reused without modification
- Reused with modification
- New

The reader can refer to [59] for a list of reuse measures relevant to object-oriented systems.

To determine a design's potential for reuse "before the fact" requires the designer to think either about the future evolution of the product or about feasible related applications. The key is to pose potential new requirements and to determine how much of the current design could be reused to create a new design to satisfy these requirements. Without actually designing new components, the designer can determine which components from the current design can be reused with or without modification in the new design and which components are not reusable.

Sizing design components is useful in estimating the following measures:

- Proportion of an existing design which is reusable.
- Proportion of a new design constructed from reused components.

One way to estimate the size of a design component is to relate it to a representative component of code. The estimator then applies a technique for sizing code such as counting lines of code or counting according to some other specific guidelines. Example guidelines for estimating the size of program elements and files appear in Appendix P.

Another way to estimate a design component's size is to develop a method for sizing components and data stores specified in a design. To compensate for differing levels of detail in the pseudo-code provided by different designers, the estimator can create a benchmark design and map the logic in target designs to the benchmark logic which has been sized. This is the approach used by the experimenter in evaluating the designs created by the subjects in the first experiment of the research studies. Discussion of the research studies appears in Chapter 6 and Chapter 7.

4.4 Measurement of Software Changeability

Measuring ease of change or changeability requires a rationale for relating observable properties of the software artifact to be changed with the human effort required to change it. *Locality of change* (a type of cohesion with respect to change) and *size of change* are observable properties that directly relate to the effort required to change software. Clearly, a software maintainer can more easily modify a system whose pieces of logic or data to be changed are located closely together, such as in the same component. With this rationale, one might argue that a monolithic system is the most easily changed. On the other hand, the maintainer can more easily understand and modify smaller components and is less likely to introduce error when working on a smaller part of a software system. The logical approach is to locate those parts that are expected to change together in the same component and separate from those parts not related with respect to the anticipated change. If the component becomes “too big,” the designer can create a hierarchical decomposition of the component into subcomponents located closely to one another through the hierarchical organization.

Software developers use size as an estimator for development or maintenance effort and for potential error in maintenance. One way to measure the size of change impact is to sum the sizes of the components that must be modified to satisfy the change. The developer can use the size of change impact to estimate the requisite maintenance effort. The size of change impact is then the sum of the size of each component involved in implementing the change. The overall goal is to reduce the size of the change impact. The mathematical expression for the size of change impact is shown in Figure 4.4.

$$\text{size of change impact} = S_c = \sum_{i=1}^{N} S_i$$

where S_c is the size of the impact of change c, N is the number components which must be modified to implement c, and S_i is the size of each component to be modified.

Figure 4.4 Mathematical expression for the size of a change impact.

The partitions in Figure 4.5 show that separating the logic not affected by the same changes helps to reduce the size of the change impact. Suppose that logic A, B, and C are affected by a required change x and that logic D, E, and F are **not** affected by change x. With the first partition, the size of the change impact is the size of component C_1 which is the sum of the sizes of all the logic pieces. The second partition separates the logic of D, E, and F from the impact of change x by putting D, E, and F in a separate component. So with the second partition, the size of the change impact is less than that for the first partition because the size of component C_1, which contains only the affected logic, is smaller.

Given: Logic *A*, *B*, and *C* are affected by change *x*. Logic *D*, *E*, and *F* are not affected by *x*.

Partition 1:

C_1: $\boxed{A, B, C, D, E, F}$

Size of change impact = size(C_1).
= size(A)+size(B)+size(C)+size(D)+size(E)+size(F).

Partition 2:

C_1: $\boxed{A, B, C}$ C_2: $\boxed{D, E, F}$

Size of change impact = size(C_1).
= size(A)+size(B)+size(C).

Figure 4.5 Partitioning to separate logic not affected by the same changes.

In addition to separating logic that is unrelated with respect to change, locating together logic affected by the same changes helps to reduce the effort of finding the relevant parts of the software. The previously presented estimate for the size of change impact, summing the sizes of the affected components, does not reflect any effort needed to find individual components related to a particular change. Therefore, as shown by the two partitions in Figure 4.6, the size of the change impact does not necessarily increase with an increase in the number of affected components.

Given: Logic *A*, *B*, *C*, *D*, *E*, and *F* are affected by change *x*.

Partition 1:

C_1: $\boxed{A, B}$ C_2: $\boxed{C}$ C_3: $\boxed{D}$ C_4: $\boxed{E, F}$

Size of change impact = size(C_1)+size(C_2)+size(C_3)+size(C_4)
= size(A)+size(B)+size(C)+size(D)+size(E)+size(F).

Partition 2:

C_1: $\boxed{A, B, C, D, E, F}$

Size of change impact = size(C_1).
= size(A)+size(B)+size(C)+size(D)+size(E)+size(F).

Figure 4.6 Partitioning to locate together logic affected by the same changes.

To account for any effort needed to find components affected by a change as well as the difficulty of coordinating modifications across components, one can add a term *wN* to the mathematical expression for the size of change impact. *N* is the number of components that must be modified, and *w* is a factor to weight the difficulty of modifying multiple components that are related to the same change. With this alternative expression for the size of change impact shown in Figure 4.7, putting logic affected by the same change into the same component will help to reduce the change impact value. The term localizing change includes both separating logic not affected by the same changes and locating together logic affected by the same changes.

$$\text{alternative size of change} = wN + \sum_{i=1}^{N} S_i$$

where N is the number components that must be modified to implement a change c, w is a factor to weight the difficulty of modifying multiple components related to the same change, and S_i is the size of each component to be modified.

Figure 4.7 Mathematical expression for the alternative size of a change impact.

One can also measure the size of the change with respect to the size of the whole system as a proportion. If the value of this proportion is large, then either the system is poorly partitioned with respect to the anticipated change or the impact of the change is large. A change with a large impact may justify the creation of new components and the replacement rather than the modification of components.

The reader has probably noticed that there is a relationship between reusability and changeability. A system with highly reusable components (especially those that do not require modification to implement similar applications) is a system with a low impact or size of change. In other words, there is an indirect correlation between the degree of reusability and the impact of change, a concept which is discussed in more detail in Section 6.2.

4.5 Evaluation Measures Used in Validating the Research Method

Validating the effectiveness of the research approach requires the comparison of software designs created by designers who apply the research approach with software designs created by designers who do not use or apply the research approach. The validation includes a measurement of evolvability as well as an assessment of structural complexity. Measurement of evolvability involves dual measures:

1. A measure of the impact of changing the designs with respect to target changes in requirements.
2. A measure of the degree of reusability with respect to target changes in requirements.

The author refers the reader to Chapter 6 for an extension discussion of the validation process and the relevant evaluation measures.

This section ends with a few notes about how some of the design measures discussed earlier are applied specifically in the validation of the research approach. The measurement of method and class size applied in this dissertation is an extension of the Henderson-Sellers size measures presented in Section 4.2. The experimenter assigned a separate weight to each attribute and method contained within a class instead of using average weights. Since the evaluation pertains to software designs rather than code, the concept of weight refers to an estimated size of the functionality to be implemented by an attribute or a method. The experimenter generated a representative or standard design that included the basic data and operations identified by applying the research approach. The experimenter then estimated the relative size of each solution element using detailed pseudo-code. For each attribute or method specified in a design produced by a subject, the experimenter either estimated its size directly from the designer's pseudo-code or compared it to a corresponding solution element in the standard design if there was insufficient pseudo-code to estimate its size. Using individual size estimates for attributes and methods, the size of a system *S* containing *N* classes is therefore represented by the mathematical expression shown in Figure 4.8.

$$\text{system size} = S_{ext} = \sum_{i=1}^{N}\left(\sum_{j=1}^{A_i} w_{ij} + \sum_{k=1}^{M_i} w_{ik}\right)$$

where N is the number of classes in the system; A_i, M_i are the numbers of attributes and methods for class i, respectively; and w_{ij}, w_{ik} are the estimated sizes or weights of attribute j and method k for class i, respectively.

Figure 4.8 Mathematical expression for the size of a system.

Control flow in object-oriented systems is typically a measure of the number of decision points within a method, e.g. an application of McCabe's complexity metric $V(G_i)$ to the flow graph G_i of each method i. $V(G_i)$ is a code-level metric and therefore difficult to measure in the specification of a software design. Therefore, the experimenter used a pseudo-implementation of the standard design discussed previously to estimate the number of decision points per basic solution element. If the pseudo-code in a design produced by one of the subjects lacked sufficient detail to determine the number of decision points, then the experimenter compared it to the corresponding solution element in the standard design and estimated the number of decision points for the element.

5 Analytical Partition of Components

As discussed in Chapter 1 and Chapter 2, a primary architectural decision involves determining the components of the system. A logical component is an encapsulation or abstraction of part of the solution and has a representative name, meaning, and interface. A physical component is the implementation of a logical component with an observable container such as a directory or file in a file system. Programming constructs such as procedures, functions, modules, and classes also serve as physical encapsulations of logic. Hierarchical composition of components enables the designer to focus on different levels of abstraction from system and subsystem behavior to the low level details of a particular algorithm.

Determining the components of an architecture consists of the following four sub-problems.

1. How to determine solution elements to satisfy the required behaviors.
2. How and why to partition (decompose) higher-level solution elements into lower-level solution elements.
3. How to partition (group) the solution elements into logical components that perform the required behaviors,
4. How to encapsulate the logical components into physical components.

The partitioning techniques discussed in this chapter primarily address the first through the third subproblems, though examples are given for subproblem four. The goal is a systematic process for partitioning the solution into components that will not only perform the correct behavior but will also simplify the process of modifying the solution to satisfy changing requirements.

This chapter is organized into the following sections.

- Section 5.1 motivates this chapter with a discussion of the rationale for an analytical partitioning process.
- Section 5.2 addresses the first sub-problem of how to determine basic design elements.
- Section 5.3 presents a systematic process for partitioning data and operations into components.
- Section 5.4 presents the mathematical foundation for the partitioning process presented in Section 5.3 and demonstrates how this foundation enables the semi-automation of the process.
- Section 5.5 describes an optimal process for partitioning control flow into components.
- Section 5.6 outlines and proves a heuristic for determining a good, though not necessarily optimal, control flow partition in polynomial time.
- Section 5.7 demonstrates how the design techniques presented in this chapter can be integrated with other existing design approaches and discusses the merit of integration.

5.1 Rationale for an Analytical Partitioning Process

Because of the safety implications and the high cost of maintaining systems with stringent reliability requirements, two fundamental goals for this research are those shown in Figure 5.1.

1. The improved understanding of *software metamorphosis* (the nature of changes to software).
2. The development of systematic techniques for designing software solutions that can be reliably and easily changed.

Figure 5.1 Fundamental goals for the thesis research.

The terminology “reduce the impact of change” means that a software solution can be modified in a reliable, timely, and cost-efficient manner. Ease of change is crucial not only during the development of new applications but also during the evolution of a product line of applications. During the development of a new application, software engineers would like to be able to experiment with alternative software solutions to achieve the desired performance or reliability. Sometimes, developers can achieve these goals via tunable software parameters. Other times, they need to tune the solution by replacing poor performing parts of the system [137]. Reusable software components that can be used alternatively to accommodate changing requirements help to reduce the impact of change.

Two fundamental rationales direct the partitioning approach, presented in this chapter, for determining reusable software components (from here on referred to simply as the *research approach*). One rationale originates from the idea of recasting large-effect algorithms into smaller-effect or partial algorithms [155].

A more general interpretation of this rationale is that software solutions implemented as large-effect components may not be as reusable as those implemented as smaller-effect components. The reader should note that decomposition of a software solution into parts is an essential part of determining an architecture for a software system. This applies not only to the low level details of algorithms but also to the high level structure of the system and subsystem. Figure 5.2 states the relevant research question.

One research question is how to recast a monolithic software solution into parts whose implementations will facilitate changes to the software solution via replaceable and reusable components.

Figure 5.2 Research question regarding the recast of monolithic software solutions.

The other rationale behind the research approach is the idea that those parts of a software solution that are affected by the same expected changes should be located in the same software component. Depending upon the type of change and the organization of the software, the software engineer may need to examine and modify multiple parts of the software system. Minimizing the number of affected components helps to simplify the process of finding the relevant parts of the software. Likewise, locating parts that change together in the same components helps to reduce the opportunity for introducing errors into the parts that are not related to the change. The relevant research question appears in Figure 5.3.

A second research question is how to design software components that localize change.

Figure 5.3 Research question regarding localization of change.

The term analytic has the following three definitions that relate to the research approach [154].

1. Separating into elemental parts or basic principles.
2. Reasoning from a perception of the parts and interrelations of a subject.
3. Following necessarily as in a logical proposition.

The research approach helps the designer to determine the basic design elements of the software solution and to partition these elements based on their relationships with respect to change. A mathematical representation for the relationship between the basic elements enables a systematic partition into components and semi-automation of the research approach. The next section discusses the types of design elements that are the focus of this research.

5.2 Determining Basic Design Elements

The designer starts with a specification of the required behavior that the software solution should perform along with a detailed description of the conditions and constraints for the behavior. The designer first identifies the basic behavioral elements in the requirements analysis. With a structured design approach, the behavior includes any of the following basic elements.

- Data stores and processes that transform the data (represented by data flow diagrams).
- Control flow between the data processing elements (represented by control flow diagrams).
- Events and conditions that change the context for appropriate behavior (represented by state transition diagrams).

As discussed in Chapter 2, with object-oriented design the nouns and verbs in the requirements specification become object entities with associated data and actions. Eventually the control flow between objects becomes apparent via scenarios of interaction. Regardless of the design approach, software solutions include basic data, operations (many of which transform data), and the flow of control between the operations.

Data, operations, and control flow are basic solution elements that become encapsulated in components that define the architecture of the software solution. The human designer may start by mapping the data, operations, and control flow, described in the behavioral analysis, directly to representative solution elements. The designer may choose different names for these solutions elements, but there should be a reverse mapping from solution elements back to the basic behavioral elements. Ways in which the designer maps behavioral elements to solution elements may vary as listed below.

- Prior knowledge (mapping based on a known solution to a similar problem).
- Brute force (direct mapping of behavioral elements to solution elements).
- Brainstorming (exploration of different mappings with the expectation of finding a suitable mapping based on some design criteria).

If the designer knows of a software solution that applies to a particular problem (in this case solution elements for which there seems to be a mapping from the behavioral elements), then he/she may be likely to use this solution without exploring alternatives.

Known solutions may not, of course, be the best solutions. Therefore, it is important to help the designer brainstorm about alternative solutions or to have a machine automatically do the brainstorming for the human. The research approach guides the human designer to first identify the basic data, operations, and control flow elements that are observable from the behavioral analysis. The next step in the design process is to refine the basic solution elements through decomposition and rearrangement or grouping. The objective is the creation of a design space in terms of basic solution elements and the determination of a good partition of these elements into components based on design constraints. As a first step toward this objective, the research approach focuses on the relationship between solution elements with respect to software evolution. The research approach directs the designer in the partition of the solution elements into components that will maximize their reuse and minimize the complexity of changing the solution to satisfy new requirements.

5.3 Approach for Partitioning Data and Operations

Basic elements of a software solution include data and operations. Following a structured approach, the designer identifies the primary data elements (denoted as data stores) and processes that transform the data. With an object-oriented approach, the designer identifies objects, some or all of which encapsulate key data for the software solution. Encapsulating data and operations, or information hiding, supports the localization of solution elements that change together. The problem is that the designer is solely responsible for determining the composition of the basic system components or objects. Prevalently used design methods do not guarantee that the designer will consider the appropriate level of reuse or group together those elements which change together.

The research approach has two primary features:

1. A manual but guided reuse and change analytic process.
2. A mathematical model and automatable algorithm for localizing solution features that change together.

Figure 5.4 enumerates the six basic process steps of the partitioning process. The following text illustrates the application of the process to the specification of components for the microwave oven software.

1. Identify the basic data and operational features of the software solution.
2. Recursively decompose the large-effect operations and identify additional data elements.
3. Enumerate the expected or feasible changes to the software solution.
4. Determine the change set of data and operations for each expected or feasible change.
5. Combine and organize into components the overlapping change sets.
6. Add other necessary components and organize remaining data.

Figure 5.4 Process for partitioning data and operations into reusable components.

Step 1: Identify the basic data and operations.

The first step is to identify the basic data and operations in the requirements or behavioral analysis. The reader should refer to the statement of work for the microwave oven software in Table B.1 of Appendix B.

The four basic functions to be performed by the microwave software are:

1. Manage the user interface.
2. Schedule a heating operation.
3. Control the electronics.
4. Drive the electronics.

The statement of work describes the data used by the microwave oven software in the sections which discuss the basic functions. By understanding these functions, the designer can determine the data that is used by the microwave oven software. Therefore, the initial set of data and operations consists only of the four functions listed above.

Step 2: Recursively decompose the large-effect operations and identify additional data elements.

The second step involves the decomposition of the large-effect operations into smaller-effect operations. The process repeats recursively until further decomposition results in operations which are not reusable or which do not help to make the solution easier to understand and design. In this example, the designer can refer directly to the statement of work to determine the sub-operations of the four basic operations defined in the first step. Depending on the level of detail in the requirements analysis, the designer may need to identify sub-operations through analysis.

The question is whether or not the basic operations should be decomposed for the purposes of partitioning. Decomposition allows the sub-operations to be reused separately and to potentially be located in separate components. The designer can think of each operation or sub-operation as a "block of logic" to be used in the construction of multiple software solutions. Reusable means that the designer can use the operation in a similar application or in a different application that would require a similar type of operation. Reuse without modification is most desirable, but reuse with minimal modification can also be useful. The designer can also consider whether or not the decomposition yields one or more sub-operations for which alternative implementations are needed (e.g., to enable multiple versions of a software solution, each with a different level of quality of service or performance). Ease of replacing individual sub-operations enables the other parts of the solution to be reused without modification. Design for reuse and ease of change are integrally related.

The sub-operations of the Manage-User-Interface operation are:

- Get input from the keyboard.
- Put text onto the display (e.g., text to describe buttons which have been pushed, error messages, and status messages).
- Stop an active heat operation if the STOP button is pushed.
- Check the sequence of keyboard inputs for each of the following heating types.
    - straight heat
    - defrost
    - reheat
- Schedule a heat operation.

From a design point of view, decomposition of the Manage-User-Interface operation is useful if one or more of the sub-operations is reusable as a separate operation. The reusability of some sub-operations, such as the keyboard input and display output, is more obvious than for others. The statement of work instructs the designer to use the predefined methods called GET_KEYBOARD and PUT_DISPLAY whose implementation will reside in a system.h file. Each heating type has its own expected sequence of button pushes and related types of keyboard input. The button sequence for a specific heating type may vary for different microwave ovens, while for other heating types it may be the same. Therefore, the creation of a separate sub-operation for each heating type would simplify the reuse, alteration, and addition of button sequences for different microwave ovens.

On the other hand, the logic to stop or to schedule a heat operation involves the preparation of a call to another part of the software and is therefore not reusable outside the context of the user interface. The Manage-User-Interface operation yields reusable sub-operations to handle the button sequences for the different heat types as well as the predefined methods to handle keyboard input and display output. The Manage-User-Interface operation is itself useful for activating the sub-operations and for “gluing together” parameters necessary to request a heat operation from the Schedule-Heating-Operation part of the software. It is not only reusable but also replaceable for the evolution of more sophisticated user interfaces.

The above analysis goes beyond that used in a previous study [65]. Previously, the decomposition of an operation T into a set of sub-operations would occur if and only if all of the sub-operations of T are reusable. Now, the decomposition of T occurs if and only if at least one sub-operation t is reusable. Another change is the addition of T to the set of reusable operations if at least one sub-operation t of T is not reusable or if T is reusable as an activator of its sub-operations. The result is the inclusion of larger-effect operations with definite potential for reuse along with the smaller-effect operations in the resulting set of operations.

In addition to the primary operations for managing the user interface, the designer also needs to consider the data input by the user (e.g., power level, duration, and weight).

- Should the data be decomposed to enable parts of it to be reused or modified separately?
- What are the necessary operations on the data to satisfy the required behavior?
- Should these operations be decomposed?
- Are there any data needed that is not apparent in the behavior analysis?

The data to manage the user interface is not complex: from a programming point of view, it is representable by scalar data structures. Likewise, the reading and writing operations on the data are built into the programming language and are therefore not significantly reusable from a design point of view. Likewise, range checks on the data can be done with a few programming statements; and the use of constants enable the allowable limits of the data to be easily changed through compilation. It does not appear to be necessary to design special operations on the data identified in the statement of work.

The statement of work also directs that the microwave oven should report status and error messages and should cancel all user programming if any error, stop, or door open condition occurs. The analysis above addressed the stop operation. Reporting error and status messages involves output to the PUT_DISPLAY operation discussed previously. While interpreting the sequence of button pushes for a particular heating operation, the software must cancel any input data and prevent further user programming of the microwave oven in the event of error or the door being opened. This logic has no clear usefulness outside this context. There appear to be no additional sub-operations that should be separated from the logic previously discussed. On the other hand, data items will be needed for the door status as well as for error and status messages. The implementation of these data are programming language specific and do not require special operations to manage them. The results of analyzing the Manage-User-Interface functionality appears in Figure 5.5.

Operations	Data	
manage_user_interface	power level	heating type
straight_heat	duration	door/door status
defrost	weight	error message
reheat	number of servings	status message
GET_KEYBOARD	weight per serving	
PUT_DISPLAY		

Figure 5.5 Analysis of the Manage-User-Interface functionality.

The analysis of the Schedule-Heating-Operation functionality is done in the same way with decomposition based on design for reusability and ease of modification. The purpose of this part of the software is to determine a power level and duration needed for a heat operation and to pass this information to the software that controls the electronics. What must be done depends on the type of heat request. Straight heating involves the conversion of the user-input duration expressed in minutes to seconds. A separate conversion operation would simplify change to the correlation between user and machine duration units. The statement of work also specifies that formulas (special operations) called DEFROST and REHEAT will handle the conversion between weight or servings and weight per serving to a power level and duration. No additional steps are needed for a stop operation.

The question is whether a separate Schedule-Heating operation is needed to encapsulate the logic to activate the conversion sub-operations discussed in the previous paragraph. All of the sub-operations are reusable or candidates for change. The Schedule-Heating operation has no other function, and the sub-operation for each heat type in the Manage-User-Interface part of the software could activate the appropriate conversion directly. There appears to be no use for a separate Schedule-Heating operation. The data used by the conversion sub-operations comes directly from the heat type sub-operations. Other necessary data items are a duration in machine units as well as error and status codes returned from the Control Electronics. The operations and data resulting from the two previous analyses appear in Figure 5.6.

Operations		**Data**	
From Previous Analysis	*New*	*From Previous Analysis*	*New*
manage_user_interface	DEFROST	power level	duration2 (ticks)
straight_heat	REHEAT	duration1 (minutes/seconds)	error code (from Control Elec.)
defrost	convert_to_ticks	weight	status code (from Control Elec.)
reheat		number of servings	
GET_KEYBOARD		weight per serving	
PUT_DISPLAY		heating type	
		door/door status	
		error message	
		status message	

Figure 5.6 Resulting operations and data after analysis of the Schedule-Heating functionality.

The Control-Electronics part of the software directs the hardware to perform a heat or stop operation, monitors the progress of a heat operation via a feedback loop, and returns error and status codes. The two basic sub-operations are the heat and stop functions. According to the statement of work, these functions “know” how the hardware devices should be coordinated to accomplish the required functionality but are not aware of the hardware interface details. Therefore, the heat and stop sub-operations consist primarily of requests to the Drive Electronics to perform specific functions such as increasing or decreasing by one notch the power on a particular power source. These requests are not reusable outside the context of a heat sub-operation.

On the other hand, the Control Electronics does include logic, such as stopping the power on all power sources, to make requests across multiple hardware devices. In addition, the requirement to stop the power on all power sources appears three times in the statement of work. This block of logic, from here on called stop_all_power_sources, is reusable within the current solution. Sub-operations for increasing or decreasing power on all power sources relate to the way in which power is controlled (currently 3 notches for each adjustment). Initiating the power source status on all power sources is also a reusable block of logic. Lastly, reading the status of all electronics may also need to be replaced or modified if the type of hardware devices is different for future microwave ovens. Encapsulation of this logic in a separate sub-operation would simplify replacement or modification

The designer needs to determine if the heat and stop sub-operations are reusable blocks of logic or are candidates for replacement or modification. The heat sub-operation contains an ordered sequence of task activations (some to activate the Drive Electronics directly and others to activate intermediate logic to handle multiple devices). The feedback loop is a reusable sequence but is also a candidate for replacement or modification to satisfy changing requirements. The reader should see Section 5.5 and Section 5.6 for an approach to designing control flow components, such as the heat sub-operation, for change. The stop sub-operation activates the stop_all_power_sources sub-operation and is therefore not a reusable block of logic.

Operations			**Data**
From Previous Analysis	*New*	*From Previous Analysis*	*New*
manage_user_interface	control_electronics	power level (user interface)	desired power level (to Control Elec.)
straight_heat	heat_operation	duration1 (minutes/seconds)	duration (to Control Elec.)
defrost	stop_all_power_sources	weight	actual power level (from Driver Elec.)
reheat	increase_all_power_sources	number of servings	power source status (sources 1-4)
GET_KEYBOARD	decrease_all_power_sources	weight per serving	power sensor status (from Dr. Elec.)
PUT_DISPLAY	initiate_status_all_power_sources	heating type	door status (from Drive Elec.)
convert_to_ticks	read_all_devices_status	door/door status (user interface)	door sensor status (from Drive Elec.)
DEFROST		error message	time (from Drive Elec.)
REHEAT		status message	timer status (from Drive Elec.)
		duration2 (ticks)	error code (from Drive Elec.)
		error code (from Control Elec.)	status code (from Drive Elec.)
		status code (from Control Elec.)	error code (to user interface)
			status code (to user interface)
			heat request (from user interface)
			power sources (4)
			power sensor
			door sensor
			timer

Figure 5.7 Resulting operations and data after analysis of the Control-Electronics functionality.

To conclude the analysis of the Control Electronics, the designer must consider the usefulness of a control_electronics operation and the data used by the Control Electronics software. The control_electronics operation contains a reusable heat sub-operation and logic to activate the stop_all_power_sources sub-operation when the user requests a stop operation. The control_electronics operation hides the details of the sub-operations used to perform a heat or stop operation. For testing purposes, it is helpful to have an intermediate component that can analyze detailed status and error codes returned by the software closer to the hardware. The control_electronics operation converts the detailed status and error codes returned by the heat and stop_all_power_sources into the more generic codes required for the manage_user_interface software. The Control Electronics data items include status and error codes as well as the status of and a representation for each hardware device being monitored. The list of operations and data after the analysis of the Control Electronics appears in Figure 5.7.

Operations		**Data**	
Previous	*New*	*Previous*	*New*
manage_user_interface	increase_power	power level (user interface)	power level (from power sensor)
straight_heat	decrease_power	duration1 (minutes/seconds)	power source status (from pwr. src.)
defrost	shut_off_power_source	weight	power sensor status (from pwr. sen.)
reheat	initiate_power_source_status	number of servings	door sensor status (from door sen.)
GET_KEYBOARD	read_power_source_status	weight per serving	door status (from door sensor)
PUT_DISPLAY	initiate_power_sensor_read	heating type	time (from timer)
convert_to_ticks	read_power_sensor_level	door/door stat. (user interface)	timer status (from timer)
DEFROST	read_power_sensor_status	error message	
REHEAT	initiate_door_status	status message	
control_electronics	read_door_status	duration2 (ticks)	
heat_operation	read_door_sensor_status	error code (from Control Elec.)	
stop_all_power_sources	set_timer	status code (from Control Elec.)	
increase_all_power_sources	check_timer_status	desired power level (to Control Elec.)	
decrease_all_power_sources		duration (to Control Electronics)	
initiate_status_all_power_sources		current power level (from Driver Elec.)	
read_all_devices_status		power source status (sources 1-4)	
		power sensor status	
		door status (from Drive Electronics)	
		door sensor status (from Drive Elec.)	
		time (from Drive Electronics)	
		timer status (from Drive Electronics)	
		error code (from Drive Electronics)	
		status code (from Drive Electronics	
		error code (to user interface)	
		status code (to user interface)	
		heat request (from user interface)	
		power sources (4)	
		power sensor	
		door sensor	
		timer	

Figure 5.8 Resulting operations and data after analysis of the Drive-Electronics functionality.

The Drive Electronics reformats Control-Electronics requests to match the hardware-specific interfaces. Isolating each of these requests in a separate sub-operation simplifies changes to the hardware interface. The statement of work specifies that the hardware interface includes a configurable CALL_HARDWARE method. The list of operations and data after the analysis of the Drive Electronics appears in Figure 5.8. The reader may note that some of the data appear to be duplicates of the same information (e.g., error/status codes from the Control Electronics and error/status codes to the user interface). Such data may store the same values during run-time; but from a design point of view, they represent information being processed or passed by different parts of the software.

The next section discusses types of changes that are feasible or likely to be made to the solution.

Step 3: Enumerate the expected or feasible changes to the software solution.

To determine the expected or feasible changes to the software solution, the designer can start with an analysis of the requirements. A domain analyst can help to identify ways in which the requirements may change over time. In Table B.2 of Appendix B, the statement of work for the microwave oven contains an analysis of the way in which the microwave oven product is expected to evolve. Table 5.1 presents an abbreviated version of the expected evolution of the microwave oven. Associated with each change is a *change signature.*

The reader may note that the items 10-12 in Table 5.1 are an expansion of the ninth item in Table B.2. It is important that similar changes be grouped together only if they are expected to happen together. If there is a possibility that one of a group of similar changes may occur separately, then that change should appear separately in the list. If the members of a group of similar changes may occur separately but are also likely to occur together, the designer should list them separately but mark them with an identifier as shown by the superscript *RCG1* after items 9-12 in Table 5.1. *RCG1* is an abbreviation for the term *related change group number 1*. Item 2 in Table B.2 yields two changes shown as changes 2 and 3 in Table 5.1.

Table 5.1 Change signatures for the expected or feasible changes to the microwave oven software.

Expected or Feasible Evolution of the Microwave Oven Software *** denotes changes suggested by the designer**	**Change Signature**
POWER-TIMER as well as TIMER-POWER button sequences to program straight heating.	HBSQ
Different defrost formula.	DFORM
Different reheat formula.	RFORM
More sophisticated control of power sources than up or down three notches.	CPSRC
Different feedback loops to allow for electronics which respond faster.	FDBL
Error and status messages, as well as weight measures, for the international market.	IMSWT
More powerful microwaves with higher limits for weight, servings, and weight per serving.	HLWS
Different configurations of hardware devices.	CHD

Expected or Feasible Evolution of the Microwave Oven Software * denotes changes suggested by the designer	Change Signature
Different type of power source with new parameters for the CALL_HARDWARE interface.RCG1	PSRC
Different type of power sensor with new parameters for the CALL_HARDWARE interface.RCG1	PSNSR
Different type of door sensor with new parameters for the CALL_HARDWARE interface.RCG1	DSNSR
Different type of timer with new parameters for the CALL_HARDWARE interface.RCG1	TIMER
Addition of programmed operations to heat specific foods such as bacon, popcorn, or vegetables.*	APO
Event-driven approach to controlling the hardware.*	EDA

The designer can suggest changes to requirements that are feasible though not listed in the anticipated evolution of the product. For instance, the designer may foresee the addition of other heating types or the replacement of one or more of the straight heating, defrost, or reheat types. Likewise, the designer may have ideas about how the solution could change to improve its performance. For instance, the designer may envision an event-driven approach to controlling the hardware rather than polling within a feedback loop. The items marked with an asterisk in Table 5.1 are those added by the designer.

The next step is to determine the impact that evolution of the microwave software would have on the solution elements that have already been identified.

Step 4: Determine the change set of data and operations for each expected or feasible change.

The objective is to determine the solution elements that would need to be replaced or modified to accomplish each of the expected or feasible evolution items. The designer must consider whether or not a solution element can be implemented in a way that would significantly limit the necessary changes. For instance, user-defined types enable the actual type of a data element to be changed in a data definition file and permeated throughout the solution via compilation. In this analysis, the designer is primarily looking for changes that cannot be isolated in data definition files.

Table 5.2 contains the results of the analysis of change impact on the data and operations for the microwave oven. In the microwave example, the definition of data types in separate files can eliminate the need to change the operations that use the data of these types. For instance, isolation of the number of each type of device in a user-defined data type localizes changes to the number of a particular device. Therefore, changing the number of power sources (change CHD) does not affect the stop_all_power_sources, increase_all_power_sources, decrease_all_power_sources, and initiate_status_all_power_sources operations.

Table 5.2 Change set of data and operations for each expected or feasible change.

Change Signature of Expected or Feasible Change	Data That Would Be Impacted by the Change ([1]Programming techniques can limit impact of change to data definition files.)	Operations That Would Be Impacted by the Change
HBSQ		straight_heat
DFORM		DEFROST
RFORM		REHEAT
CPSRC		increase_all_power_sources, decrease_all_power_sources
FDBL		heat_operation
IMSWT	error message, status message[1]	DEFROST, REHEAT (e.g., different formulas for English versus metric weights)
HLWS	constants for the limits on the weight, number of servings, and weight per serving[1]	no change impact
CHD		heat_operation, read_all_devices_status
PSRC		initiate_power_source_status, read_power_source_status, increase_power, decrease_power, shut_off_power_source
PSNSR		initiate_power_sensor_read, read_power_sensor_level, read_power_sensor_status
DSNSR		initiate_door_status, read_door_status, read_door_sensor_status
TIMER		convert_to_ticks, set_timer, check_timer_status
APO	heat type[1]	manage_user_interface
EDA		no change impact (new heat_operation with the same interface)

The next step is to locate the date and operations into components that localize the expected or feasible impact of change.

Step 5: Combine and organize into components the overlapping change sets.

One way to localize change to the microwave software is to create a component for each expected or feasible change and to put into the component the data and operations that would be impacted by the change. Figure 5.9 shows the change sets resulting from the change impact analysis for the microwave oven software.

Each set represents a potential component. The reader should note that the change sets do not contain the data whose change impact can be localized in data definition files.

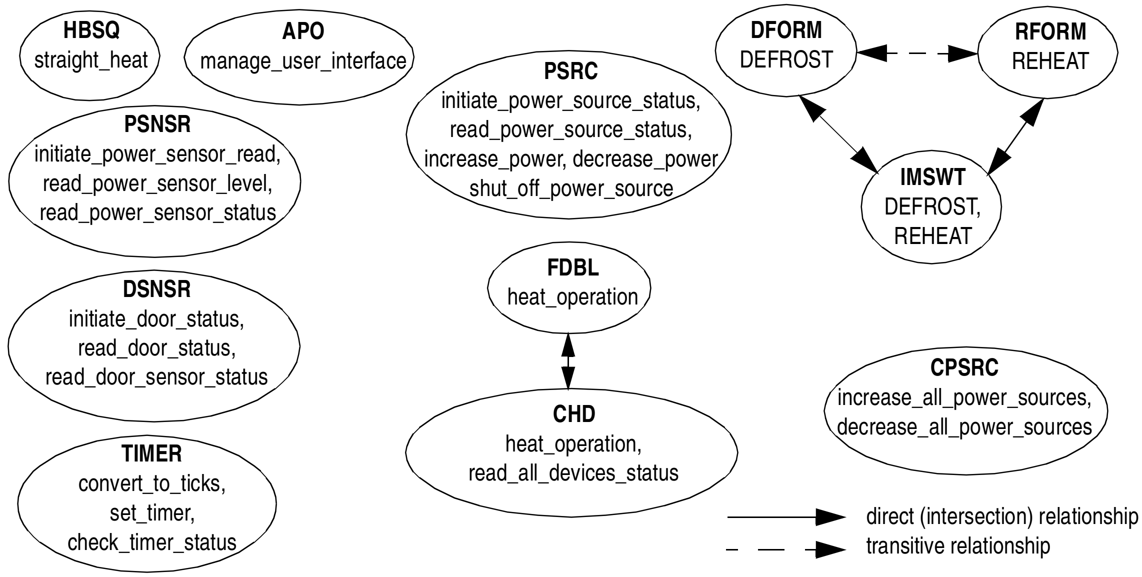


Figure 5.9 Change sets resulting from the change impact analysis.

The problem is that some of the change sets contain the same operations. In Figure 5.9, the change sets with solid arrows connecting them have non-empty intersections. Change sets with broken arrows connecting them are related through transitive closure on the intersection relation. Merging the change sets eliminates the possibility of duplicate operations. Figure 5.10 displays the change sets after the application of the transitive closure on the intersection relation.

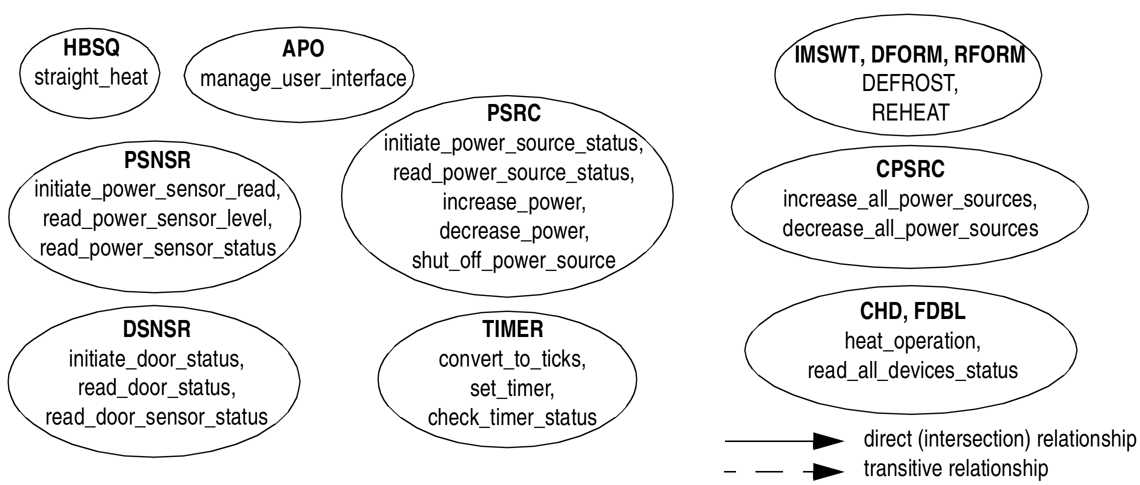


Figure 5.10 Change sets after transitive closure on the intersection relation.

Step 6: Add other necessary components and organize remaining data.

After identifying the components from the change impact analysis, the designer will need to add components for the operations not affected by the changes and to put the remaining data into existing or new components. In the microwave oven software example, the operations not included in any of the change sets are the defrost and reheat operations. The designer should put these into separate components since they are not related with respect to reuse and change. Likewise, the designer can locate the remaining data with the operations that use it. The software engineer may choose to implement the data items as local variables or as parameters which are part of interfaces between operations.

Figure 5.11 shows the organization of the data and operations for the microwave software into components. The components could represent class definitions when an object-oriented approach is being used. Alternatively, the components could represent files that contain the definitions of procedures or functions. The implementor can also locate the software device drivers, *RCG1* members, in a larger component such as a file (C++ version) or directory (java version) to facilitate easy replacement of all of the hardware devices.

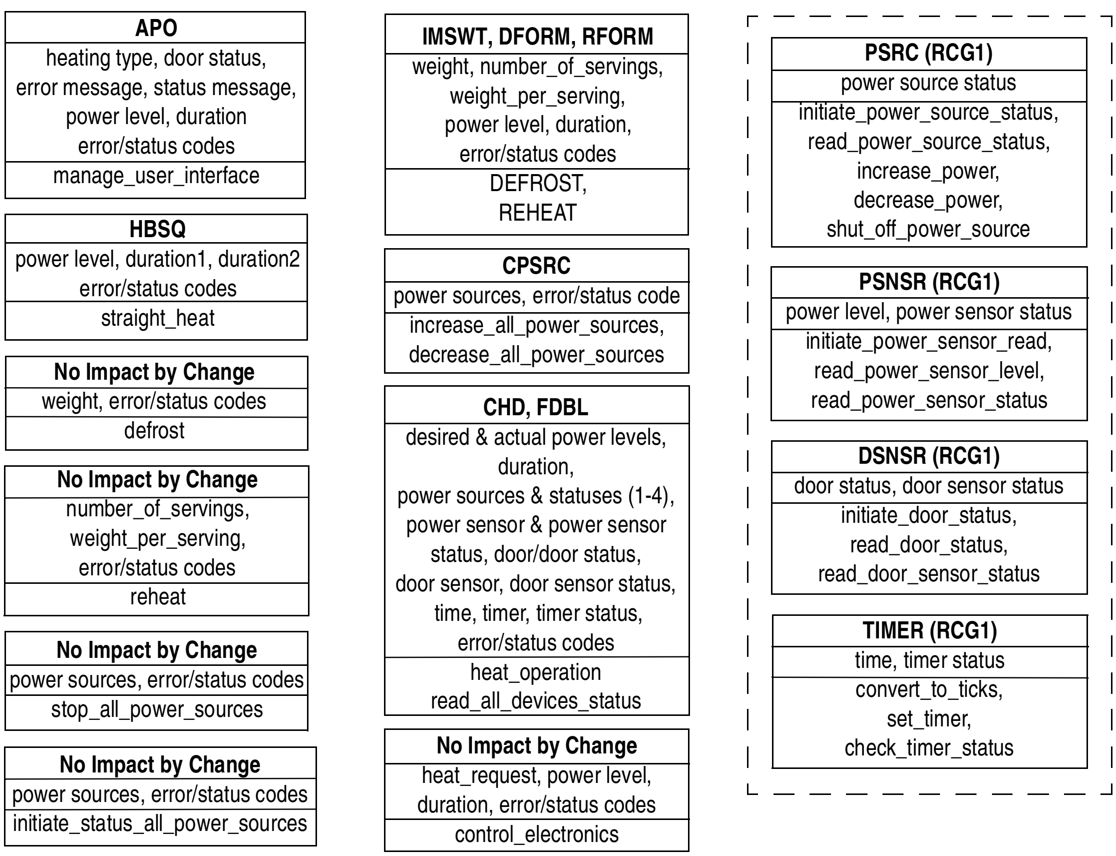


Figure 5.11 Components resulting from the reuse and change impact analysis.

5.4 Mathematical Foundation for Data and Operations Approach

The goal is to automate the partitioning process as much as possible. Automation requires a precise way to represent each step in the partitioning process. This section shows those steps of the process which have a mathematical representation that motivates a semi-automatable partitioning algorithm. The representation for each step of the process appears with application to the microwave oven software. The text notes the decisions that must be made by the designer.

Identifying the basic data and operations and decomposing the large-effect operations are primarily manual processes. As shown in Figure 5.12, we can store the results of these processes in sets *D*, *O*, and *DO*. Likewise, we can program a computer to remember the solution elements (data and operations) associated with particular problem elements (requirements statements) as well as the decompositions of these operations. There is currently no automatable process for determining appropriate solution elements for an arbitrary set of software requirements.

$$D = \{d \bullet DataItem\} = \{powerLevel, duration1, weight, numberOfServings, weightPerServing, \ldots\}$$
$$O = \{o \bullet Operator\} = \{manageUerInterface, straightHeat, defrost, reheat, \ldots\}$$
$$DO = D \cup O$$

Step 1: Identify basic data and operations.

Step 2: Recursively decompose large-effect operations and identify additional data elements.

Figure 5.12 Mathematical representation of steps 1 and 2.

As with steps 1 and 2, enumerating the expected or feasible changes currently requires human analysis. With respect to automation, we can program a computer to remember the changes associated with particular requirements or solutions. More “intelligent” machines in the future may be able to process a set of requirements and suggest ways in which the requirements might evolve. As shown in Figure 5.13, we can store the enumerated changes in a set *C* for use in later steps of the change analysis process.

$$C = \{c \bullet Change\} = \{HBSQ, DFORM, RFORM, CPSRC, \ldots\}$$

Step 3: Enumerate the expected or feasible changes to the software solution and create a set of changes (represented here by their change signatures).

Figure 5.13 Mathematical representation of step 3.

Step 4 involves the semi-automatable process of determining the change set of data and operations associated with each change. One way to help automate this process is to define a change impact relation *CI* that associates each change with a set of data or operations which are impacted by the change. Determining whether or not a change *c* impacts a particular data or operation *do* (the *impact* function) requires human judgement. We can currently program a computer to remember the impact that changes have on specific data and operations, but future programs may be able to determine the impact of a change in requirements on a software design. If a design is expressible in a well-defined design language, then possibly "compiler-like" programs could locate those parts of a design that are related to a requirements change. Currently, the designer must input the data and operations associated with each change $c \, \varepsilon \, C$ (sets of duples representable as a table). Then as shown in Figure 5.14, the computer can create a set *CS* of change sets (one set of impacted data and operations for each change $c \, \varepsilon \, C$).

$$
\begin{aligned}
CI &= \{(c, do) \bullet C \times DO | c \rightarrow impact(do)\} \\
CS &= \{cs \subseteq DO | (\exists c \bullet C | (cs = CI(c)) \wedge (\forall (do \in cs), \exists ((c, do) \in CI)))\} \\
&= \{\{straightHeat\}, \{DEFROST\}, \{REHEAT\}, \{DEFROST, REHEAT\}, \ldots\}
\end{aligned}
$$

Step 4: Identify the change sets.

For each anticipated change *c*, create a change set *cs* of data and operations that are impacted by *c*.
CS is the set of all change sets.

Figure 5.14 Mathematical representation of step 4.

Step 5 is fully automatable. As shown in Figure 5.15, the *Overlap* relation identifies change sets which have non-empty intersections. The *Affinity* relation identifies change sets which are related through the *Overlap* relation (non-empty intersection) or which are related through transitive closure on intersection. *Same-Component* puts together into a single set those change sets which are related through the *Affinity* relation (non-empty intersections or relation through transitive closure on intersection). The result is a set of equivalence classes in which each equivalence class contains the union of the change sets related through the *Affinity* relation. Each one of these equivalence classes (set of data and operations) becomes a software component that will be implemented (packaged) according to the type of programming language and design style being used.

$$Overlap = \{(cs_1, cs_2) \bullet CS \times CS \mid (cs_1 \cap cs_2 \neq \emptyset)\}$$

$$Affinity =$$

$$\{(cs_1, cs_2) \bullet CS \times CS \mid (((cs_1, cs_2) \in Overlap) \vee (\exists(cs_3 \bullet CS);((cs_1, cs_3) \in Affinity \wedge (cs_3, cs_2) \in Affinity)))\}$$

$$SameComponent = \{sc \subseteq CS \mid (\forall(cs_1, cs_2 \in CS);((cs_1 \in sc) \wedge (cs_2 \in sc) \Leftrightarrow (cs_1, cs_2) \in Affinity))\}$$

$$= \{\{straightHeat\}, \{DEFROST, REHEAT\}, \{manageUserInterface\}, \ldots\}$$

Step 5: Combine and organize into components the overlapping change sets.

Figure 5.15 Mathematical representation of step 5.

The last step of adding necessary components and organizing the remaining data is semi-automatable. As shown in Figure 5.16, *NewComponent* defines a set of sets where each set contains precisely one operation which is a member of O and is not associated with any change $c \varepsilon C$. This part of the process is automatable. The last part of the process is to organize the remaining data into the components which contain operations that use the data. The human designer must determine the operations that use each data element $d \varepsilon D$ (the *uses* function) that is not currently in a component.

$$NewComponent$$
$$= \{nc \subseteq O \mid \langle \forall o_1, o_2 \in O, (o_1 \in nc \Rightarrow o_2 \notin nc) \wedge (o_2 \in nc \Rightarrow o_1 \notin nc)\rangle \wedge \langle o \in nc \rightarrow \neg\exists c \bullet C,(c \rightarrow impact(o))\rangle\}$$

$$ExpandedComponent$$
$$= \{ec \subseteq DO \mid \langle(\exists sc \bullet SameComponent, sc \subseteq ec) \vee (\exists nc \bullet NewComponent, nc \subseteq ec)\rangle$$
$$\langle(\forall(d \bullet D, d \in ec \Rightarrow \exists c \bullet C,(c \rightarrow impact(d)))) \vee (\exists o \bullet O,(o \in ec \wedge o \rightarrow uses(d)))\rangle\}$$
$$= \{\{powerLevel, duration1, duration2, errorStatusCode, straightHeat\}, \{weight, \ldots, REHEAT\}, \ldots\}$$

Step 6: Add necessary components and organize remaining data.

Figure 5.16 Mathematical representation of step 6.

Once the designer inputs the operations used by each remaining data item (representable as a table), application of *ExpandedComponent* places each remaining data item into the component that contains one or more operations that use it. As defined, *ExpandedComponent* does not support global data. If more than one component contains operations that use a particular data item, then each component will contain a copy of the data item. This is not likely to happen if the designer is careful to list separately each data that is used by a different part of the software as was done with the error and status codes communicated from one part of the microwave oven software to another. The reader may suggest a definition of *ExpandedComponent* to support global data.

In summary, the research approach is a semi-automatable process that helps the designer to identify, decompose, and partition data and operations into reusable components that localize the impact of changing requirements on the software solution. With the research approach, the following design decisions require human analysis and judgement.

- What data and operations (basic solution elements) are appropriate for a specific set of requirements?
- How can the large-effect operations be decomposed into smaller-effect operations, and should they?
- What are the expected or feasible changes to requirements or the software solution, and should they be decomposed?
- Which data and operations would need to be modified or replaced to implement a specific change?
- Which operations use which data?

The research approach provides the designer with precise guidelines for answering these questions. Most importantly, the answers to these questions enable the automatable partition of solution elements that localize change.

The next section discusses the research approach to organizing the flow of control between solution elements in a way that simplifies changes to control flow.

5.5 Approach for Optimal Partition of Control Flow Components

The software designer's goal is to organize the flow of control between the required tasks in a way that simplifies the process of changing the control flow. The consideration of alternative orders of task execution during the requirements analysis phase can help the software engineer to design for change. There are at least two basic design strategies for simplifying changes to control flow as listed in Figure 5.17.

1. Isolating the knowledge of the order in which tasks should be activated from the implementation of the tasks.
2. Organizing tasks in a way that localizes changes in the order of their execution.

Figure 5.17 Design strategies for simplifying changes to control flow.

The first strategy is to decouple the part of a software solution which may change (in this case the ordering of task activations) from the part not involved in the change (the tasks themselves). The object-oriented design community has codified a design pattern called *Mediator* that is an application of this strategy. The objective of this pattern is to simplify changes to the interactions between a group of objects. This is done by isolating the application-specific interactions between *Colleague* objects within *Mediator* objects which act as coordinators [50].

In general, changes to control flow are more easily done when the activations of the required tasks are located in components that are separate from the components containing the implementations of the task. This can be done by embedding the high level flow of control in control components that activate task components. The problem is how to organize the task activations into control flow components so that the resulting control components can be easily modified to satisfy feasible changes to requirements. This section discusses two processes for partitioning task activations into control flow components: (1) an optimal but complex partitioning process, and (2) a heuristic for obtaining a good, though not optimal, partition of task activations. Lastly, the section will discuss how these processes also apply to the second strategy of organizing the tasks (without separating the task activations from the tasks) in a way that localizes changes in the order of their execution.

Figure 5.18 defines the concepts used to explain the research approach to partitioning control flow.

- A **sequence *S*** of tasks, $S = <t_1, t_2,..., t_n>$, refers to the execution order of a set of tasks T, $T = \{t_1, t_2, ..., t_n\}$ where n is the cardinality of the set. An **empty sequence *S***, denoted ***S* = <>**, is a sequence for which $T = \emptyset$.
- The **length of a sequence *S***, denoted **|*S*|**, is the number of task activations in the sequence.
- Given two sequences $X = <x_1, x_2,..., x_n>$ and $Y = <y_1, y_2,..., y_m>$, ***X*|*Y*** (read "***X* concatenated with *Y***") is a sequence S such that $S = <x_1, x_2,..., x_n, y_1, y_2,..., y_m>$.
- A sequence $X = <x_1, x_2,..., x_m>$ is a **subsequence of *S*** $= <t_1, t_2,..., t_n>$ if there exists a mapping from X to S such that $x_1 = t_i, x_2 = t_{i+1},..., x_m = t_{i+m-1}$ and $X \neq S$.
- Readers should note that the above definition of a subsequence differs from a definition in which a sequence $Z = <z_1, z_2,..., z_k>$ is a subsequence of sequence $X = <x_1, x_2,..., x_m>$ if there exists a strictly increasing sequence $<i_1, i_2,..., i_k>$ of indices of X such that for all $j = 1, 2,..., k$, $x_{ij} = z_j$. With this definition, $X = <a,c,e>$ would be a subsequence of $Y = <a,b,c,d,e>$ [41]. With the definition that applies to the research approach, $<a,b,c>$ is a subsequence of Y; while $<a,c,e>$ is not.
- Given a sequence $S = <t_1, t_2,..., t_n>$ and $T =\{t_1, t_2, ..., t_n\}$, a **singleton subsequence of *S*** activates precisely one task t_i, where t_i e T.
- A **partition *P* of a sequence *S*** is an ordered set of subsequences, $\{X_1, X_2,..., X_n\}$ where n is the cardinality of the set, such that $S = X_1 | X_2 |...| X_n$. Given that $S = <a,b,c>$, the minimal partition of S is $\{<a,b,c>\}$. The maximal partition of S is $\{<a>,<b>,<c>\}$. For simplicity, let the ordered set of subsequences without the set notation represent a partition. From here on, the representation for the minimal partition of $S = <a,b,c>$ is $<a,b,c>$; and the representation for the maximal partition of S is $<a><b><c>$. The set of all feasible partitions of S is $\{<a><b><c>, <a><b,c>, <a,b><c>, <a,b,c>\}$.
- A **loop *L*** of a sequence S, denoted **Loop(*S*)**, is a repeating sequence such that $L = <>$ or $L = S \,|\, \text{Loop}(S)$.

Figure 5.18 Terminology for the research approach to partitioning control flow.

Partitions generated using the research approach should preserve the required order of task activations in the control components. It is therefore not meaningful to group a required order of task activations <*a*,*b*,*c*> into control components containing <*a*,*c*> and <*b*>. Therefore for the purposes of the research approach, <*a*,*c*><*b*> is not a partition of <*a*,*b*,*c*>; and <*a*,*c*> is not a subsequence of <*a*,*b*,*c*>.

The step-wise process outlined in Figure 5.19 enables the designer to generate the control flow components that will optimally reduce the complexity of evolving the required control flow.

1. Develop a metric for determining the difficulty and “error-proneness” of modifying the control components to make a required change in control flow.
2. Determine the way in which the control components would have to be modified and relate this process to the metric. In other words, determine how the modification steps relate to the counting that must be done for the metric.
3. Express the required control flow as a sequence and generate all partitions of this sequence.
4. For each partition and for each alternative order to the required control flow, “walk-through” the necessary modifications to change the required sequence and apply the metric. Save these values in a table that stores the change complexity value for each partition with respect to every alternative sequence.
5. For each partition, sum the change complexity values to derive the total across all alternative sequences. The partition with the minimum total is the best design choice for the required control flow and proposed alternative sequences.

Figure 5.19 Process for generating optimally “good” control flow components.

The following text and diagrams demonstrate the process with an example from the embedded microwave oven software discussed in Chapter 2.

The requirements statement for the microwave oven specifies a simplistic algorithm for controlling the electronics. The required sequence of application-level tasks within the loop are <*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*> where each letter corresponds, as shown in Figure 5.20, to the activation of a specific task for controlling the electronics. *Loop*(<*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*>) represents the loop of tasks in the order specified by the sequence <*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*>. The full sequence is then <*A*> | *Loop*(*S*) or <*A*> | *Loop*(<*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*>). The requirements statement also specifies that microwave ovens whose electronics respond faster would allow for different feedback loops. The required as well as the alternative sequences of tasks which are feasible appear in Figure 5.21. Note that changes from the original sequence only involve tasks within the loop. Therefore the focus is initially on the loop sequence <*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*>. A discussion of how to incorporate the design of task *A* and the loop logic as well as how to handle modifications that move tasks into or out of a loop appears later.

A - Set the timer to the desired duration.
Loop
- *B* - Initiate the door status.
- *C* - Initiate the power source status.
- *D* - Initiate the power sensor read.
- *E* - Read the power sensor to check the actual power achieved.
- *F* - If the power is under the desired level, increase the power by 3 notches on all sources.
- *G* - If the power is over the desired level, decrease the power by 3 notches on all sources.
- *H* - Check the timer. If it has expired, then stop all of the power sources and return a status code.
- *I* - Read the status of all of the electronics. If any have malfunctioned or if the door is open, stop all of the power sources.

Endloop

Figure 5.20 Order of tasks in the Control Electronics of the microwave oven.

Original or required sequence of tasks:
<*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*>

Alternative task sequences:
<*B*,*C*,*D*,*E*,*F*,*G*,*I*,*H*>
<*B*,*C*,*D*,*I*,*E*,*F*,*G*,*H*>

Figure 5.21 Required and alternative task sequences for the Control Electronics of the microwave oven.

Given a required sequence of tasks and alternative sequences of these tasks, one can apply the optimal process described below to design the control logic. The first step is to develop a change complexity metric to measure the human effort needed to change the control flow components.

Step 1: Develop the change complexity metric.

The proposed metric is based upon the following rationale. The human performs the modifications to software to satisfy changing requirements. Localized change involves fewer components and less effort than nonlocalized change. Small, simple components are easier to understand and modify correctly than are large, complex components. A basic way to determine the effect of a change is to sum the size of each control flow component which must be modified or, in this case, to count the number of task activations contained in the modified control flow components. The resulting change complexity metric is shown in Figure 5.22. The reader should see Chapter 4 for a discussion of factors which affect the complexity of modifying software and of the metrics which represent these factors. Component size, or software size, in general is a major contributor to the complexity of maintaining software.

Given x is a component to be modified, let $|x|$ be the size of x. Let X be the set of all x.

$$\text{Change complexity metric} = \sum_{x_i \in X} |x_i|.$$

Figure 5.22 Change complexity metric to measure the effort to modify control flow components.

Step 2: Determine the process for modifying control components and relate it to the change complexity metric.

There are two basic types of designs for activating the control components which execute the task activations. In the first type, a master controller component activates the other control components in the proper order; whereas in the second type, the control components trigger each other. Suppose the required sequence within the loop shown in Figure 5.21 is partitioned into the subsequences <*B*,*C*,*D*> and <*E*,*F*,*G*,*H*,*I*>. With the Master Control architecture, shown in Figure 5.23, the Master Controller activates the control components C_1 and C_2. Component C_1 activates the tasks *B*, *C*, and *D*; and component C_2 activates tasks *E*, *F*, *G*, *H*, and *I*.

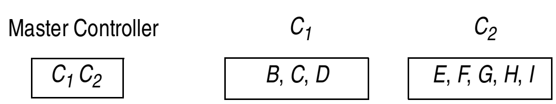


Figure 5.23 Master Control architecture for control flow components.

With the trigger architecture, diagrammed in Figure 5.24, component C_1 activates the tasks *B*, *C*, and *D*, before activating component C_2. Component C_2 activates tasks *E*, *F*, *G*, *H*, and *I*.

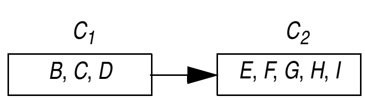


Figure 5.24 Trigger architecture for control flow components.

The Master Control architecture isolates the knowledge of the order of the control components from these components. For instance with the Master Control architecture, component C_1 has no "awareness" that component C_2 follows it in the execution sequence. With the trigger architecture, component C_1 is aware of component C_2 because it must activate C_2. Reversing the order of C_1 and C_2 requires a reorder of the C_1 and C_2 activations in the Master Controller. The complexity of this change is the size of the master controller component which must be modified. The Master Controller contains two task activations for C_1, and C_2. Figure 5.25 shows the change complexity.

change complexity = size(Master Controller) = 2 task activations

Figure 5.25 The size of the Master Controller represents the complexity of modifying this component.

Making the same change with the trigger architecture requires changing C_1 to not activate C_2 and C_2 to activate C_1. Alternatively, all of the task activations in C_2 could be moved to C_1 and all of those in C_1 to C_2. The latter process involves the same components and therefore yields the same change complexity. The change complexity is the sum of the sizes of components C_1 and C_2 as shown in Figure 5.26.

change complexity = size(C_1) + size(C_2) = 3 + 5 = 8 task activations

Figure 5.26 The sum of the sizes of the modified components C_1 and C_2 represents the change complexity.

In this case, the Master Control architecture is easier to change than the trigger architecture.

From here on, the examples apply the Master Control type of control flow architecture. In practice, the designer can compare the change complexities for designs which apply different control flow architectures for isolating task activations from the implementation of tasks. The next steps serve to find the partition of the required task activation sequence which minimizes the complexity of changing to alternative sequences.

Step 3: Express the required control flow as a sequence and generate all partitions of this sequence.

The required loop sequence is <*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*>. A process for enumerating all of the partitions appears in Figure 5.27. The interested reader can generate the partitions not shown in Figure 5.27. The complete list of partitions appears in Appendix C.

1. <*B*><*C*><*D*><*E*><*F*><*G*><*H*><*I*>
2. <*B*><*C*><*D*><*E*><*F*><*G*><*H*,*I*>
3. <*B*><*C*><*D*><*E*><*F*><*G*,*H*><*I*>
4. <*B*><*C*><*D*><*E*><*F*><*G*,*H*,*I*>
5. <*B*><*C*><*D*><*E*><*F*,*G*><*H*><*I*>
6. <*B*><*C*><*D*><*E*><*F*,*G*><*H*,*I*>

...

126. <*B*,*C*,*D*,*E*,*F*,*G*><*H*,*I*>
127. <*B*,*C*,*D*,*E*,*F*,*G*,*H*><I>
128. <*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*>

Figure 5.27 Partitions of the required task sequence.

Step 4: For each partition with respect to each alternative sequence, do the following sub-steps.

- Determine the modifications needed to derive the alternative sequence from the original sequence.
- Apply the metric to determine the complexity of the modifications.
- Store the change complexity values in a table.

Some partitions, such as <*B*><*C*><*D*><*E*><*F*><*G*><*H*><*I*>, involve singleton subsequences. Placing the task activation from a singleton subsequence directly in the Master Controller eliminates the redundancy of "activating a control flow component which activates precisely one task." This approach does not affect the overall change complexity: the size of the Master Controller increases by one for each activation of a single task and by one for each activation of a control flow component that activates a single task. Therefore, the Master Control architecture for the partition <*B*><*C*><*D*><*E*><*F*><*G*><*H*><*I*> consists of one master control component which contains all of the task activations. The Master Control architecture for the partition <*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*> locates the activations of all of the tasks in a single control flow component. In this case, the Master Controller would contain only the activation of the resulting control flow component. The extra level of indirection yields no benefit. A better design strategy is to place all of the task activations in the Master Controller. For both of the partitions discussed above, any change to the required control sequence involves the Master Controller and therefore has a change complexity which is the size of the Master Controller.

To determine the change complexity for each partition and for each change, the designer must walk-through the modifications to the architecture for a particular partition and sum the size of the components which are modified. There are two alternative sequences considered for this example. Modifications to the required sequence are not cumulative: to obtain each alternative sequence, the designer starts with the control flow components of a particular partition and modifies them to obtain the desired alternative sequence. Figure 5.28 and Figure 5.29 provide example walk-throughs for two different partitions and both alternative sequences. Columns 2 and 3 of the table in Appendix C list the change complexity values for modifying the control flow components resulting from a particular partition to change the required sequence <*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*> to alternative sequences <*B*,*C*,*D*,*E*,*F*,*G*,*I*,*H*> or <*B*,*C*,*D*,*I*,*E*,*F*,*G*,*H*>, respectively.

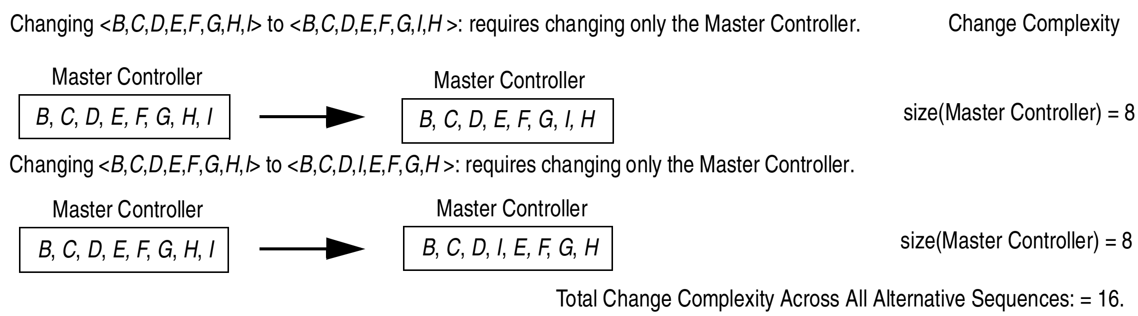


Figure 5.28 Walk-throughs for partition *<B><C><D><E><F><G><H><I>*.

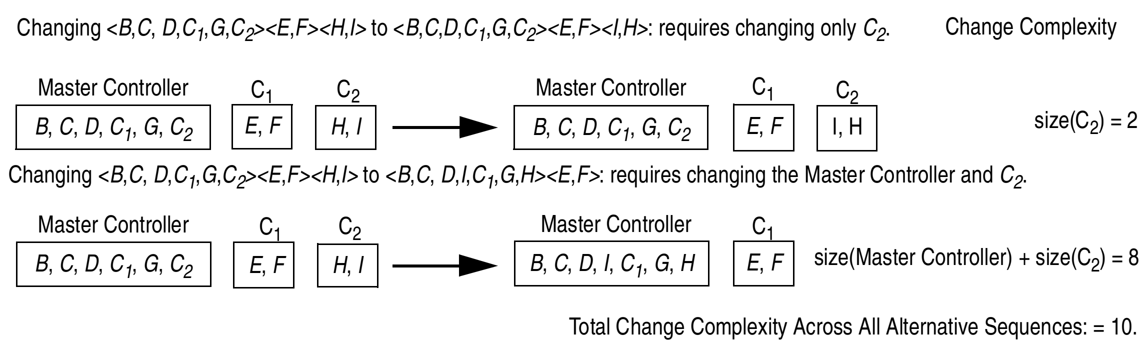


Figure 5.29 Walk-throughs for partition *<B><C><D><E,F><G><H,I>*.

The next step is to sum, for each partition, columns 2 and 3 of the table that contains the change complexity values across the alternative sequences.

Step 5: For each partition, sum the values to derive the total across all alternative sequences.

The interested reader may complete the change complexity table, which is shown in Appendix C, for all 128 feasible partitions and sum columns 2 and 3 for each row to determine the partitions with the lowest complexity value. The partitions ending with *<E,F,G><H,I>* (namely, *<B><C><D><E,F,G><H,I>*, *<B><C,D><E,F,G><H,I>*, *<B,C><D><E,F,G><H,I>*, and *<B,C,D><E,F,G><H,I>*) have a minimal total complexity value of 7. These partitions isolate the parts of the original sequence *<B,C,D,E,F,G,H,I>* which change. In this case, the subsequences *<E,F,G,H,I>* and *<H,I>* contain the parts which change (*<E,F,G,H,I>* to *<I,E,F,G,H>* and *<H,I>* to *<I,H>*. Partitioning *<E,F,G,H,I>* into *<E,F,G>* and *<H,I>* lowers the total complexity across both changes because it also isolates the part which does not change, the invariant *<E,F,G>*. This pattern (invariant sequence of tasks) motivates a heuristic, discussed in the next section, for reducing the complexity of obtaining a good partition of the required control flow sequence.

5.6 Approach for Heuristically Good Partition of Control Flow Components

The previous section discussed a process for determining the partition of the required sequence of task activations that optimally minimizes the effort needed to generate alternative sequences. The problem is that this is a complex process. Walking through the component modifications to determine the complexity of changing the required sequence to an alternative sequence is a tedious and manual process that is required for each partition across all of the alternative sequences. The number of partitions of a sequence S increases exponentially with the length of S: the cardinality of the set of all partitions is 2^{n-1} where n is the length of S. For small values of n, this can be done manually; but as n increases, the time required to find the optimal partition is costly. The computational complexity of the process is $(O(2^n))$. For the interested reader, the proof for the number of partitions of a sequence appears in Figure 5.30.

Hypothesis: The number of partitions for a sequence of length n is 2^{n-1}.

Proof:
Let $X_1 = \langle x_1 \rangle$ and $X_2 = \langle x_1, x_2 \rangle$ be the base sequences.

- The length of X_1 is 1, and the length of X_2 is 2. X_1 has exactly one partition, $\langle x_1 \rangle$. The number of partitions of X_1 is $1 = 2^{(1-1)} = 2^{n-1}$ for $n = 1$.
- X_2 has exactly two feasible partitions, $\langle x_1 \rangle\langle x_2 \rangle$ and $\langle x_1, x_2 \rangle$. Therefore the number of partitions of X_2 is $2 = 2^{(2-1)} = 2^{n-1}$ for $n = 2$.

Let us assume that the number of partitions for a sequence X_j of length j is 2^{j-1}.
Now we must show that the inductive hypothesis is true for sequences of length j+1.

Let us form a sequence X_{j+1} by concatenating the sequence X_1 with sequence X_j, $X_1 \mid X_j$. The length of X_{j+1} is the length of X_1 plus the length of X_j or j+1.

There are strictly two ways to form partitions of X_{j+1}.

- One way is to group x_1 by itself. The number of possible ways to group the other elements of X_{j+1}, namely those contributed by X_j, would be the number of partitions of X_j or 2^{j-1}.
- The other way is to group x_1 with the first element of X_j. Likewise, the number of partitions containing x_1 that are grouped with the first element of X_j is the number of partitions of X_j or 2^{j-1}.

So the total number of partitions of X_{j+1} is $2^{j-1} + 2^{j-1} = 2^j = 2^{n-1}$ for $n = j+1$.

Hence, the inductive hypothesis holds for sequences of length j+1.

Figure 5.30 Inductive proof for the number of partitions of a sequence.

One could apply a combinatorial optimization algorithm such as a genetic or simulated annealing algorithm [53,88]. The change complexity metric would provide the rationale for an objective function. The primary issue would be how to automate the process of “walking-through” the change to a control flow sequence in order to determine the change complexity value. Constraining the design objectives eliminates the need to generate all of the partitions. What is needed is a polynomial-time heuristic for determining a good, though not necessarily optimal, partition of the control flow into components. This section describes a partitioning pattern that tends to minimize the overall change complexity. The remainder of this section proves the "*goodness*" of this pattern for most potential changes to control flow and notes the exceptional cases. Lastly, the section outlines a polynomial process for determining control components that exhibit this pattern.

Locating each longest invariant subsequences, such as $\langle B,C,D \rangle$ in the microwave oven example, in a single component by itself helps to reduce the overall change complexity. Figure 5.31 presents the formal definitions for the terms *invariant subsequence* and *longest invariant subsequence*.

Let:

- $S = \{s_i\}$ = the set of plausible sequences of task activations, where $s_i = \langle t_{i1}, \ldots, t_{in} \rangle$ and $t_{ij} \neq t_{ik}$ for every $j \neq k$.
- $T = \{t_i\}$ is the set of application-level tasks.

If $v = \langle t_1, \ldots, t_m \rangle$ with $1 < m \leq n$, then Invariant(v) $\leftrightarrow \forall\ s_i, \exists\ j$: $t_{ij} = t_1, \ldots, t_{i(j+m-1)} = t_m$.

LI, the set of longest invariant subsequences, = $\{l_j\}$: Invariant(l_j) $\wedge \neg$ ($\exists\ l_k$: Invariant(l_k) $\wedge$ l_j = Subsequence(l_k)).

In the microwave oven example, $\langle B,C \rangle$ and $\langle E,F \rangle$ are invariant subsequences; whereas $\langle B,C,D \rangle$ and $\langle E,F,G \rangle$ are longest invariant subsequences. $\langle B,C,D,E,F,G \rangle$ is not an invariant subsequence because in future versions of the microwave oven software, task I may occur between tasks D and E.

Figure 5.31 Definitions of an invariant subsequence and a longest invariant subsequence.

From a locality point of view, placing each longest invariant subsequence into a single component by itself intuitively makes sense: those subsequences which may change are separate from those subsequences which are not likely to change. Components that contain invariant subsequences would not need to be involved in future changes to the control flow. Verification of this logic requires answers to the following questions.

1. Are there cases when the placement of a longest invariant subsequence in a single component does not reduce the change complexity metric (does not simplify change)?
2. Are there cases when altering a component that contains a longest invariant subsequence helps to reduce the change complexity metric (simplifies change)?
3. How likely are the exceptional cases to occur?

Validating the ease of using the invariant subsequence pattern requires answers to the following questions.

1. How does one locate a longest invariant subsequence automatically?
2. How can the location of longest invariant subsequences be combined with the optimal partitioning approach?

The answer to the first two questions is an analytical proof that placing each longest invariant subsequence into a single control flow component usually, but not always, reduces the change complexity metric.

Proof of the Longest Invariant Subsequences Heuristic:

Let the statements in Figure 5.32 hold.

- $S = <X, Y, Z>$ is a sequence of task activations with subsequences X, Y, and Z.
- Y is a longest invariant subsequence in S and has a length greater than or equal to 2. $Y = <y_1,\ldots, y_{|Y|}>$, $|Y| \geq 2$.
- P(X) is a partition of a subsequence X.
- MC represents the master controller component.
- [X] represents a component containing a subsequence X, and [P(X)] represents the components containing a partition of X.
- If X is empty, then [P(X)] consists of no components.
- C([X]), the **change complexity for a component containing a subsequence *X***, be:
  - 0, if the component need not be changed.
  - $|[X]| = |X|$, the size of the component containing X which equals the **length of *X***, if the component must be changed. $C([MC]) = |MC|$.
- $C([P(X)]) = \sum C([x_i])$ where $x_i \in P(X)$.

Figure 5.32 Definitions for analyzing the goodness of the invariant subsequence pattern.

The general form for partitioning a sequence *XYZ* that contains a longest invariant subsequence *Y* is shown in Figure 5.33. From here on, the term invariant subsequence pattern refers to this partition.

The general form for partitioning a sequence *XYZ* with a longest invariant subsequence *Y* into components is:

$$[P(X)]\ [Y]\ [P(Z)].$$

If either X or Z is empty, then [P(X)] or [P(Z)] contains no components, respectively.
$MC_{P(XYZ)}$ contains a call to [Y] as well as to each component in [P(X)] and [P(Z)].

Figure 5.33 Longest invariant subsequence pattern.

Since *X* and *Z* are not invariant, there must be one or more ways in which they are likely to change. There are three ways to change a sequence, as listed in Figure 5.34.

1. Add a new subsequence to the sequence.
2. Delete a subsequence from the sequence.
3. Reorder the tasks in the sequence.

Figure 5.34 Three basic ways to change a sequence.

We will analyze each type of change to a variant sequence and outline the exceptional cases. Exceptional cases are those in which the change complexity is not minimized by the longest invariant subsequence pattern.

Adding a New Subsequence to a Sequence:

Suppose we wish to add the subsequence *U* to *XYZ*. There are three general locations for U as listed in Figure 5.35.

1. In between two tasks from *X* or between two tasks from *Z*.
2. At the beginning of *X* or at the end of *Z*.
3. At the end of *X* or at the beginning of Z.

Figure 5.35 Locations for adding a subsequence *U* to *XYZ* where *Y* is an invariant subsequence.

For the first location, component(s) from strictly [P(*X*)] or from strictly [P(*Z*)] must be involved in the change. The addition of tasks from *Y* to the [P(*X*)] or [P(*Z*)] components would increase the change complexity value. Therefore, locating *Y* in a single component by itself helps to reduce the change complexity.

For the second location, there are several ways to add *U* to the beginning of *X* or to the end of *Z* as discussed in Figure 5.36.

- If *U* is not a singleton, select one of the following options to position *U* appropriately and to minimize the change complexity metric.
  - Create a new component [*U*], and add a call to it in the *MC* before the calls to the component(s) of [P(*X*)] or after the calls to the component(s) of [P(*Z*)].
  - Add *U* to the beginning of the first [P(*X*)] component or to the end of the last [P(*Z*)] component.
- If *U* is a singleton, select one of the following options to position *U* appropriately and to minimize the change complexity metric.
  - Add *U* to the *MC*.
  - Add *U* to the beginning of the first [P(*X*)] component or to the end of the last [P(*Z*)] component.

Figure 5.36 Adding a subsequence *U* at the beginning of *X* or at the end of *Z*.

As with the first location, component(s) from strictly [P(*X*)] or from strictly [P(*Z*)] or the *MC* must be involved in the change. The addition of tasks from *Y* to the [P(*X*)] or [P(*Z*)] components could increase the change complexity value. Therefore, locating *Y* in a single component by itself helps to reduce the change complexity.

For the third location, there are several ways to add *U* to the end of *X* or to the beginning of *Z* as described in Figure 5.37.

- If *U* is not a singleton, select one of the following options to position *U* appropriately and to minimize the change complexity metric.
  - Create a new component, [*U*], and add a call to it in the *MC*.
  - Add *U* to the end of the last [P(*X*)] component or to the beginning of the first [P(*Z*)] component.
  - Add *U* to the beginning or end of [*Y*].
- If *U* is a singleton, select one of the following options to position *U* appropriately and to minimize the change complexity metric.
  - Add *U* to the *MC*.
  - Add *U* to the end of the last [P(*X*)] component or to the beginning of the first [P(*Z*)] component.
  - Add *U* to the beginning or end of [*Y*].

Figure 5.37 Adding a subsequence *U* at the end of *X* or at the beginning of *Z*.

Sometimes, C[*Y*] is less than the complexity of other appropriate components for locating *U* such as [*U*], a component in [P(*X*)], a component in [P(*Z*)], or [*MC*]. For example, suppose the sequence <*a,b,c,d,e,f,g,h,i,j,k*> contains a longest invariant subsequence <*e,f,g*>. Following the longest invariant pattern, the general partition of <*a,b,c,d,e,f,g,h,i,j,k*> is [P(<*a,b,c,d*>)] [<*e,f,g*>] [P(<*h,i,j,k*>)]. $MC_{\mathrm{P}(<a,b,c,d,e,f,g,h,i,j,k>)}$ contains calls to the components of [P(<*a,b,c,d*>)] and [P(<*h,i,j,k*>)] as well as to [<*e,f,g*>]. Adding the subsequence <*o,p*> between the tasks *d* and *e* can be done in three ways.

1. Create a component [<*o,p*>] and add a call to it in *MC* after the call(s) to the [P(<*a,b,c,d*>)] components. C([*MC*]) ≥ 3, depending on the number of components in [P(<*a,b,c,d*>)] and [P(<*h,i,j,k*>)].
2. Add <*o,p*> to the end of the last component in [P(<*a,b,c,d*>)].
   2 ≤ C(last component in [P(<*a,b,c,d*>)]) ≤ 4.
3. Add <*o,p*> to the beginning of [<*e,f,g*>]. C[<*e,f,g*>] = 3.

The worst cases occur when [P(<*a,b,c,d*>)] is [<*a,b,c,d*>] and when [P(<*h,i,j,k*>)] is [<*h,i*>] [<*j,k*>], [<*h,i,j*>] with *k* in the *MC*, or [<*i,j,k*>] with *h* in the *MC*. In a worst case, the complexity of changing the invariant component is less than changing either [P(<*a,b,c,d*>)] or *MC*: C[<*e,f,g*>] = 3 ≤ 4 = [<*a,b,c,d*>] = C([*MC*]).

When a new subsequence *U* should be located immediately before or after an invariant subsequence *Y*, adding *U* to the component that contains *Y* may result in a lower change complexity value. This depends on the composition of *X*, *Y*, *Z* as well as on the P(*X*) and P(*Z*) which minimize the complexity across all feasible changes to control flow. For all other placements of a new subsequence *U*, involving the invariant *Y* will increase rather than reduce the change complexity.[68]

Deleting a Subsequence from a Sequence:

A subsequence to be deleted must exist in one or more components of [P(*X*)] or [P(*Z*)]. The removal of the subsequence does not involve [*Y*]. The problem is that the deletion may result in an empty component or in a singleton. Let us consider each situation separately. When the entire subsequence of a component in [P(*X*)] or [P(*Z*)] is removed, the *MC* will contain a call to an empty component. There are several actions that can be taken as explained in Figure 5.38.

1. Leave the *MC* as it was before the removal of the subsequence and let it call an empty component (at the expense of the overhead for a call). The change complexity is zero.
2. Remove the call from the *MC*. The change complexity is C([*MC*]).
3. Move two tasks from an adjacent component to the empty component. This can be done if a component adjacent to the empty component has four or more tasks in it. The change complexity is C([component from which two tasks are taken]) + C([component from which the subsequence is removed]).
4. Repartition the tasks in the components closest to the empty component so that two tasks from a component adjacent to the empty component are available. The change complexity is C([repartitioned components]) + C([component from which the subsequence is removed]). This action includes action 3.

Figure 5.38 Alternative actions when deletion of a subsequence results in an empty component.

The intuitively simple approach is action two. But if the *MC* is large, the change complexity value of *MC* may be more than the change complexity value(s) of the component(s) needed to move tasks into the empty component (associated with actions 3 or 4). Use of the component [*Y*], which contains the invariant subsequence, may reduce the change complexity value. As before, this depends on the composition of *X*, *Y*, *Z* as well as on the P(*X*) and P(*Z*) which minimize the complexity across all feasible changes to control flow. Figure 5.39 illustrates an exceptional case when deletion of a subsequence results in an empty component and altering the invariant subsequence reduces the change complexity value.

Suppose that we start with a sequence <*a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p*> that contains an invariant subsequence <*h,i,j,k*>. The general partition P(<*a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p*>) is [P(<*a,b,c,d,e,f,g*>)] [<*h,i,j,k*>] [P(<*l,m,n,o,p*>)]. $\mathrm{MC}_{\mathrm{P}(<a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p>)}$ contains calls to the components of [P(<*a,b,c,d,e,f,g*>)] and [P(<*l,m,n,o,p*>)] as well as to [<*h,i,j,k*>].

Now suppose that we want to delete subsequence <*f,g*>. If P(<*a,b,c,d,e,f,g*>) is [<*a,b,c,d,e*>] [<*f,g*>], then deletion of <*f,g*> will result in an empty component. Suppose further that [P(<*l,m,n,o,p*>)] contains two components.

The complexity of removing the call to [<*f,g*>] from *MC* is C([*MC*]) = 5. The complexity to move tasks *d* and *e* from [<*a,b,c,d,e*>] to the empty component is C[<*a,b,c,d,e*>] = 5. The complexity to move tasks *h* and *i* from the invariant component [<*h,i,j,k*>] to the empty component is C[<*h,i,j,k*>] = 4. In this example, altering the invariant subsequence reduces the change complexity value.

Figure 5.39 Alteration of the invariant subsequence after deletion results in an empty component.

Deletion of a subsequence may result in a singleton. As explained in Figure 5.39, there are several actions that can be taken.

1. Leave the singleton in the original component. The change complexity is zero.
2. Put the singleton into the *MC* in place of the call to the component that had contained it. The change complexity is C([*MC*]).
3. Move a task from an adjacent component into the component that contains the singleton. This can be done if a component adjacent to the empty component has three or more tasks in it. The change complexity is C([component from which the task is taken]) + C([component from which the subsequence is removed]).
4. Repartition the tasks in the components closest to the empty component, so that a task from a component adjacent to the empty component is available for the empty component. The change complexity is C([repartitioned components]) + C([component from which the subsequence is removed]). This action includes action three.

Figure 5.40 Alternative actions when deletion of a subsequence results in a singleton.

Again, the intuitively simple approach is action two. But if the *MC* is large, the change complexity value of *MC* may be more than the change complexity value(s) of the component(s) needed to move a task into the component that contains the singleton (associated with actions 3 or 4). Use of the component [*Y*], which contains the invariant subsequence, may reduce the change complexity value. As before, this depends on the composition of *X*, *Y*, *Z* as well as on the partition which minimizes the complexity across all feasible changes to the control flow. Figure 5.41 illustrates an exceptional case when deletion of a subsequence results in a singleton component and altering the invariant subsequence reduces the change complexity value.

Suppose that we start with a sequence <*a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p*> as described in Figure 5.39. Now suppose that we want to delete subsequence <*f*>. If P(<*a,b,c,d,e,f,g*>) is [<*a,b,c,d,e*>] [<*f,g*>], then deletion of <*f*> will result in a singleton component [<g>]. Suppose further that [P(<*l,m,n,o,p*>)] contains two components.

The complexity of replacing the call to [<*f,g*>] with *g* in *MC* is C([*MC*]) = 5. The complexity to move task *e* from [<*a,b,c,d,e*>] to the singleton component is C[<*a,b,c,d,e*>] = 5. The complexity to move task *h* from the invariant component [<*h,i,j,k*>] to the singleton component is C[<*h,i,j,k*>] = 4. In this example, altering the invariant subsequence reduces the change complexity value.

Figure 5.41 Alteration of the invariant subsequence after deletion results in a singleton.

Reordering the Tasks in a Sequence:

Reordering the tasks in a sequence is a combination of deleting and adding subsequences: a subsequence is removed from one part of the original sequence and added to another part. The previous discussions about deleting and adding subsequences apply. The difference is that one must consider the adding actions that will be feasible after each possible deletion action. The idea is to minimize the total change complexity, the sum of the deletion and addition complexities. Figure 5.42 is an example of a reordering that results in an empty component after the deletion action.

> Suppose that we start with a sequence <*a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p*> that contains an invariant subsequence <*h,i,j,k*>. The general partition is [P(<*a,b,c,d,e,f,g*>)] [<*h,i,j,k*>] [P(<*l,m,n,o,p*>)].
>
> *Change 1*: Suppose we want to be able to easily reorder the original sequence to produce a new sequence <*c,d,e,f,g,h,i,j,k,a,b,l,m,n,o,p*>. A good partition for this change would be [P(<*a,b,c,d,e,f,g*>)] as [<*a,b*>] [<*c,d,e,f,g*>] and [P(<*l,m,n,o,p*>)] as [<*l,m,n,o,p*>]. $MC_{P(<a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p>)}$ would call [<*a,b*>], [<*c,d,e,f,g*>], [<*h,i,j,k*>], and [<*l,m,n,o,p*>]. Reordering would involve "deleting" the call to [<*a,b*>] in *MC* from its position before the call to [<*c,d,e,f,g*>] and "adding" it back to the *MC* in the position directly after the call to [<*h,i,j,k*>]. C[*MC*] would be 4.
>
> *Change 2*: Suppose we also want to be able to easily reorder the original sequence to produce a new sequence <*a,b,o,p,c,d,e,f,g,h,i,j,k,l,m,n*>. A good partition for this change would be [P(<*a,b,c,d,e,f,g*>)] as [<*a,b*>] [<*c,d,e,f,g*>] and [P(<*l,m,n,o,p*>)] as [<*l,m,n*>] [<*o,p*>]. $MC_{P(<a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p>)}$ would call [<*a,b*>], [<*c,d,e,f,g*>], [<*h,i,j,k*>], [<*l,m,n*>], and [<*o,p*>]. Reordering would involve "deleting" the call to [<*o,p*>] in *MC* from its position after the call to [<*l,m,n*>] and "adding" it back to the *MC* in the position directly after the call to [<*a,b*>]. C[*MC*] would be 5.
>
> *Change 3*: Lastly, suppose we also want to be able to easily reorder the original sequence to produce a new sequence <*a,b,c,d,e,f,g,n,h,i,j,k,l,m,o,p*>. A good partition for this change would be [P(<*a,b,c,d,e,f,g*>)] as [<*a,b,c,d,e,f,g*>] and [P(<*l,m,n,o,p*>)] as [<*l,m*>] [<*o,p*>] with *n* in *MC*. $MC_{P(<a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p>)}$ would call [<*a,b,c,d,e,f,g*>], [<*h,i,j,k*>], [<*l,m*>], *n*, and [<*o,p*>]. Reordering would involve "deleting" *n* in *MC* from its position after the call to [<*l,m*>] and "adding" it back to the *MC* in the position directly before the call to [<*h,i,j,k*>]. C[*MC*] would be 5.
>
> *Total complexity across changes*: Let us consider each "good" partition from above across all three changes.
>
> *Partition 1* - [<*a,b*>] [<*c,d,e,f,g*>] [<*h,i,j,k*>] [<*l,m,n,o,p*>]:
>
> Change 1 - Complexity is 4 as discussed above.
> Change 2 - Delete <*o,p*> from [<*l,m,n,o,p*>] and add it at the end of [<*a,b*>]. Complexity is 7.
> Change 3 - Delete *n* from [<*l,m,n,o,p*>] and add it at the beginning of [<*h,i,j,k*>]. Complexity is 9.
> The total complexity across all three changes is 20.
>
> *Partition 2* - [<*a,b*>] [<*c,d,e,f,g*>] [<*h,i,j,k*>] [<*l,m,n*>] [<*o,p*>]:
>
> Change 1 - Delete call to [<*a,b*>] in MC. Add it directly after the call to [<*h,i,j,k*>]. Complexity is 5.
> Change 2 - Complexity is 5 as discussed above.
> Change 3 - Delete *n* from [<*l,m,n*>]. Add it at the beginning of [<*h,i,j,k*>]. Complexity is 7.
> The total complexity across all three changes is 17.
>
> *Partition 3* - [<*a,b,c,d,e,f,g*>] [<*h,i,j,k*>] [<*l,m*>] [*n* in *MC*] [<*o,p*>]:
>
> Change 1 - Delete <*a,b*> from [<*a,b,c,d,e,f,g*>] and add it at the beginning of [<*l,m*>]. Complexity is 8.

Change 2 - Delete <*o,p*> from [<*o,p*>]. Add it between *b* and *c* in [<*a,b,c,d,e,f,g*>]. Remove call to [<*o,p*>] in *MC*. Complexity is 14.
Change 3 - Complexity is 5 as discussed above.
The total complexity across all three changes is 27.

Partition 2 has the lowest complexity across all three likely changes. For change 3, this partition will involve the component [<*h,i,j,k*>] that contains an invariant subsequence in order to lower the change complexity.

Figure 5.42 Reordering that may involve a component containing an invariant subsequence.

Involvement of a component that contains a longest invariant subsequence may reduce the change complexity for reorders that result in empty or singleton components or in additions immediately before or after the invariant subsequence. For all other cases, involving a component that contains a longest invariant subsequence tends to increase the change complexity. The invariant subsequence pattern is a heuristically good way to partition control flow among components.

Process for Applying the Heuristic:

To efficiently apply the heuristic, the software designer needs a polynomial-time algorithm for locating the longest invariant subsequences across all expected or potential permutations of the original control flow sequence. The designer also needs a process for partitioning the remaining variant subsequences. Figure 5.43 contains a polynomial-time algorithm for locating the longest invariant subsequences, where *m* is the number of alternative sequences and *n* is the length of the required sequence. The computational complexity of the algorithm is $O(m*n)$.

As displayed in Figure 5.43, a bitmap represents *the immediately preceding relationship* between the task activations across the alternative control sequences. A one in the *j*-th bit position means that the *j*-th task activation in the required control flow sequence immediately precedes the ($j+1$)-th task activation in the required sequence across all projected permutations. The reader should note that the *n*-th task has no successor; hence, the bitmap contains a zero in the *n*-th bit.

The logic of the algorithm is to scan each alternative control flow sequence (the list of potential sequences from the requirements analysis) and to locate each task activation which is **not** followed by the same activation that succeeds it in the required sequence. When such a case is found, the bit for that task activation is set to zero in the bitmap. After all alternative sequences are scanned, the bitmap contains the information needed to locate the longest invariant sequences.

The algorithm then scans the bitmap by starting from the first bit (which corresponds to the first task activation in the required sequence) and moving across the bitmap to the last bit (which corresponds to the last task activation in the required sequence). As it scans, the algorithm records task activation subsequences associated with strings of 1's and 0's in the bitmap. Strings of 1's (plus the next immediate 0-bit) represent longest invariant subsequences. Strings of 0's (not including the 0 ending a string of 1's) represent longest variant subsequences.

Algorithmic Step **Complexity**

1. Initialize bitmap to [1|1|1|1|1|1|1|0] for the required sequence <*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*>. *O*(*n*)
2. Scan the *m* alternative control sequences to find task activations that do not precede the same task activation that they precede in the required sequence. *O*(*m***n*)

   For each of *m* alternative sequences
       For each of *n* task activations in an alternative sequence
           Does the task precede a different task from that which it precedes in the required task?
           If yes, set its position in the bitmap to zero.
       Endfor
   Endfor

   /* The bitmap is altered as follows while scanning the alternative sequence <*B*,*C*,*D*,*E*,*F*,*G*,*I*,*H*>.

   Evaluate *B*. Do not alter bitmap because task *B* still precedes task *C*.
   Evaluate *C*. Do not alter bitmap because task *C* still precedes task *D*.
   Evaluate *D*. Do not alter bitmap because task *D* still precedes task *E*.
   Evaluate *E*. Do not alter bitmap because task *E* still precedes task *F*.
   Evaluate *F*. Do not alter bitmap because task *F* still precedes task *G*.
   Evaluate *G*. Alter bitmap to [1|1|1|1|1|0|1|0] because task *G* does not precede task *H*.
   Evaluate *I*. Keep bitmap [1|1|1|1|1|0|1|0] because, although task *I* is not at the end of the alternative sequence, its bit is already zero.
   Evaluate *H*. Alter bitmap to [1|1|1|1|1|0|0|0] because task *H* does not precede task *I*.

   The bitmap after analyzing <*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*> is [1|1|1|1|1|0|0|0].
   Analyze <*B*,*C*,*D*,*I*,*E*,*F*,*G*,*H*> starting with [1|1|1|1|1|0|0|0], the current contents of the bitmap. The bitmap after analyzing <*B*,*C*,*D*,*I*,*E*,*F*,*G*,*H*> is [1|1|0|1|1|0|0|0] */
3. Scan the bitmap to locate the invariant and variant subsequences. *O*(*n*)

   The ones-strings, each of which ends with a zero, represent the longest invariant subsequences of <*B*,*C*,*D*> and <*E*,*F*,*G*>. The zero-string which represents a variant subsequence is <*H*,*I*>.
4. Determine the "heuristically good" partition of the original or required control flow sequence. *O*(*n*)
   - Place each longest invariant subsequence in a separate control flow component.
   - Place each subsequence which precedes or follows a longest invariant subsequence into a separate component.
   - The resulting "heuristically good" partition is <*B*,*C*,*D*><*E*,*F*,*G*><*H*,*I*>.
5. Add a master controller component to activate singleton tasks (resulting from subsequences each of which contains only one task activation) and the control flow components from step 4.

Figure 5.43 Polynomial-time process for determining a "heuristically good" partition of a control sequence, with application to the microwave oven Control-Electronics feedback loop.

The next step is to place each longest invariant subsequence in a separate component. The optimal method can be used to partition the remaining task activations. For simplicity, the designer can place each non-singleton variant subsequence in its own component. As discussed earlier, the master controller component contains singleton task activations as well as the loop logic. The result of using this heuristic to partition the Control-Electronics control flow for the microwave oven is shown in Figure 5.44. To view another application of this process, the reader should see [66,67]. The partitioning process described in Figure 5.43 is fully automatable. The input to the program are the required and alternative task sequences. The output is a list of the resulting components and the task or control flow component activations to be contained in each component.

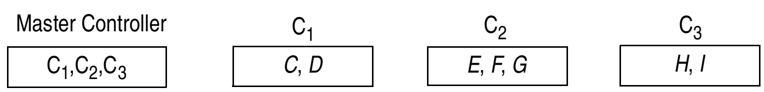


Figure 5.44 Heuristically good partition of the Control-Electronics control flow for the microwave oven.

The reader may recall the microwave oven Control-Electronics logic presented in Figure 5.20. The logic included a task *A* to initialize the timer as well as the loop logic to repeatedly execute the sequence $<B,C,D,E,F,G,H,I>$. To isolate the knowledge of how the feedback loop sequence may vary, the designer can hierarchically apply the concept of the Master Control architecture as shown in Figure 5.45. The first level master controller component, MC_1, activates task *A* and the loop logic. The loop logic repeatedly activates the second level master controller component, MC_2, which activates components C_1, C_2, and C_3, in the proper order before returning control to the loop logic in MC_1. The component C_1 activates the tasks *B*, *C*, and *D*, before returning control to MC_2. Likewise, components C_2 and C_3 activate their own task subsequences before returning control to MC_2. MC_1 does not know about the execution order of C_1, C_2, and C_3; and MC_2 has no knowledge of task *A* or the loop logic.

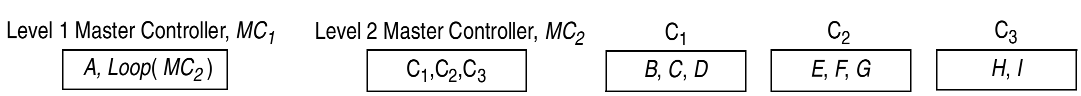


Figure 5.45 Hierarchical application of the Master Control architecture.

Section 5.5 initially discussed two different design strategies for simplifying changes to control flow. One way is to isolate the knowledge of the order in which tasks should be activated from the implementation of the tasks. Section 5.5 presented an optimal but complex process for partitioning a control flow sequence into control flow components that localize changes in the execution order of the tasks. Section 5.6 showed a polynomial-time process for achieving a heuristically good partition of task activations into control flow components. The examples in these sections applied the Master Control type of control flow architecture.

The other strategy for simplifying changes to control flow is to organize tasks (not just their activations) in a way that localizes changes in the order of their execution. With this approach, the implementation of each task activates the next task in the sequence. The designer can apply the optimal process to partition the tasks (not just the task activations) in a way that minimizes the complexity of changing the order in which they are executed. The sequence <*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*> represents the required order of executing the tasks, but the components of a partition contain the implementation of each task in the related subsequence. For instance, the implementation of the partition <*B*,*C*,*D*><*E*,*F*,*G*><*H*,*I*> consists of the following three components.

1. A component which contains the implementations of tasks *B*, *C*, and *D*.
2. A component which contains the implementations of tasks *E*, *F*, and *G*.
3. A component which contains the implementations of tasks *H* and *I*.

The change complexity is the size of the implementation components which must be modified to obtain an alternative control flow sequence. The reader should see Chapter 4 for a discussion of metrics to measure the size of a component.

The next section demonstrates how the research approach can be integrated with existing design approaches.

5.7 Integration with Existing Design Approaches

The research approach is about partitioning (decomposition and grouping) basic solution elements into reusable components. All non-monolithic software solutions consist of components that encapsulate data, operations, and control flow. The interactions between components involve the exchange of data or the flow of control. Therefore, the research approach is applicable to any design method that involves the determination of components. As mentioned earlier, the containers for components may be language-oriented structures such as class or module definitions as well as physically-oriented structures such as files, directories, disks, etc.

Partitioning is appropriate at any level of abstraction from the creation of system and subsystem components to the definition of low-level modules. Likewise, large-scale data and operations may be compositions of smaller-scale data elements and procedures that are reusable solution elements and that are partitioned into subcomponents within the context of a larger component. In other words, partitioning may involve abstraction through decomposition and hierarchical composition.

Lastly, the author would like the reader to note that the use of object-oriented components for the microwave oven software examples reflects the target programming environment for the empirical validation of the research approach. **As stated before, the research approach applies to any design method that involves the determination of components.**

The next chapter describes the process used to design the empirical studies for validating the research approach described in this chapter.

6 Validation of the Proposed Software Design Approach

Adoption of a new design approach requires verification and validation. Verification shows that the design method does what it is intended to do, and validation demonstrates that the method is useful. Chapter 5 demonstrated via construction and logical argument that the research approach partitions into separate components basic elements of a software solution that are impacted by the same changes. The author demonstrated the application of the method to the generation of a software architecture for a microwave oven. The next step is to validate whether or not the method helps other designers to determine an evolvable software architecture. Validation requires empirical research involving human subjects.

This chapter describes the process used to design the empirical studies for validating the research approach. The process integrates empirical techniques traditionally used in the fields of psychology and sociology along with issues and measures relevant to software engineering. From here on, the dissertation refers to the empirical studies used to validate the research approach as the research studies. The research studies consisted of two experiments. This chapter outlines the relevant research issues for the design of the research studies and explains why the ones designated for this thesis were chosen.

The organization of this chapter is as listed below.

- Section 6.1 describes the research issue space for the design of the research studies.
- Section 6.2 discusses the direct evaluation of the research approach through the assessment of changeability.
- Section 6.3 discusses the indirect evaluation of the research approach through the assessment of structural complexity.
- Section 6.4 explains the evaluation of design effort conducted as part of the research studies.
- Section 6.5 outlines a generic process for designing an empirical test of a software design approach.

For a comprehensive discussion of issues in the evaluation of software engineering methods and tools, the reader should see [89].

6.1 Empirical Research Issue Space

Testing the effectiveness of a software design method is a challenging task. The challenge is two-fold: determining the effect of the method on the resulting software artifacts or products as well as on the software development process. Software design, as well as implementation and maintenance, requires substantial manual effort. An effective design method enables the designer to create a blueprint for a software system that will achieve functional correctness and will be easy for the maintainer to change with minimal error. Testing the impact of a software design method on the resulting software product features and development process (dependent variables) requires an experiment involving human subjects who will perform a representative design task.

Figure 6.1 outlines the factors that characterize the research issue space for the design of the research studies. The three primary areas that characterize this space are:

- Design approach used by the software designers.
- Measures used to determine the effectiveness of the design approach.
- Experimental issues.

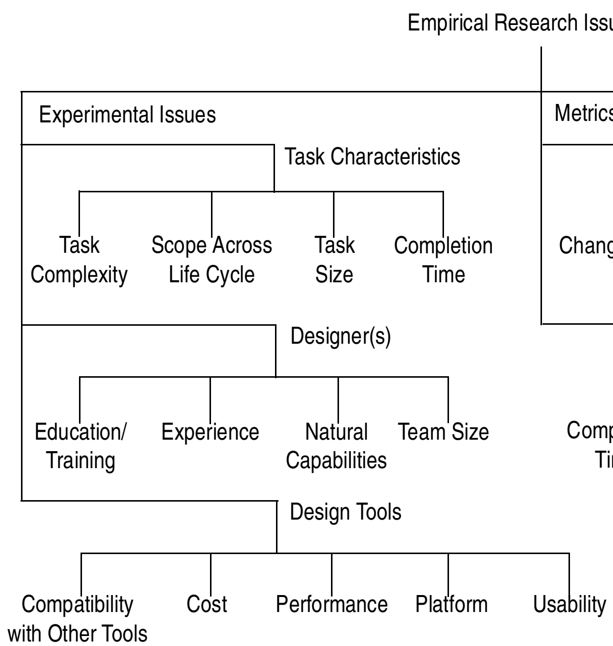


Figure 6.1 Research issue space for the design of empirical studies.

The primary independent variable to be tested is the **software design approach** (or research approach). This approach has two major facets: (1) a rationale for thinking about change and reuse, and (2) a method for applying the design rationale to determine the basic components in a software system. The approach is applicable to various design approaches, such as structured or object-oriented design. For object-oriented styles of design, the output is a partition of data and operations into files, classes, methods, and method calls. The basic components are class definitions and files. The goal of the research studies is to determine the effectiveness of both the rationale and the method parts of the design approach. The rationale is a prerequisite for the method.

Table 6.1 Potential values for the design approach factors.

Design Rationale	Design Method
Designer's Own Rationale	Designer's Own Design Method
Proposed Rationale	Proposed Design Method (Partitioning)

Table 6.2 Independent variable varied across treatment groups.

	Group 1 (Control)	Group 2 (Experimental)	Group 3 (Experimental)	(not applicable)
Design Rationale	Base Design Rationale	Proposed Rationale	Proposed Rationale	Base Design Rationale
Design Method	Base Design Method	Base Design Method	Proposed Design Method	Proposed Design Method

As shown in Table 6.1, there are two values for each of the two aspects of a software design approach to be tested. The values for the research approach are the "Proposed Rationale" and the "Proposed Design Method." There are 2x2 or 4 potential combinations of these values as shown in Table 6.2. The combination of "Designer's Own Rationale" and "Proposed Design Method" is not relevant for the research studies because application of the proposed design method (part of the research approach) requires use of the proposed rationale. The other three combinations define the three values for the independent variable being tested, one for each of the three treatment groups, as enumerated in Figure 6.2.

1. The *basic* lecture on the use of abstraction, decomposition, and inheritance to determine the classes for an object-oriented design. The *control group* (CG) in the research studies received this type of treatment.
2. The basic lecture plus a discussion of the meaning and importance of designing for change and reuse (e.g. the development of a family of software products), henceforth called the *basic+rationale* lecture. The first treatment group designated the *rationale group* (RG) in the research studies received this type of treatment.
3. The basic+rationale lecture plus instruction in using the experimental design method for determining evolvable software architectures composed of reusable components, henceforth called the *basic+rationale+method* lecture. The second treatment group designated the *rationale+method group* (RMG) in the research studies received this type of treatment.

Figure 6.2 Detailed description of the treatment groups.

In addition to the software design approach, other important independent variables that can affect the types of designs produced by software designers follow.

- Design task
- Designer's knowledge and skill
- Tools used by the designer

In order to test the effects of the research approach independently, the experimental design must control variance in the above factors across the treatment groups. Control involves eliminating the effect of an independent variable or equalizing its effect across all subjects. Unless mentioned otherwise, the research studies apply the latter type of control.

The task complexity, scope across the life cycle, task size, and completion time characterize the task to be performed by the subjects in the research studies. Table 6.3 lists feasible values for each factor. One factor is the task complexity. New tasks (those not done before) may be more difficult for the designer. Likewise, the use of components which execute concurrently or which are distributed across multiple processors can increase the complexity of a design. Most importantly, the relationships between the components in a software architecture impact the design complexity. Another factor is the scope of the task. Depending on the design approach to be tested, the designer may need to analyze the behavior of the system before designing the software, a situation in which the experimental task would include behavioral analysis as well as design. With sufficient time, the experimenter may want to test the impact of the resulting designs on implementation and maintenance. Representative task sizes and completion times appear in the table.

Table 6.3 Experimental Issues - Potential values for the task factors.

Task Complexity	Scope Across Life Cycle	Task Size	Completion Time
Novelty	Behavioral Analysis (Including Analysis of Existing System)	1-10 Components	Class Assignment (Hours)
Concurrency	Design (Including Redesign of Existing System)	11-50 Components	Homework Assignment (Days)
Distribution	Implementation, Integration, & Test	51-100 Components	Project Assignment (Weeks)
Functional Interrelationships	Maintenance and Product Evolution	100+ Components	

A designer's education/training, experience, and natural capabilities also affect the quality of the designs that he/she creates. Values for each of these factors appear in Table 6.4. A survey of each subject's education/

training, experience, and college major occurred before the start of each research study. Assessing the natural capabilities of individual software designers requires expertise in psychological testing that goes beyond the scope of the research studies. Based on the results of the pre-experiment surveys, the research studies used blocking and randomization to control the individual differences between subjects. The other factor to be considered is the team size. For the research studies, each subject completed the assigned task independently (a team size of one).

Table 6.4 Experimental Issues - Potential values for factors related to the designer.

Education/Training	Experience	Natural Capabilities	Team Size
Programming Courses	Programming in the Small	Abstraction	1-2
Software Engineering Courses	Programming in the Large	Organization	3-10
Software Architecture Courses	Across Life Cycle	Attention to Detail	10+
Training in Design Method	Different Types of Components	Different Application Domains	
Domain Specific Courses	Different Projects in Application Domain		

As shown in Table 6.5, there are multiple factors that determine the impact that software design tools have on a designer's work. Factors such as compatibility with other tools, performance, platform availability and variance across platforms, as well as usability directly affect the ways in which a designer uses a design tool. Cost is a factor in selecting a design tool and in determining the number of licenses for host machines, the amount of design tool training, and the frequency of upgrades. Design tool quality and availability along with sufficient training in the use of a tool can affect the designer's ability to produce good designs. The research studies eliminated the impact of software design tools by having the subjects not use them.

Table 6.5 Experimental Issues - Potential values for the design tool factors.

Compatibility With Other Tools	Cost	Performance	Platform(s)	Usability
Integration of Artifacts	Initial Purchase	Error Proneness	Availability on Preferred Platform	Ease of Use
Integration of Methods	Licensing	Design Method Support	Variance in Functionality and Performance Across Different Platforms	Understandability
	Training	Response Time	Portability of Artifacts Produced on Different Platforms	Help Facility
	Upgrades			

The research studies focused on the measurement of changeability via comparisons of product metrics whose values were assessed for the designs completed by the subjects. As shown in Figure 6.1, the product metrics include both changeability as well as structural complexity metrics. The research studies evaluated the impact of the research approach on the design of evolvable software via direct and indirect evaluation as described in Figure 6.3. The next two subsections discuss these evaluations in detail.

> *A direct evaluation, through assessing changeability.* The objective was to observe any effects that the research approach might have on ease of change.
>
> *An indirect evaluation, through assessing structural complexity.* The objective was to determine any correlation between software complexity and software evolvability.This was done by comparing the variance in the degree to which the resulting designs enhance ease of change along with variance in the complexity of these designs.

Figure 6.3 Direct and indirect evaluation of the effects of the research approach.

Additionally, the research studies collected process information about the design effort itself, through measurement of the design time and the types/numbers of errors detected during design review.

6.2 Direct Evaluation Through Assessment of Changeability

Direct assessment of changeability is a new technique introduced by the research studies. The idea is to determine the difficulty of changing the designs resulting from the research studies. Previous approaches to changeability assessment were primarily based on the assessment of maintenance effort after the software had been implemented. With the new process, the evaluator identifies the parts of a design which must be modified as well as the parts which can be used without modification to satisfy new requirements (those specified during requirements or behavioral analysis as the expected or feasible evolution of the required software behavior). The evaluator uses a standard method for “sizing” these parts. The evaluator for the research studies used the method discussed in Section 4.5.

Two resulting ratios measure the evolvability of a software design as denoted in Figure 6.4.

> *Design’s change impact*: a ratio of the size of the design parts that require modification and the size of the entire design.
>
> *Design’s degree of reusability*: a ratio of the size of the design parts which can be reused without modification and the size of the entire design.

Figure 6.4 Ratios for measuring the evolvability of a software design.

The goal is for a design to have a low change impact and a high degree of reusability.

In the case of the research studies, the sum of these two ratios for each design is 100%. This is a theoretical estimate. In practice, some of the parts to modified may actually be replaced by entirely new pieces of design. Since the exact size of a new design piece is not known before its creation, the experimenter chose to use the size of the related old design piece as an estimate. This approach differs from more typical assessments of software reusability which distinguish components reused without modification, components reused with modification, and new components. The reason for the difference is that typical assessments are generally made after the software is implemented, used, and maintained.

Another feature of this new approach to evaluating software designs is the opportunity for *granular assessment*. The definition of "part" to be modified and "part" to be reused without modification can be at whatever level of abstraction is useful to the experimenter. For both of the research studies, the experimenter assessed changeability at the levels of routine (methods) and components (classes). The second research study also included assessment of changeability at the big component (file) level. As the reader can note later, goodness at one level of granularity does not necessarily guarantee goodness at another level.

6.3 Indirect Evaluation Through Assessment of Structural Complexity

The experimenter was interested in assessing correlation between the structural complexity of the designs produced by the subjects and the evolvability of these designs. A strong correlation between a particular structural complexity measure and the changeability measure may identify an *evolvability predictor*. An evolvability predictor is a structural feature of a design which is measurable and correlated to the changeability of the design. A poor value for an evolvability predictor may signal the need for redesign.

The first research study included the collection of structural complexity data at the routine, component, and system levels as shown in Table 6.6. The definitions for the detailed structural complexity measures appear in Section 4.5.

Table 6.6 Structural complexity measures for the first research study.

Structural Complexity Level	Type of Structural Complexity Measure	Detailed Type of Structural Complexity Measure
Routine	Size	Number of routine attributes for a particular routine (number of parameters + number of local variables)

Structural Complexity Level	Type of Structural Complexity Measure	Detailed Type of Structural Complexity Measure
“	Size	Size of a particular routine (number of routine attributes + size of routine logic as measured using the method described in Section 4.5)
“	Coupling	Number of calls to other routines contained within a particular routine
“	Control Flow	V(G) of a particular routine
Component	Size	Number of component level attributes for a particular component (e.g. number of variables defined within a class but external to the methods which are defined as part of the class)
“	Size	Number of routines contained within a particular component (e.g. number of methods defined as part of a class)
“	Size	Size of a particular component [(number of component variables) + (sum of the sizes of the routines contained within the component)]
“	Coupling	Fan-in for a particular component (number of other components which activate a routine which is part of the component)
“	Coupling	Fan-out for a particular component (number of other components whose routines are activated by the component)
“	Coupling	Number of calls to external routines from a routine defined within the component
“	Control Flow	V(G) of a particular component
System	Size	Number of components in the entire software system
“	Size	Size of the entire software system [(sum of the size of each component in the system) + (sum of the size of any files which contain data type or data definitions that are external to the components)]
“	Control Flow	V(G) for the entire software system

Since the second research study involved the redesign of parts of an existing software system, the original design fixed many of the structural complexity features. Therefore, the experimenter did not assess structural complexity for this study.

6.4 Evaluation of Design Effort

The research studies evaluated the effects of the research approach on the design process itself by having the subjects measure and record time and error data. Table 6.7 lists the types of activities that were timed by the subjects in the first research study. Table 6.8 shows the activities that were timed by the subjects in the second research study. The reader should recall that **RG** and **RMG** are abbreviations for the Rationale Group and Rationale+Method Group, respectively.

Table 6.7 Activities timed by the subjects in the first research study.

Activity Type	Detailed Activity for Each Activity Type	Treatment Group
Requirements Analysis	Functional requirements -- Read and understand the software requirements.	all groups
"	Functional changes -- Read and understand functional and data changes.	RG and RMG
"	Control flow changes -- Read and understand control flow changes.	RG and RMG
Design	Data and operations -- Identify the basic data and operations.	RMG
"	Reuse analysis -- Analyze operations for reusability and decompose as needed.	RMG
"	Data dependencies -- Identify data change dependencies.	RMG
"	Other dependencies -- Identify other change dependencies.	RMG
"	Change sets -- Formulate change sets.	RMG
"	Components -- Determine the objects (classes), method, and data.	all groups
"	Component interfaces -- Specify the parameters for the methods.	all groups
"	Component behavior -- Specify via pseudo-code the basic logic as well as the flow of control within and between objects.	all groups
"	Reuse analysis-- Analyze components for reusability and modify the design as needed.	RG
"	Change analysis-- Analyze the design for changeability and alter the design as needed.	RG
Design Review	Components -- Verify that the components support all of the requirements correctly.	all groups
"	Interfaces -- Verify that the component interfaces are complete, correct, and consistent.	all groups
"	Component Behavior -- Verify that the logic and flow of control within and between components is correct and complete. Remember that the software system must have a beginning and a graceful way to end.	all groups

Table 6.8 Activities timed by the subjects in the second research study.

Project Assignment	Detailed Activity for Each Project Assignment	Treatment Group
Redesign of the Recoverable Virtual Memory (RVM) Facility	Review the Coda Client requirements specification.	all groups

Project Assignment	Detailed Activity for Each Project Assignment	Treatment Group
“	Think about the current design of the Coda client RVM facility.	“
“	Determine a new design for the RVM facility.	“
“	Document the new design for the RVM facility.	“
“	Document the rationale for the new design of the RVM facility.	“
“	Review deliverables.	“
Redesign of the Kernel-Venus Interface	Review the Coda Client requirements specification.	“
“	Think about the current design of the Kernel-Venus Interface.	“
“	Determine a new design for the Kernel-Venus Interface.	“
“	Document the new design for the Kernel-Venus Interface.	“
“	Document the rationale for the new design of the Kernel-Venus Interface.	“
“	Review deliverables.	“
Evaluation of the New RVM Design	Evaluate the spatial performance of the RVM design.	“
“	Evaluate the temporal performance of the RVM design.	“
“	Evaluate the impact of change on the RVM design.	“
“	Review deliverables.	“
Evaluation of the New Kernel-Venus Interface Design	Read and think about assignment.	“
“	Evaluate the Kernel-Venus interface (Deliverable 1: Questions 1-4).	“
“	Evaluate the Kernel-Venus interface (Deliverable 1: Question 5).	“
“	Evaluate the Kernel-Venus organization for ease of change.	“
“	Review deliverables.	“

The subjects received instructions to count the errors that they detected when reviewing their designs or design evaluations. In general, errors were of type omission (missing specification of solution element or element required for the task) or of type commission (incorrect specification of solution element or element required for the task). All of the treatment groups for each study were to check for the same types of errors.

For the first research study, the types of errors to be identified were those shown in Table 6.9. Those to be detected in the second research study are shown in Table 6.10.

Table 6.9 Types of errors identified by the subjects in the first research study.

Type of Errors	Detailed Types of Errors
Omission	Classes -- Missing object definitions (classes)
"	Methods -- Missing method definitions
"	Data -- Missing data definitions
"	Interfaces -- Missing parameters in the method interfaces
"	Control Flow -- Missing method calls
Commission	Classes -- Class definitions that are not needed or incorrectly named. Include inconsistencies between the definition of an object and the declaration of variables to be instances of the objects.
"	Methods -- Incorrect method definitions or declarations
"	Data -- Incorrectly defined data (e.g. wrong data type) in a class or method definition
"	Interfaces -- Incorrectly specified parameters
"	Control Flow -- Method calls which are out of order or not needed

Table 6.10 Types of errors identified by the subjects in the second research study.

Project Assignment	Type of Errors	Detailed Types of Errors
Redesign of the RVM Facility & Redesign of the Kernel-Venus Interface	**Omission**	Missing deliverable in the deliverables packet
"	"	Missing class definition in the design
"	"	Missing method or function definition in the design
"	"	Missing data structure or user-defined type in the design
"	"	Missing method call
"	"	Missing English description
"	"	Other
"	**Commission**	Incorrect deliverable in the deliverables packet
"	"	Incorrect class in a deliverable

Project Assignment	Type of Errors	Detailed Types of Errors
“	“	Incorrect method or function in a deliverable
“	“	Incorrect data structure or user-defined type in a deliverable
“	“	Incorrect method call
“	“	Incorrect English description
“	“	Other
Evaluation of the New RVM Design	**Omission**	Missing deliverable in the deliverables packet
“	“	Missing calculation in the spatial performance evaluation
“	“	Missing calculation in the temporal performance evaluation
“	“	Missing indication of class that would be affected by a change
“	“	Missing indication of method or function that would be affected by a change
“	“	Missing indication of file that would be affected by a change
“	“	Other
“	**Commission**	Incorrect deliverable in the deliverables packet
“	“	Incorrect calculation in the spatial performance evaluation
“	“	Incorrect calculation in the temporal performance evaluation
“	“	Incorrect indication of class that would be affected by a change
“	“	Incorrect estimation of size of class
“	“	Incorrect indication of method or function that would be affected by a change
“	“	Incorrect estimation of size of method or function
“	“	Incorrect indication of file that would be affected by a change
“	“	Incorrect estimation of size of file
“	“	Other
Evaluation of New Kernel-Venus Interface	**Omission**	Missing deliverable in the deliverables packet
“	“	Missing data structure in Deliverable 1: Questions 1-4 (For more information, see Task A in Appendix Q.)
“	“	Missing explanation in Deliverable 1: Questions 1-4 (For more information, see Task A in Appendix Q.)

Project Assignment	Type of Errors	Detailed Types of Errors
“	“	Missing indication of program element that would be affected by a change
“	“	Missing indication of class that would be affected by a change
“	“	Missing indication of method or function that would be affected by a change
“	“	Missing indication of file that would be affected by a change
“	“	Other
“	**Commission**	Incorrect deliverable in the deliverables packet
“	“	Incorrect data structure listed in Deliverable 1: Questions 1-4 (For more information, see Task A in Appendix Q.)
“	“	Incorrect explanation in Deliverable 1: Questions 1-4 (For more information, see Task A in Appendix Q.)
“	“	Incorrect indication of program element that would be affected by a change
“	“	Incorrect estimation of size of program element
“	“	Incorrect indication of class that would be affected by a change
“	“	Incorrect estimation of size of class
“	“	Incorrect indication of method or function that would be affected by a change
“	“	Incorrect estimation of size of method or function
“	“	Incorrect indication of file that would be affected by a change
“	“	Incorrect estimation of size of file
“	“	Other

For the first research study, the experimenter evaluated the designs produced by the subjects with respect to error. The experimenter conducted her own extensive evaluation of error after observing that the subjects reported very few errors. The experimenter limited her detailed evaluation to the software designs produced in the first study. These designs, having been constructed from “scratch”, exhibited more architectural variety and demonstrated a wider variety of error types. Table 6.11 shows the types of errors that the experimenter identified in the designs produced by the subjects in the first research study.

Table 6.11 Types of errors detected by the experimenter for the first research study.

Type of Errors	Detailed Types of Errors
Omission	Missing component and routine names
"	Missing component variables
"	Missing required operations
"	Missing parameters in routine interface
"	Missing parameters in routine call
"	Missing local variables
"	Missing routine calls
"	Missing status/error checks
"	Missing status/error codes
Commission	Incorrect component and routine names
"	Incorrect component variables
"	Incorrect required operations
"	Incorrect parameters in routine interface
"	Incorrect parameters in routine call
"	Incorrect local variables
"	Incorrect routine calls
"	Incorrect status/error checks
"	Incorrect status/error codes

For additional information about measuring the software development process, the reader may refer to [47,54].

6.5 Process for Designing an Empirical Test of a Software Design Approach

A systematic and step-wise process can simplify the design of an empirical study to test a software design approach. The process should guide the researcher in the identification of the experimental design features and decisions to be made about these factors. The experimental design features considered for the research studies are those shown in Figure 6.5.

- Purpose of the experiment: Hypotheses about the software design approach.
- Independent variables: Factors which can affect the dependent variables.
- Dependent variables: Software process or product variables whose measurement will indicate the goodness of the software design approach.
- Condition variables: Independent variables whose effects are to be determined.
- Nuisance variables: Independent variables whose effects are to be controlled.
- Subjects: Software designers who perform the assigned tasks after receiving instruction about software design (a treatment).
- Tasks: A prescribed software development task that focuses on software design.
- Statistical procedures for verifying the experimental results.

Figure 6.5 Experimental design factors for the research studies.

Figure 6.6 outlines the process that was used to design the research studies. This process is generic and applicable to the design of other empirical studies for validating a software design approach. Chapter 7 discusses the application of this process for the research studies.

1. Identify the observable features and measurable impacts of the research approach.
   a. What software qualities indicate goodness with respect to the purpose of the research approach?
   b. What software artifacts (products) are the output of the research approach?
   c. What measurable features of these artifacts indicate their goodness?
   d. What software process features are measurable and interesting?
   e. What is the desirable impact that the research approach should have on the product and process features?
2. Determine and specify the product and process factors to be measured (dependent variables).
   a. Which measurable features explored in step one will help to answer the questions that the researcher has about the research approach?

      For ideas, see the Metrics sub-tree of the Empirical Design Space in Figure 6.1.
   b. What artifacts need to be produced or process data collected to determine values for the target measures?
3. Identify the observable factors which could affect the dependent variables (independent variables).
   a. What aspects of the research approach are to be evaluated?

      For ideas, see the Research Approach sub-tree of the Empirical Design Space in Figure 6.1.
   b. Do the dependent variables identified in Step 2 adequately evaluate the impact of the research approach?
   c. What characteristics of the software designer could affect the dependent variables?

      For ideas, see the Designer sub-tree of the Empirical Design Space in Figure 6.1.
   d. What characteristics of the software designer's work environment could affect the dependent variables?

      For ideas, see the Design Tool sub-tree of the Empirical Design Space in Figure 6.1.

4. Classify the independent variables as condition or nuisance variables.
   a. Those variables whose effect is to be evaluated are condition variables.
   b. Those variables whose effect is to be controlled are nuisance variables.
5. Analyze the condition variables and specify the treatment groups.
   a. Which combinations of values for the condition variables make sense?
   b. How many treatment groups are needed so that each group receives a unique treatment as represented by one of the feasible combinations of values for the condition variables.
6. Analyze the nuisance variables and specify a way to control the potential effects of each nuisance variable.
   a. Can the potential effects of a specific nuisance variable be eliminated completely?
   b. How could the effects of the nuisance variables be equalized across the treatment groups?
      (1) Would random assignment of the subjects to treatment groups equalize these effects?
      (2) Do some characteristics of the subjects occur in blocks (in other words, there is a finite and manageable number of categories for a characteristic)?
      (3) Is it feasible and practical to first assign the subjects to blocks and then randomly assign an equal number of subjects from each block to each of the treatment groups?
      (4) Could each subject receive all of the different treatments in a random order without the interference of learning (within-subjects design)?
      (5) Which assignment of subjects to treatments best eliminates the effects of the nuisance variables (e.g. random assignment of subjects, random assignment of subjects with blocking, or a within-subjects design)?
      (6) Can the nuisance variables be held constant for all subjects?
      (7) Does it matter if the subjects interact with subjects within their treatment groups or with subjects in other treatment groups? If so, how can the experimenter control any potential for interaction?
7. Identify the group of subjects.
   a. What is the general population of subjects to which the hypotheses about the research approach are applicable?
   b. Is it possible to find a sample of subjects who represent the target population?
   c. Are there ethical, legal, and institutional procedures that must be followed to conduct an experiment using human subjects?
      (1) Are there permissions that must be received from the institution in which the experiment is to be conducted?
      (2) How might one recruit volunteers from a potential group of subjects?
      (3) How should the subjects formally consent to participate in a research study?
      (4) If the subjects receive compensation for their participation, how might one record the receipt of this compensation?
8. Specify the detailed task to be performed by the subjects.
   a. What resources do the subjects need to perform the task? For instance, use of the research approach to create a software design requires a specification of software requirements and an

analysis of the expected or feasible software evolution.

   b. Where will the subjects perform the task?

   c. What are the complexity, scope, and size of the task?

   d. What is the estimated time to complete the task?

   e. If a within-subjects design is used, can the same task be performed for each treatment? If not, how will different tasks be equalized with respect to factors such as difficulty?

9. Determine the statistics needed to verify the experimental results and check that the experimental design will generate the necessary data.

Figure 6.6 Process for designing the research studies.

7 Empirical Research Studies and Results

The goal for the two research studies was to determine if the subjects who were trained in the use of the research approach were able and more likely to produce better designs than those who were not trained in this approach. The measure of goodness, change impact, was the same for each study. Though controlled, the characteristics of the subjects and the task were purposely different for each study. The objective was to test hypotheses stated in Chapter 1 regarding the research approach.

This chapter outlines for each research study the experimental design factors, summary statistics, analyses of variance, hypothesis tests, and other statistical calculations. This chapter includes the sections below.

- Section 7.1 outlines the experimental design factors for the research studies.
- Section 7.2 overviews the most important experimental results and conclusions.
- Section 7.3 presents the change impact results for the first experiment.
- Section 7.4 discusses the structural complexity results for the first experiment.
- Section 7.5 shows the design effort results for the first experiment.
- Section 7.6 presents the change impact results for the second experiment.
- Section 7.7 discusses design effort results for the second experiment.
- Section 7.8 presents final observations and conclusions about the empirical studies.
- Section 7.9 concludes with anecdotal information about the research studies.

7.1 Experimental Design Factors

This section steps through the process described in Chapter 6. It outlines the experimental design factors for the research studies. The factors are the same for both experiments except when specifically noted.

Step 1: Identify the observable features and measurable impacts of the research approach.

Table 7.1 explains the empirical design decision that addresses each process question in step one.

Table 7.1 Step one in the design of an empirical study.

Process Question	Empirical Design Decision
What software qualities indicate goodness with respect to the purpose of the research approach?	Target software qualities: • Evolvability • Structural complexity features which may be evolvability predictors
What software artifacts (products) are the output of the research approach?	Specification of a software architecture; namely: • Components • Behavior of the components • Interactions between the components via the interfaces to the components

Process Question	Empirical Design Decision
What measurable features of these artifacts indicate their goodness?	As described in Figure 6.4 of Section 6.2: • Change impact • Degree of reusability ratios for measuring the evolvability of a software design Table 6.6 in Section 6.3 lists the structural complexity metrics used for measuring size, coupling, and control flow at the levels of routine, component, and system. The experimenter assessed the structural complexity of the resulting designs only for the first experiment, as explained earlier.
What software process features are measurable and interesting?	Target process features: • Time spent on each type of task activity • Number and type of errors detected during review The following tables in Section 6.4 provide details regarding the measurement of time and error. • Table 6.7 lists the activities timed by the subjects in the first experiment. • Table 6.8 lists the activities timed by the subjects in the second experiment. • Table 6.9 lists the types of errors detected and counted by the subjects in the first study. • Table 6.10 lists the types of errors detected and counted by the subjects in the second study. • Table 6.11 lists the types of errors detected and counted by the experimenter for the first study.
What is the desirable impact that the research approach should have on the product and process features?	The goal is for use of the research approach to result in: • Software designs with lower change impact and higher degrees of reusability. • Less time needed for product evolution. • Lower potential for error during product evolution. The research studies use the impact of change to gauge the impact on evolution time and potential error during product evolution. Note 7.1 explains the relationship between the impact of change and the impact on evolution time and potential error during product evolution.

Note 7.1 The rationale for this estimation is the fact that the size of the impacted system directly affects the expected time and potential error involved with changing the software. A larger impact means that more of the system is involved in change and therefore that the time needed for change as well as the potential for error would be higher.

The research approach does not specifically attempt to reduce the time required for design and does not specify the design review process. Therefore, the experimenter did not formulate hypotheses regarding time for design-related activities or error detection during design review. For exploratory purposes only, the experimenter directed the subjects to record time spent on the activities of the assigned task as well as the number and type of errors detected during review. It is logical to expect that designers who are less experienced in using the research approach would take more time than those who are experienced in its use to determine an appropriate set of software components. It may also be probable that designers experienced in using the research approach may require less time for generating a software architecture than they would require with the use of another design method. The research approach provides step-wise directions and guidelines for design decisions that are currently not covered well by popular design methods.

Step 2: Determine and specify the product and process factors to be measured (dependent variables).

Table 7.2 explains the empirical design decision that addresses each process question in step two.

Table 7.2 Step two in the design of an empirical study.

Process Question	Empirical Design Decision
Which measurable features explored in step one will help to answer the questions that the researcher has about the research approach?	As discussed in Section 6.2, both experiments involved direct evaluation through assessment of changeability. As presented in Section 6.3, only the first study included indirect evaluation via assessment of structural complexity.
What artifacts need to be produced or process data collected to determine values for the target measures?	Both experiments required the subjects to produce a design specifying a software architecture. • For the first study, the design must satisfy the requirements for the creation of a software system from scratch. • For the second study, the design must satisfy the original requirements plus new design objectives (redesign of part of an existing software system). The process data includes the time and error data discussed in Step 1.

Step 3: Identify the observable factors which could affect the dependent variables (independent variables).

Table 7.3 explains the empirical design decision that addresses each process question in step three.

Table 7.3 Step three in the design of an empirical study.

Process Question	Empirical Design Decision
What aspects of the research approach are to be evaluated?	There are two primary features of the research approach. 1. Design rationale 2. Design approach
Do the dependent variables identified in Step 2 adequately evaluate the impact of the research approach?	Yes. • Evolvability is measurable through assessment of changeability. • Evolvability predictors are structural complexity measures which are correlated to changeability.
What characteristics of the software designer could affect the dependent variables?	Characteristics of the software designer are the following. • Education/training • Experience • Natural capabilities • Team size
What characteristics of the software designer's work environment could affect the dependent variables?	Characteristics of the software design work environment are the following. • Software design tools • Environmental conditions which affect the designer's ability to create software designs (e.g. lighting, desk space, and heating/cooling)

Step 4: Classify the independent variables as condition or nuisance variables.

Table 7.4 lists the condition or nuisance variables identified in step four.

Table 7.4 Step four in the design of an empirical study.

Condition Variables	Nuisance Variables
Design rationale	Designer's education/training, experience, and natural capabilities as well as team size
Design approach	Software design tools
	Environmental conditions which affect the designer's ability to create software designs (e.g. lighting, desk space, and heating/cooling)

Step 5: Analyze the condition variables and specify the treatment groups.

Table 7.5 explains the empirical design decision that addresses each process question in step five.

Table 7.5 Step five in the design of an empirical study.

Process Question	Empirical Design Decision
Which combinations of values for the condition variables make sense?	See Table 6.1 in Section 6.1.
How many treatment groups are needed so that each group receives a unique treatment as represented by one of the feasible combinations of values for the condition variables.	3 treatment groups See Table 6.2 in Section 6.1 for a list of the treatment groups and Figure 6.2 for a detailed description of each group.

Step 6: Analyze the nuisance variables and specify a way to control the potential effects of each nuisance variable.

Table 7.6 explains the empirical design decision that addresses each process question in step six.

Table 7.6 Step six in the design of an empirical study.

Process Question	Empirical Design Decision
Can the potential effects of a specific nuisance variable be eliminated completely?	Yes. The experimenter can eliminate the effects of the design tools. The subjects in the research studies did not use software design tools. They did use word processors for documenting their designs, but the experimenter did not consider the type of word processor to be a nuisance variable.
How could the effects of the nuisance variables be equalized across the treatment groups? Would random assignment of the subjects to treatment groups equalize these effects?	 Yes. Assignment to treatment groups was random.

Process Question	Empirical Design Decision
Do some characteristics of the subjects occur in blocks (in other words, there are a finite and manageable number of categories for a characteristic)?	Characteristics which occur in blocks: • Major or academic program of study • Education/training • Experience • Availability to attend a “treatment” lecture (day/time)
Is it feasible and practical to first assign the subjects to blocks and then randomly assign an equal number of subjects from each block to each of the treatment groups?	According to the pre-experiment surveys completed by the subjects for the first experiment: 1. Most of the participants were engineering or science students. Some participants were students pursuing degrees involving the technology-oriented aspects of theater production systems. 2. All of the participants had some experience with programming in the small (e.g. programs of less than 2000 lines of executable code). 3. A few participants had taken a high school programming course; but most were enrolled in their first programming course. In the first experiment, blocking was according to major or academic program and day/time availability for the treatment lecture. The treatment lectures were held outside of the class from which the participants were recruited. The body of subjects for the second experiment consisted of the following two primary groups. 1. Undergraduate computer science and engineering students 2. Graduate students with undergraduate degrees in engineering/science and education/experience in computing According to the pre-experiment surveys completed by the subjects for the second experiment, the levels of education/training and experience were similar within each of the two groups. In the second experiment, blocking was also according to academic program. Day/time availability was not a factor because the treatment lectures were conducted during the time of the class from which the participants were recruited. The teaching assistant for the class identified groups of students that typically worked together on class projects. Whenever possible, the experimenter also blocked with respect to student working groups.
Could each subject receive all of the different treatments in a random order without the interference of learning (within-subjects design)?	No. Random order is not feasible. The effects of learning are theoretically controllable by having each subject respond independently to each treatment with treatments administered in the following order: 1. Basic lecture, 2. Basic+rationale lecture, 3. Basic+rationale+method lecture. Time constraints and the difficulty of equalizing the task performed by the subject across different treatments precluded a within-subjects design.
Which assignment of subjects to treatments best eliminates the effects of the nuisance variables (e.g. random assignment of subjects, random assignment of subjects with blocking, or a within-subjects design)?	As discussed above, random assignment of subjects with blocking according to academic program was the practical choice for both experiments. The first experiment also included blocking according to day/time availability for the treatment lecture. In the second experiment, subjects who most potentially would interact with each other received the same treatment when possible.

Process Question	Empirical Design Decision
Can the environment variables which are a nuisance be held constant for all subjects?	Due to time/space limitations, it was not practical to have the subjects complete the assigned task in one location with identical environmental conditions. The subjects were to perform the software design task in environments similar to those in which they would complete a typical class assignment or project. The expectation was that the differences in environment across subjects would vary randomly. The effects of environment on software design are beyond the scope of the research studies.
Does it matter if the subjects interact with subjects within their treatment groups or with subjects in other treatment groups? If so, how can the experimenter control any potential for interaction?	The subjects were to perform their assigned task independently. The experimenter used several approaches to encourage the subjects to work on their design task independently. • The instructors for the courses from which the subjects were recruited awarded the participants bonus points for completing the required task. The experimenter assigned bonus points according to the completeness, quality, and independence of the resulting designs. • The experimenter carefully explained that the award of bonus points was not based on competition. Each design could potentially earn the maximum bonus points. • The experimenter explained to the subjects the ethical responsibilities of participants in research studies. • The experimenter offered help sessions to answer questions that the subjects had about the assigned task. • For the second research study, the experimenter assigned students who typically worked together on class projects to the same treatment group whenever possible.

Step 7: Identify the group of subjects.

Table 7.7 explains the empirical design decision that addresses each process question in step seven.

Table 7.7 Step seven in the design of an empirical study.

Process Question	Empirical Design Decision
What is the general population of subjects to which the hypothesis(es) about the research approach are applicable?	The general population includes software designers. This population exhibits high diversity with respect to education, experience, skill, and domain expertise in software design. The population is also geographically disperse. Sampling this population and controlling the variance in the characteristics/backgrounds of the subjects is extremely difficult if not impossible. The use of student subjects was the most practical option for the experimenter. The experimenter thought that the impact of a precise design approach would be greater for beginner designers or for designers with some education/experience (non-experts). Beginner designers, in particular, have difficulty understanding the types of decisions that a designer must make. There were two target populations of Carnegie Mellon University students. 1. Students beginning their study of software design 2. Students with education/experience in software design but without expert-level understanding of software design

<table>
<tr><th>Process Question</th><th>Empirical Design Decision</th></tr>
<tr><td>Is it possible to find a sample of subjects who represent the target population?</td><td>To better control the differences among subjects, the experimenter searched for groups of students with common levels of education and experience. For statistical purposes, the experimenter looked for groups with the most students to recruit as subjects.

The experimenter looked for classes of students studying software design or software architecture from which to recruit subjects. Due to time constraints and the need to cover requisite course requirements, many classes of students studying software related topics were not available to participate in experiments.

The instructor found two courses of approximately 80-100 students each from which to recruit subjects for two experiments.
1. Beginning undergraduate course in the design/implementation of software using C++. The course was primarily offered to students in science and engineering who were not majoring in computer science.
2. Graduate course in the study and design of distributed software systems. The course was required for students pursuing a master's degree in networking and information technology. Upper-level computer science and engineering students also enrolled in the course.</td></tr>
<tr><td>Are there ethical, legal, and institutional procedures that must be followed to conduct an experiment using human subjects?

Are there permissions that must be received from the institution in which the experiment is to be conducted?</td><td>Appendix D contains the Carnegie Mellon University Human Subject Request for the research studies. The experimenter submitted copies of Appendices E and F with the request.</td></tr>
<tr><td>How might one recruit volunteers from a potential group of subjects?</td><td>Appendix E shows an example of a call for subjects to participate in a research study.

The reader should note that Carnegie Mellon University policy requires student participation in experimental studies to be voluntary. A student participant has the right to withdraw from the study at any time without penalty other than not receiving compensation for participation.

Mandatory participation in an empirical study as part of the requirements for a course and use of artifacts resulting from the study for grading purposes is, in general, not permissible. The course instructor would need to justify that participation in the proposed empirical study is a reasonable learning objective for the course. This may be more easily done for research-oriented courses at the graduate level.

Participation was voluntary for both experiments. The task for the second experiment was one of the project options for the course taken by the students from which the subjects were recruited. Though some students completed the task without participating in the experiment, most students volunteered to be subjects for the experiment.</td></tr>
<tr><td>How should the subjects formally consent to participate in a research study?</td><td>Appendix F consists of an example of a form to obtain a subject's consent to participate in a research study. The consent form should request the participant to acknowledge that he/she has been advised as to:
1. His/her right to withdraw at any time without penalty.
2. The confidentiality of his/her name and of his/her association with data collected in the empirical study. This especially applies to any publication of the results of the study.</td></tr>
</table>

Process Question	Empirical Design Decision
If the subjects receive compensation for their participation, how might one record the receipt of this compensation?	Appendix G provides an example of a form to record a subject's receipt of compensation for participation in an experiment.

Table 7.8 summarizes the characteristics of the subjects in the research studies with respect to factors outlined in Table 6.4.

Table 7.8 Characteristics of the subjects who participated in the research studies.

Experiment	Education & Training	Experience	Natural Capabilities	Team Size
1	Programming Courses	Programming in Small	Organization	1
2	Programming& Software Engineering Courses	Programming in Small	Abstraction, Organization, & Attention to Detail	1

Step 8: Specify the detailed task to be performed by the subjects.

Table 7.9 explains the empirical design decision that addresses each process question in step eight. The reader should recall that RG and RMG are abbreviations for the Rationale Group and Rationale+Method Group, respectively.

Table 7.9 Step eight in the design of an empirical study.

Process Question	Empirical Design Decision
What resources do the subjects need to perform the task?	For the first experiment, the required resources were: • Analysis of requirements in the statement of work for the microwave oven software. (all treatment groups) See Table B.1 in Appendix B. • Analysis of the requirements for product evolution in the statement of work for the microwave oven software (RG and RMG groups). See Table B.2 in Appendix B. For the second experiment, the required resources were: • Analysis of the behavior for the Coda Client, the client-side software for the Coda distributed file system. (all treatment groups) • Coda Client software and related documents [39]. (all treatment groups) • Analysis of the requirements for evolution of the Coda Client. (RG and RMG groups) See Appendix M and Appendix O.

Process Question	Empirical Design Decision
Where will the subjects perform the task?	The subjects were expected to perform their assigned task in the environments in which they normally completed class assignments.

What are the complexity, scope, and size of the task?	The task performed by the subjects in the research study were:
	• For the first experiment, the design of the software for a microwave oven.
	• For the second experiment, the following project assignments:
	  • Analysis of the Coda Client software (preparation for the redesign assignments).
	  • Redesign of the Recoverable Virtual Memory facility of the Coda Client or RVM. See Appendix L.
	  • Redesign of the Kernel-Venus interface for the Coda Client. See Appendix N.
	  • Evaluation of the redesign of the RVM facility. See Appendix P.
	  • Evaluation of the redesign of the Kernel-Venus interface. See Appendix Q.

What is the estimated time to complete the task?	The time required for the research studies was: • For the first experiment, similar to a homework assignment or about 15 hours. • For the second experiment, conducted over a period of about 14 weeks.
If a within-subjects design is used, can the same task be performed for each treatment? If not, how will different tasks be equalized with respect to factors such as difficulty?	Due to the effects of learning, subjects for a within-subjects experiments would have to perform a new design task for each treatment. An experimenter would need to demonstrate that the design tasks across the treatments are of equal complexity, scope, and size. Exploration of ways to estimate the difficulty of design tasks was beyond the scope of the research studies. Some additional notes regarding the feasibility of a within-subject design are the following. • The available time for the first experiment was not sufficient for a within-subjects experiment. • For the second experiment, the use of three design tasks of equal complexity did not correlate to the project objectives for the course as discussed above. A within-subjects experiment was therefore not suitable.

Table 7.10 summarizes the characteristics of the tasks performed by the subjects in the research studies with respect to factors outlined in Table 6.3.

Table 7.10 Characteristics of the tasks for the research studies.

Experiment	Task Complexity	Scope Across Life Cycle	Task Size	Completion Time
1	Novelty	Design	1-10 Components	Homework (days)
2	Novelty, Functional Interrelationships	Analysis, Redesign, & Design Evaluation	11-51 Components	Project (weeks)

Step 9: Determine the statistics needed to verify the experimental results and check that the experimental design will generate the necessary data.

Table 7.11 outlines the statistics identified in step nine. Column two of the table indicates the data that is required for each technique, and column four specifies whether or not the data can be collected from the study.

Table 7.11 Step nine in the design of an empirical study.

Statistic (for each subject group)	**Required Data**	**Type of Data (Product/ Process)**	**Is this collectable data? (yes/no)**
For the change impact of the designs within each treatment group: For each level of evaluation (routine, component, and routine with comparative sizing): For each change and across all changes: • mean • median • standard deviation • maximum • minimum For the change impact means of the treatment groups: Across all changes: • analysis of variance • hypothesis test (F statistic)	For each design: For each change and across all changes: • Total size of routines reused with modification • Total size of components reused with modification • Total comparative size of routines reused with modification • Total size of routines reused without modification • Total size of components reused without modification • Total comparative size of routines reused without modification • Total software system size	product	yes
For each structural complexity measure and for the designs within each treatment group: mean For selected structural complexity measures: For the designs within a treatment group and for all designs: Correlation between the structural complexity measure and the mean change impact across all changes	For each design: • Value for each structural complexity measure • Mean change impact across all designs	product	yes

Statistic (for each subject group)	**Required Data**	**Type of Data (Product/ Process)**	**Is this collectable data? (yes/no)**
For the time spent on each design activity by the subjects in each treatment group: • mean • median • standard deviation • maximum • minimum For the total time spent on the assigned task by the subjects in each treatment group: • mean • median • standard deviation • maximum • minimum For the means (of the total time) across the treatment groups: • analysis of variance • hypothesis test (F statistic)	For each subject: Time spent on each activity of the assigned task	process	yes
For each type of error to be detected by the subjects in each treatment group: • mean • median • standard deviation • maximum • minimum For the means (of the total number of errors detected by the subjects) across the treatment groups: • analysis of variance • hypothesis test (F statistic)	For each subject: The number of each type of error detected by the subject during the design review.	process	yes
For each type of error to be detected by the experimenter for the designs in each treatment group: • mean • median • standard deviation • maximum • minimum For the means (of the total number of errors detected by the experimenter) across the treatment groups: • analysis of variance • hypothesis test (F statistic)	For each design: The number of each type of error detected by the experimenter.	product	yes

7.2 Overview of the Experimental Results and Conclusions

This section overviews the most important results of the empirical studies used to validate the research approach. The experimenter used three types of comparisons as described below to evaluate the effectiveness of the research approach. The primary measure for comparison was changeability. The reader should note that the author uses the term "conclusion" to mean a general idea or understanding reached as a result of analyzing a set of detailed observations.

1. A comparison of the designs produced by the experimental groups to a benchmark design created by the experimenter who followed the research approach resulted in the following observation and conclusion.

   Observation: The benchmark design results are significantly better than those for any experimental group.

   Conclusion: The research approach effectively reduces the impact of change.

2. A comparison of the results from each experimental group support the following observation and conclusion

   Observation: There are no significant differences between groups.

   Upon closer examination, it was apparent that many subjects who were taught the research approach did not use it correctly in the development of their designs.

   Conclusion: The research approach requires different or more training.

3. A comparison of designs produced by subjects who followed the research approach with those of the benchmark design resulted in the following observation and conclusion.

   Observation: Those subjects who were taught and correctly applied the research approach achieved results similar to the benchmark results.

   Conclusion: The research approach can be learned and is effective when applied.

The research approach is more effective at systematically reducing the impact of change than human intuition or general design practices. The experiments show that a traditional classroom lecture with practice examples may not provide sufficient training to enable beginning designers to apply the research approach. A problem for future research is how to best transfer this technology to practice.

The sections which follow provide detailed results that support the general conclusions.

7.3 Experiment 1: Change Impact

This section presents the summary statistics for the change impact values observed for the designs in each treatment group as well as for the benchmark design. The benchmark design is an example of a design for the microwave oven software created by a designer who is reasonably considered to be an expert in the use of the research approach. The experimenter created the benchmark design, shown in Appendix H.

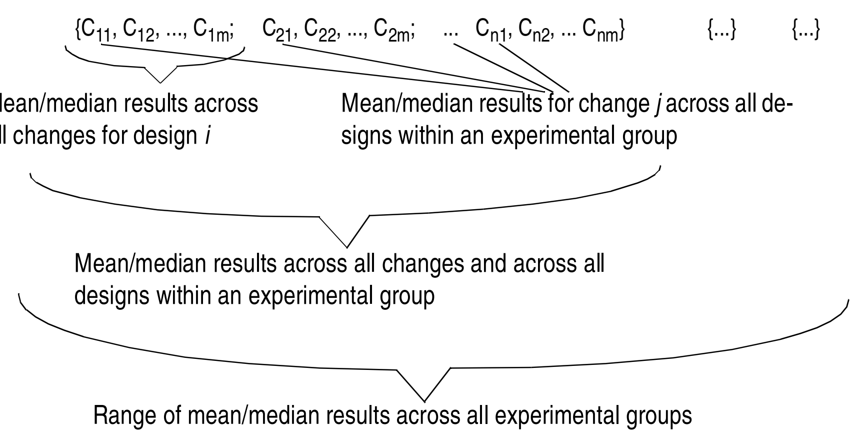
In review,

> Change impact is a measure of the part of a software system which must be modified to satisfy new requirements for the software. When measured across a set of new requirements, change impact is an average of the impact for the modifications needed to satisfy each new requirement. Mean change impact is the average change impact for the designs produced by a group of subjects, e.g. the Rationale+Method Group. Reducing change impact is important because it helps to reduce the amount of effort needed to evolve the software system.

As shown in Figure 7.1, the author reports the change impact results as mean and median values with respect to the following sets of change impact values.

- Across all types of change for a particular design *i*.
- Across all designs within an experimental group for a particular type of change *j*.
- Across all types of change and across all designs within an experimental group.

She also reports the results as a range of mean or median values across the experimental groups.



C_{ij} is the change impact value for design *i* within an experimental group and change *j*.
{…} represents the change impact values for an experimental group.

Figure 7.1 Types of change impact results.

Table 5.1 presented the types of change and their related change signatures for the expected evolution of the microwave oven software. From here on, the text refers only to the change signatures.

7.3.1 Summary Statistics for Change Impact

The change impact analysis included three different evaluations as described in Figure 7.2. The experiment used the method described in Section 4.5 to size routines and components specified in the designs produced by the subjects.

1. Change impact at the routine level: ratio of the total size of the routines reused with modification to the total size of all routines.
2. Change impact at the component level: total size of the components reused with modification to the total size of all components.
3. Change impact at the routine level with comparative sizing: total size of the comparative routines reused with modification to the total size of all routines.

   Comparative routines encapsulate the logic specified by the subject but have “standardized” sizes which reflect the size of this logic in the benchmark design. Comparative routines do not necessarily map directly to benchmark routines, although in many cases direct mapping was possible. Rather, pieces of logic in the routines map to similar pieces of logic in the benchmark design.

Figure 7.2 Types of evaluations for the change impact.

The third type of evaluation addresses the variance in the level of detail across the designs produced by the subjects. Based on the logic from the detailed benchmark design, the experimenter determined a comparative size for each routine specified as part of a subject’s design.

Analysis of change impact at the routine level across all types of changes:

Table 7.12 shows the summary statistics for the change impact at the routine level across all changes and all subjects within a treatment group. The reader can observe the results regarding the change impact at the routine level in Figure 7.3.

- The mean for the control group, 17.26%, is the lowest achieved by the treatment groups.
- The largest difference between the change impact means for any two of the treatment groups is the difference between the means for the control and rationale groups or 2.61%.
- The **mean change impact** for the **benchmark design** is:
  - 0.29 to 0.33 as large as the mean change impact for any of the treatment groups.
  - Approximately **3 times better** than the mean change impact for any of the treatment groups.
    (Dividing the mean change impact for a treatment group by 3 approximates the mean change impact for the benchmark design.)
- The **median change impact** for the **benchmark design** is:
  - 0.18 to 0.31 as large as the median change impact for any of the treatment groups.
  - Approximately **3-5 times better** than the median change impact for any of the treatment groups.

Figure 7.3 Results of the change impact across all changes at the routine level.

Table 7.12 Change impact at the routine level across all types of changes.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	17.26%	19.87%	18.58%	5.69%
Median	12.03%	20.27%	12.85%	3.71%

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Standard Deviation	15.28%	15.99%	18.17%	6.75%
Maximum	54.95%	97.30%	83.51%	25.27%
Minimum	0.00%	0.00%	0.00%	0.00%

Conclusions about the change impact at the routine level:

- There does not appear to be a significant difference between the mean change impact for each treatment group. The analysis of variance in Subsection 7.3.2 verifies this.
- The benchmark design shows a significantly lower mean and median change impact than that achieved by any of the treatment groups.

Analysis of change impact at the routine level for particular changes:

For the results of the change impact at the routine level for each type of change, the reader should see Table R.2 through Table R.15 in Appendix R. The reader can observe the results regarding the change impact at the routine level for each type of change in Figure 7.4.

- Except for the DFORM and RFORM changes, the mean change impact for the benchmark design is:
  - **0.023*** (IMSWT, 0.00% for the benchmark in comparison to 43.87% for the Rationale Group) **to 0.96** (CHD, 25.27% for the benchmark in comparison to 26.27% for the Control Group) as large as the mean change impact for any of the treatment groups.

    *Technically, 0.00%/43.87% is 0.00. This would mean that the benchmark design is 1/0.00 or infinitely better than the Rationale Group's Design. The author does not consider that this makes practical sense, particularly because a 0.00% change impact value does not reflect the effort needed to determine that there is no change impact. Hence, every occurrence of 0.00% is replaced by 1.00% in the comparative type of results.
  - Approximately **1.04 to 44 times better** than the mean change impact for any of the treatment groups.
- For the DFORM and RFORM changes, the mean change impact for all treatment designs as well as the benchmark design is 0.00%.

  All subjects correctly used the related system routine calls as specified in the design requirements. Changes to these routines, therefore, did not impact any of the designs.
- Except for the DFORM and RFORM changes, the median change impact for the benchmark design is:
  - **0.022** (IMSWT, 0.00% for the benchmark in comparison to 44.58% for the Rationale+Method Group) **to 0.96** (CHD, 25.27% for the benchmark in comparison to 26.47% for the Control Group) as large as the median change impact for any of the treatment groups.
  - Approximately **1.04 to 45 times better** than the median change impact for any of the treatment groups.
- Not considering the null DFORM and RFORM changes, there is a large range of change impact values for the different types of changes.
  - The minimum change impact is 0.00% (HLWS, Control Group and Rationale+Method Group).
  - The maximum change impact is 83.51% (PSRC, Rationale+Method Group).
  - In half of the cases, the standard deviation for the Rationale+Method Group is larger than those for the other treatment groups. In the other half of the change types, the standard deviation for the Control Group is the largest.

Figure 7.4 Results of the change impact for individual changes at the routine level.

Conclusions about the change impact at the routine level for particular changes:

- The variance within the treatment groups is high for many types of change. The Control Group and Rationale+Method Group show the largest variance within their groups.
- The benchmark design shows a lower (and often, substantially lower) mean and median change impact than that achieved by any of the treatment groups for all types of change.

Analysis of change impact at the component level across all types of changes:

Table 7.13 shows the summary statistics for the change impact at the component level across all changes and all subjects within a treatment group. The reader can observe the results regarding the change impact at the component level in Figure 7.5.

- The mean for the control group, 38.18%, is the lowest achieved by the treatment groups.
  - The largest difference between the change impact means for any two of the treatment groups is the difference between the means for the control and rationale groups or 6.81%.
- The **mean change impact** for the **benchmark design** is:
  - 0.19 to 0.22 as large as the mean for any of the treatment groups.
  - Approximately **5 times better** than the mean change impact for any of the treatment groups.
- The **median change impact** for the **benchmark design** is:
  - 0.08 to 0.14 as large as the median for any of the treatment groups.
  - Approximately **7 to 12 times better** than the median change impact for any of the treatment groups.

Figure 7.5 Results of the change impact across all changes at the component level.

Table 7.13 Change impact at the component level across all types of changes.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	38.18%	44.99%	41.66%	8.57%
Median	26.23%	42.68%	36.44%	3.59%
Standard Deviation	32.16%	32.87%	32.09%	10.36%
Maximum	100.00%	100.00%	95.65%	26.60%
Minimum	0.00%	0.00%	0.00%	0.00%

Conclusions about the change impact at the component level:

- There does not appear to be a significant difference between the mean change impact for each treatment group. The analysis of variance in Subsection 7.3.2 verifies this.
- The benchmark design shows a significantly lower mean and median change impact than that achieved by any of the treatment groups.

Analysis of change impact at the component level for particular changes:

For the results of the change impact at the component level for each type of change, the reader should see Table R.2 through Table R.15 in Appendix R. The reader can observe the results regarding the change impact at the component level for each type of change in Figure 7.6.

- Except for the DFORM and RFORM changes, the **mean change impact** for the **benchmark design** is:
  - **0.014** (IMSWT, 0.00% for the benchmark in comparison to 73.22% for the Rationale Group) **to 0.78** (FDBL, 26.60% for the benchmark in comparison to 34.14% for the Rationale+Method Group) as large as the mean change impact for any of the treatment groups.
  - Approximately **1.28 to 73 times better** than the mean change impact for any of the treatment groups.
  - For an explanation of the null DFORM and RFORM cases, the reader should see the discussion in the change analysis at the routine level for particular changes.
- Not considering the null DFORM and RFORM changes, the **median change impact** for the **benchmark design** is:
  - 1.04% larger than the median for the Control Group in the FDBL and EDA changes.
  - For all other types of change, **0.014** (IMSWT, 0.00% for the benchmark in comparison to 72.26% for the Rationale Group) **to 0.82** (CHD, 26.60% for the benchmark in comparison to 32.31% for the Control Group) as large as the median change impact for any of the treatment groups.
  - Approximately **1.22 to 73 times better** than the median change impact for any of the treatment groups (except in the FDBL and EDA cases).
- Not considering the null DFORM and RFORM changes, there is a large range of change impact values for the different types of changes.
  - The minimum change impact is 0.00% (HLWS, Control Group and Rationale+Method Group).
  - The maximum change impact is 100% (IMSWT, Control Group; Timer, Control Group and Rationale Group).

Figure 7.6 Results of the change impact for individual changes at the component level.

Conclusions about the change impact at the component level for particular changes:

- The variance within the treatment groups is high for most types of change.
- The benchmark design shows a lower (and often, substantially lower) mean and median change impact than that achieved by any of the treatment groups for all types of change.

Analysis of change impact at the routine level with comparative sizing across all types of changes:

Table 7.14 shows the summary statistics for the change impact at the routine level with comparative sizing across all changes and all subjects within a treatment group. The reader can observe the results regarding the change impact at the routine level with comparative sizing in Figure 7.7.

- The mean for the rationale+method group, 20.50%, is the lowest achieved by the treatment groups.
- The largest difference between the change impact means for any two of the treatment groups is the difference between the means for the rationale and the rationale+method groups or 5.36%.
- The **mean change impact** for the **benchmark design** is:
  - 0.22 to 0.28 as large as the mean for any of the treatment groups.
  - Approximately **4 times better** than the mean change impact for any of the treatment groups.

- The **median change impact** for the **benchmark design** is:
  - 0.14 to 0.27 as large as the median for any of the treatment groups.
  - Approximately **4 to 7 times better** than the median change impact for any of the treatment groups.

Figure 7.7 Results of the change impact across all changes at the routine level with comparative sizing.

Table 7.14 Change impact at the routine level with comparative sizing across all types of changes.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	23.08	25.86	20.50	5.69
Median	13.64	26.54	14.78	3.71
Standard Deviation	18.15	20.62	18.27	6.75
Maximum	54.29	99.45	85.23	25.27
Minimum	0.00	0.00	0.00	0.00

Conclusions about the change impact at the routine level with comparative sizing:

- There does not appear to be a significant difference between the mean change impact for each treatment group. The analysis of variance in Subsection 7.3.2 verifies this.
- The benchmark design shows a significantly lower mean and median change impact than that achieved by any of the treatment groups.

Analysis of change impact at the routine level with comparative sizing for particular changes:

For the results of the change impact at the routine level with comparative sizing for each type of change, the reader should see Table R.2 through Table R.15 in Appendix R. The reader can observe the results regarding the change impact at the routine level with comparative sizing for each type of change in Figure 7.8.

- Except for the DFORM and RFORM changes, the **mean change impact** for the **benchmark design** is:
  - **0.025** (IMSWT, 0.00% for the benchmark in comparison to 40.78% for the Rationale Group) **to 0.70** (CHD, 25.27% for the benchmark in comparison to 36.74% for the Rationale+Method Group) as large as the mean change impact for any of the treatment groups.
  - Approximately **1.43 to 41 times better** than the mean change impact for any of the treatment groups.
  - For an explanation of the null DFORM and RFORM cases, the reader should see the discussion in the change analysis at the routine level for particular changes.
- Not considering the null DFORM and RFORM changes, the **median change impact** for the **benchmark design** is:
  - 1.02% larger than the median for the Control Group in the HBSQ and DSNSR changes.
  - For all other types of change, **0.025** (IMSWT, 0.00% for the benchmark in comparison to 40.86% for the Control Group) **to 0.89** (ADO, 9.07% for the benchmark in comparison to 10.16% for the Rationale Group) as large as the median change impact for any of the treatment groups.
  - Approximately **1.12 to 41 times better** than the median change impact for any of the treatment groups (except in the HBSQ and DSNSR cases).

- Not considering the null DFORM and RFORM changes, there is a large range of change impact values for the different types of changes.
    - The minimum change impact is 0.00% (HLWS, Rationale+Method Group).
    - The maximum change impact is 99.45% (IMSWT, Rationale Group).

Figure 7.8 Results of the change impact for individual changes at the routine level with comparative sizing.

Conclusions about the change impact at the routine level with comparative sizing for particular changes:

- The variance within the treatment groups is high for most types of change.
- The benchmark design shows a lower (and often, substantially lower) mean and median change impact than that achieved by any of the treatment groups for all types of change.

7.3.2 Analyses of Variance for Change Impact

The next question is whether or not the difference between the mean change impact across all changes for each treatment group is due to the different treatments or to experimental error. The answer to this question requires one-way analysis of the variance in the change impact values within each group as well as across the groups. This is done for each of the levels of evaluation. Those readers who are not familiar with the concepts of analysis of variance, hypothesis testing via the F statistic, and analysis of correlation may refer to Appendix S which discusses these statistical techniques. The author also refers the reader to [82,96] for more detailed explanations and example applications.

Because the sizes of the treatment groups were not equal, the experimenter used three techniques (as discussed in Appendix S) for analyzing variance with unequal sample sizes. In the first case, designs with the poorest scores (highest mean change impact values) were removed to create equally sized groups. The experimenter reasoned that subjects with lower innate ability were more likely to resign from the experiment. Therefore, the groups with fewer remaining subjects had a higher percentage of subjects with the potential to create better designs. The experimenter's reasoning is based on comments from students who attempted but did not complete their designs. Of the three techniques, using equally sized groups probably yields the least reliable results.

The conclusion is the same for any observed differences in the mean change impact across the treatment groups at the different levels of change analysis.

Conclusion:

> The mean change impacts at the routine level, component level, and routine level with comparative sizing are not significantly different between the treatment groups. Any observed differences are due to experimental error alone (not due to differences in treatments).

Detailed calculations and results for analysis of variance are shown in Appendix T.

7.3.3 Analysis of Covariance for Change Impact

Since the analysis of variance did not yield any significant difference between the mean change impact for each treatment group at any design level, the experimenter applied an analysis of covariance with respect to three concomitant variables or covariates. The three covariates were the following.

- Total time spent on completing the design task
- Largest program written before completing the design task (programming experience)
- Number of computer programming courses taken before completing the design task (computer-related education)

The subjects submitted the total time that they spent on the design task and provided background information about their pre-experiment related education and experience. The experimenter analyzed the correlation between the mean change impact across all changes and at all design levels with respect to each covariate. This process consisted of a correlation between the mean change impact and covariate values for all subjects as well as a correlation within each treatment group.

The correlation tables (Table 7.15, Table 7.16, Table 7.16, and Table 7.18) indicate the following:

- For all subjects and at any design level, mean change impact across all changes has little correlation with any of the covariates.
- For the Control Group, mean change impact across all changes and time have a weak, direct correlation at the component level (0.72) .
- For the Rationale Group, mean change impact across all changes and programming experience have a weak, indirect correlation at the routine level (-0.67) and at the component level (-0.79).
- For the Rationale+Method Group, mean change impact across all changes and time have a moderate, direct correlation at the routine level and at the routine level with comparative sizing (both 0.86).
- For the Rationale+Method Group, mean change impact across all changes and programming experience have a weak, indirect correlation at the component level (-0.70).

As one would expect, the analysis of covariance with respect to each covariate resulted in acceptance of H_0 and rejection of H_1.

Conclusion:

> Differences in the mean change impact across changes at the three design levels between the treatment groups are not due to differences in the treatments, even when one takes into account any experimental error due to differences in the subjects with respect to time spent on the task, programming experience, and computer-related education.

Details of the calculations and results for the analysis of covariance appear in Appendix U.

Correlation Between Subject Characteristics and Change Impact

Table 7.15 Correlation between subject characteristics and change impact at the routine, component, and routine with comparative sizing levels. Use of equal size sample groups with 18 total subjects.

Characteristic of the Subject	Correlation to Change Impact at the Routine Level	Correlation to Change Impact at the Component Level	Correlation to Change Impact at the Routine Level With Comparative Sizing
Total Time Spent on Completing the Design Task	0.40	0.33	0.33
Largest Program Written Before Completing the Design Task	0.18	0.13	0.25
Number of Computer Programming Courses Taken Before Completing the Design Task	-0.27	-0.21	-0.22

Table 7.16 Correlation between subject characteristics of the Control Group and change impact at the routine, component, and routine with comparative sizing levels. Use of equal size samples of 6 each.

Characteristic of the Subject	Correlation to Change Impact at the Routine Level	Correlation to Change Impact at the Component Level	Correlation to Change Impact at the Routine Level With Comparative Sizing
Total Time Spent on Completing the Design Task	0.43	0.72	0.38
Largest Program Written Before Completing the Design Task	0.29	0.06	0.37
Number of Computer Programming Courses Taken Before Completing the Design Task	-0.26	-0.26	-0.28

Table 7.17 Correlation between subject characteristics of the Rationale Group and change impact at the routine, component, and routine with comparative sizing levels. Use of equal size samples of 6 each.

Characteristic of the Subject	Correlation to Change Impact at the Routine Level	Correlation to Change Impact at the Component Level	Correlation to Change Impact at the Routine Level With Comparative Sizing
Total Time Spent on Completing the Design Task	0.25	0.31	0.22

Characteristic of the Subject	Correlation to Change Impact at the Routine Level	Correlation to Change Impact at the Component Level	Correlation to Change Impact at the Routine Level With Comparative Sizing
Largest Program Written Before Completing the Design Task	-0.67	-0.79	-0.53
Number of Computer Programming Courses Taken Before Completing the Design Task	-0.49	-0.33	-0.29

Table 7.18 Correlation between subject characteristics of the Rationale+Method Group and change impact at the routine, component, and routine with comparative sizing levels. Use of equal size samples of 6 each.

Characteristic of the Subject	Correlation to Change Impact at the Routine Level	Correlation to Change Impact at the Component Level	Correlation to Change Impact at the Routine Level With Comparative Sizing
Total Time Spent on Completing the Design Task	0.86	0.68	0.86
Largest Program Written Before Completing the Design Task	-0.06	-0.70	-0.10
Number of Computer Programming Courses Taken Before Completing the Design Task	-0.14	0.05	-0.21

7.4 Experiment 1: Structural Complexity

The focus in this section is on the correlation between the values obtained for the structural complexity measures and the mean change impact across all changes, as measured across all design levels and for all treatment groups. The goal is to find structural complexity measures with strong correlations to mean change impact that can serve as predictors of change complexity in designs.

7.4.1 Summary Statistics for Structural Complexity Measures

For each treatment group as well as the benchmark design, Table 7.19, Table 7.20, and Table 7.21 show the mean value for each structural complexity measure at the routine, component, and system levels, respectively. The reader may note the difference between the benchmark design and treatment group means for measures of size. The mean for the benchmark design is larger than those for the treatment groups because the benchmark design, in general, contained much more detail than the designs created by the subjects in the treatment groups.

Table 7.19 Structural complexity measures across the routines in a design and the designs in a group.

Type of Structural Complexity Measure Per Routine	Mean Value for the Control Group	Mean Value for the Rationale Group	Mean Value for the Rationale+Method Group	Value for the Benchmark Design
Number of Routine Attributes	1.67	1.21	1.05	6.97
Size of a Routine	9.97	9.86	8.39	25.10
Number of Calls to Other Routines	3.64	3.27	3.69	4.14
Routine V(G)	2.64	2.76	2.47	5.17

Table 7.20 Structural complexity measures across the components in a design and designs in a group.

Type of Structural Complexity Measure Per Component	Mean Value for the Control Group	Mean Value for the Rationale Group	Mean Value for the Rationale+Method Group	Value for the Benchmark Design
Number of Component-Level Attributes	2.25	3.34	2.64	1.71
Number of Routines in a Component	5.04	5.06	3.63	4.00
Component Size	46.99	41.43	30.54	53.71
Fan-In	0.73	0.82	0.76	3.29
Fan-Out	1.43	1.42	1.74	3.71
Number of Calls to External Routines	16.48	11.98	12.46	8.57
Component V(G)	11.98	10.06	8.00	10.71

Table 7.21 Structural complexity measures for the system in a design and across the designs in a group.

Type of Structural Complexity Measure	Mean Value for the Control Group	Mean Value for the Rationale Group	Mean Value for the Rationale+Method Group	Value for the Benchmark Design
Number of Routines	18.50	19.42	17.38	28.00
Number of Components	3.83	4.17	4.75	13.00
System Size	175.83	153.67	123.00	376.00
System V(G)	41.67	37.33	30.38	75.00
Comparative System Size	320.00	311.25	318.25	376.00
Comparative System V(G)	68.83	68.50	71.00	75.00

7.4.2 Correlation Between Structural Complexity Measures and Change Impact

Table 7.22 indicates the correlation between structural complexity measures and change impact across all changes and all treatment groups. Appendix V contains the tables which show the correlation between structural complexity and change impact within each treatment group.

Table 7.22 Correlation between structural complexity measures and mean change impact across all changes and all treatment groups (total of 26 designs).

Structural Complexity Measure	Correlation to Mean Change Impact at the Routine Level	Correlation to Mean Change Impact at the Component Level	Correlation to Mean Change Impact at the Routine Level With Comparative Sizing
Mean Number of Routine Attributes Per Routine	-0.15	-0.01	-0.12
Mean Routine Size	0.38	0.30	0.40
Mean Number of Calls to Other Routines Per Routine	0.57	0.34	0.54
Mean Routine V(G)	0.60	0.40	0.58
Mean Number of Component Level Attributes Per Component	0.01	-0.15	0.02
Mean Number of Routines Per Component	-0.20	0.21	-0.12
Mean Component Size	0.11	0.39	0.22
Mean Fan-In Per Component	-0.40	-0.53	-0.45
Mean Fan-Out Per Component	-0.27	-0.39	-0.33
Mean Number of Calls to External Routines Per Component	0.39	0.38	0.40
Mean Component V(G)	0.35	0.47	0.38
Number of Routines in the System	-0.67	-0.58	-0.72
Number of Components in the System	-0.53	-0.68	-0.63
System Size	-0.23	-0.25	-0.24
System V(G)	-0.34	-0.31	-0.37
Comparative System Size	-0.26	-0.46	-0.28
Comparative System V(G)	-0.09	0.04	-0.01

Observations from the correlation across all treatment groups:

- No strong correlations between structural complexity and change impact are evident.
- There is a weak, indirect correlation between the number of routines in a system (partitioning) and the mean change impact at the routine level (-0.67) and at the routine with comparative sizing level (-0.72).
- There is a weak, indirect correlation between the number of components (partitioning) in a system and the mean change impact at the component level (-0.68) and at the routine with comparative sizing level (-0.63).

Observations from the correlation within the Control Group:

- There is a strong, direct correlation between the *mean number of calls to other routines per routine* (coupling) and the mean change impact at all design levels (0.95 to 0.97).
- There is a weak, direct correlation between the *mean routine V(G)* or control flow complexity and the mean change impact at the routine level (0.61) and at the component level (0.74).
- There is a moderate, indirect correlation between the *mean fan-in per component* (coupling) and the mean change impact at all design levels (-0.76 to -0.88).
- There is a moderate, indirect correlation between the *mean fan-out per component* (coupling) and the mean change impact at all design levels (-0.83 to -0.88).
- There is a moderate to strong, indirect correlation between the *number of routines in the system* (partitioning) and the mean change impact at all design levels (-0.83 to -0.98).
- There is a moderate to strong, indirect correlation between the *number of components in the system* (partitioning) and the mean change impact at all design levels (-0.80 to -0.92).
- There is a weak, indirect correlation between the *comparative system size* and the mean change impact at all design levels (-0.66 to -0.79).

Observations from the correlation within the Rationale Group:

There is little correlation between structural complexity and change impact at any design level.

Observations from the correlation within the Rationale+Method Group:

- There is a weak to moderate, direct correlation between the *mean number of routine attributes per routine* and the mean change impact at the component level (0.81) and at the routine with comparative sizing level (0.68).
- There is a weak to moderate, direct correlation between the *mean routine size* and the mean change impact at all design levels (0.70 to 0.88).
- There is a moderate, direct correlation between the *mean number of calls to other routines per routine* (coupling) and the mean change impact at all design levels (0.77 to 0.86).
- There is a weak to moderate, direct correlation between the *mean routine V(G)* or control flow complexity and the mean change impact at all design levels (0.71 to 0.89).
- There is a weak, indirect correlation between the *mean number of component level attributes per component* and the mean change impact at the component level (-0.72).
- There is a weak to moderate, direct correlation between the *mean component size* and the mean change impact at the component level (0.88) and at the routine with comparative sizing level (0.70).

- There is a moderate to strong, indirect correlation between the *mean fan-in per component* (coupling) and the mean change impact at all design levels (-0.87 to -0.95).
- There is a weak to moderate, direct correlation between the *mean number of calls to external routines per component* and the mean change impact at all design levels (0.65 to 0.87).
- There is a weak to strong, direct correlation between the *mean component V(G)* and the mean change impact at all design levels (0.64 to 0.92).
- There is a weak to strong, indirect correlation between the *number of routines in the system* (partitioning) and the mean change impact at all design levels (-0.75 to -0.95).
- There is a moderate to strong, indirect correlation between the *number of components in the system* (partitioning) and the mean change impact at all design levels (-0.83 to -0.94).
- There is a weak, indirect correlation between the *comparative system size* and the mean change impact at the component level (-0.62) and at the routine with comparative sizing level (-0.69).

Notes:

- The Control Group and Rationale+Method Group display moderate to strong correlations between mean change impact and the following structural complexity measures.
  - Mean number of calls to other routines per routine (coupling, direct correlation)
  - Mean fan-in per component (coupling, indirect correlation)
  - Number of routines in the system (partitioning, indirect correlation)
  - Number of components in the system (partitioning, indirect correlation)
- The Control Group and Rationale+Method Group display weak between comparative system size and mean change impact.
- The indirect correlation between mean fan-in per component and change impact may not be intuitive. Why would high coupling be related to low change impact? A component may encapsulate a substantial number of services that change together and are needed by other components.

7.5 Experiment 1: Design Effort

This section contains summary statistics for the time spent by the subjects on the design activities and for the number of errors detected by the subjects and by the experimenter in the first experiment. Subsection 7.5.1 displays the summary statistics for time. Subsection 7.5.3 presents the summary statistics for the number of errors (by type) detected by the subjects, and Subsection 7.5.5 shows the summary statistics for the number of errors (by type) detected by the experimenter.

The analyses of variance for the mean total time, mean total number of errors detected by the subjects, and mean total number of errors detected by the experimenter appear in Subsection 7.5.2, Subsection 7.5.4, and Subsection 7.5.6, respectively.

The analyses of variance show:

- Differences in the mean total time between the treatment groups are **not** due to the differences in treatments.
- Differences in the mean total number of errors detected by the subjects may or may not be due to the differences in treatments.

  The analysis of variance by unweighted means indicates that H_0 should be rejected, while the analysis of variance by weighted means indicates that H_0 should be accepted. $F_{calculated}$ ~ 3.39 in the first case and 3.21 in the second case, with $F_{0.05} = 3.39$.

  If one rejects H_0, then the Rationale+Method Group detected, on average, fewer errors than the other treatment groups. It is inconclusive whether this means that the subjects in the Rationale+Method Group had fewer errors in their designs or whether they are not as good at detecting errors. One might reason that differences in detection capability are randomly distributed across the treatment groups and that the lower mean total errors for the Rationale+Method Group is due to fewer errors in this group's designs.
- Differences in the mean total number of errors detected by the experimenter are **not** due to the differences in treatments.

7.5.1 Summary Statistics for Time

Table 7.23 contains the summary statistics for the total time recorded by the subjects in each treatment group.

Table 7.23 Summary statistics for time (minutes) spent on design activities for each treatment group.

Type of Activity	Statistic	Control Group	Rationale Group	Rationale+Method Group
Analysis				
	Mean	56.50	105.92	75.63
	Median	63.50	88.00	55.00
	Standard Deviation	17.60	86.31	77.28
	Maximum	75.00	300.00	180.00
	Minimum	30.00	0.00	15.00
Design				
	Mean	211.67	310.17	314.25
	Median	186.50	272.50	295.00
	Standard Deviation	81.98	146.50	111.09
	Maximum	360.00	600.00	480.00
	Minimum	130.00	145.00	165.00
Review				

Type of Activity	Statistic	Control Group	Rationale Group	Rationale+Method Group
	Mean	109.83	79.25	53.88
	Median	75.00	70.00	35.50
	Standard Deviation	105.19	45.97	48.14
	Maximum	317.00	165.00	150.00
	Minimum	20.00	15.00	15.00
All Activities				
	Mean	378.00	495.33	443.75
	Median	355.00	454.00	468.00
	Standard Deviation	117.95	232.81	182.42
	Maximum	577.00	925.00	720.00
	Minimum	260.00	229.00	225.00

7.5.2 Analysis of Variance for Total Time

This subsection contains the analysis of variance calculations and results for the total time recorded by the subjects in each treatment group.

Analysis of variance using *unequal* sample sizes and *unweighted means analysis*:

Table 7.24 Experiment 1 ANOVA: Total Time, Unequal Sample Sizes, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=6, n_2=12, n_3=8

Table 7.25 Experiment 1 ANOVA: Total Time, Unequal Sample Sizes, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 32{,}816.91$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 23$	$MSE = \frac{SSE}{df_{MSE}} = 39{,}524.58$

Table 7.26 Experiment 1 ANOVA: Total Time, Unequal Sample Sizes, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.83$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,23) = 3.39$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance using *unequal* sample sizes and *weighted means analysis*:

Table 7.27 Experiment 1 ANOVA: Total Time, Unequal Sample Sizes, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=6, n_2=12, n_3=8

Table 7.28 Experiment 1 ANOVA: Total Time, Unequal Sample Sizes, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 32{,}846.26$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 23$	$MSE = \frac{SSE}{df_{MSE}} = 39{,}524.58$

Table 7.29 Experiment 1 ANOVA: Total Time, Unequal Sample Sizes, Weighted Means Analysis -Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = .83$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,23) = 3.39$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

7.5.3 Summary Statistics for Errors Detected by Subjects

Table 7.30 contains the summary statistics for the number of errors detected by the subjects in each treatment group.

Table 7.30 Summary statistics for number and types of errors detected by each treatment group.

Type of Error	Statistic	Control Group	Rationale Group	Rationale+Method Group
Omission				
Classes				
	Mean	0.33	0.44	0.14
	Median	0.00	0.00	0.00
	Standard Deviation	0.82	0.53	0.38
	Maximum	2.00	1.00	1.00
	Minimum	0.00	0.00	0.00
Data Definitions				
	Mean	2.00	2.70	0.25
	Median	2.50	2.00	0.00
	Standard Deviation	1.67	2.36	0.46
	Maximum	4.00	8.00	1.00
	Minimum	0.00	0.00	0.00
Method Definitions				
	Mean	2.67	2.10	1.38
	Median	0.50	1.50	0.50
	Standard Deviation	3.78	2.81	2.39
	Maximum	8.00	9.00	7.00
	Minimum	0.00	0.00	0.00
Interface Parameters				
	Mean	2.17	1.57	0.88
	Median	1.50	2.00	0.00
	Standard Deviation	2.40	0.79	1.36
	Maximum	6.00	2.00	3.00

Type of Error	Statistic	Control Group	Rationale Group	Rationale+Method Group
	Minimum	0.00	0.00	0.00
Method Calls				
	Mean	1.33	1.13	0.88
	Median	0.50	1.00	0.00
	Standard Deviation	1.75	1.36	1.73
	Maximum	4.00	4.00	5.00
	Minimum	0.00	0.00	0.00
Commission				
Classes				
	Mean	1.00	1.50	0.29
	Median	0.00	0.50	0.00
	Standard Deviation	2.24	1.93	0.76
	Maximum	5.00	5.00	2.00
	Minimum	0.00	0.00	0.00
Data Definitions				
	Mean	3.67	1.63	1.25
	Median	2.50	2.00	0.50
	Standard Deviation	4.32	1.19	1.75
	Maximum	12.00	3.00	4.00
	Minimum	0.00	0.00	0.00
Method Definitions				
	Mean	1.50	2.00	0.00
	Median	1.00	1.00	0.00
	Standard Deviation	1.64	2.65	0.00
	Maximum	4.00	8.00	0.00
	Minimum	0.00	0.00	0.00
Interface Parameters				
	Mean	2.33	1.25	1.00

Type of Error	Statistic	Control Group	Rationale Group	Rationale+Method Group
	Median	2.50	1.00	0.00
	Standard Deviation	2.07	1.28	1.51
	Maximum	5.00	3.00	4.00
	Minimum	0.00	0.00	0.00
Method Calls				
	Mean	1.83	3.11	0.88
	Median	1.50	2.00	0.50
	Standard Deviation	2.23	4.17	1.13
	Maximum	6.00	12.00	3.00
	Minimum	0.00	0.00	0.00
Total Errors **(Number of Omission + Number of Commission)**				
	Mean	18.67	12.75	6.88
	Median	16.50	10.50	7.00
	Standard Deviation	13.60	12.34	6.79
	Maximum	41.00	45.00	17.00
	Minimum	5.00	0.00	0.00

7.5.4 Analysis of Variance for Total Number of Errors Detected by Subjects

This subsection contains the analysis of variance calculations and results for the total number of errors detected by the subjects in each treatment group.

Analysis of variance using *unequal* sample sizes and *unweighted means analysis*:

Table 7.31 Experiment 1 ANOVA: Total Number of Errors Detected by Subjects, Unequal Sample Sizes, Unweighted Means Analysis -Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=6, n_2=12, n_3=8

Table 7.32 Experiment 1 ANOVA: Total Number of Errors Detected by Subjects, Unequal Sample Sizes, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \dfrac{SST}{df_{MST}} = 278.09$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 23$	$MSE = \dfrac{SSE}{df_{MSE}} = 81.93$

Table 7.33 Experiment 1 ANOVA: Total Number of Errors Detected by Subjects, Unequal Sample Sizes, Unweighted Means Analysis -Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \dfrac{MST}{MSE} = 3.39$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,23) = 3.39$
Accept/Reject H_0	Reject H_0 since $F_{Calculated} = F_{\alpha}$.

Analysis of variance using *unequal* sample sizes and *weighted means analysis*:

Table 7.34 Experiment 1 ANOVA: Total Number of Errors Detected by Subjects, Unequal Sample Sizes, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=6, n_2=12, n_3=8

Table 7.35 Experiment 1 ANOVA: Total Number of Errors Detected by Subjects, Unequal Sample Sizes, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \dfrac{SST}{df_{MST}} = 81.93$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 23$	$MSE = \dfrac{SSE}{df_{MSE}} = 262.77$

Table 7.36 Experiment 1 ANOVA: Total Number of Errors Detected by Subjects, Unequal Sample Sizes, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 3.21$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,23) = 3.39$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

7.5.5 Summary Statistics for Number of Errors Detected by Experimenter

Table 7.37 contains the summary statistics for the number of errors detected by the experimenter for each treatment group.

Table 7.37 Summary statistics for number and types of errors detected by the experimenter.

Type of Error	Statistic	Control Group	Rationale Group	Rationale+Method Group
Omission				
Number of Component Attributes				
	Mean	6.17	3.50	5.00
	Median	4.00	2.00	6.00
	Standard Deviation	5.27	3.58	3.02
	Maximum	15.00	13.00	9.00
	Minimum	1.00	1.00	0.00
Required Operations				
	Mean	7.17	9.75	14.38
	Median	7.00	7.50	14.00
	Standard Deviation	2.23	6.52	8.07
	Maximum	10.00	27.00	26.00
	Minimum	4.00	1.00	2.00

Type of Error	Statistic	Control Group	Rationale Group	Rationale+Method Group
Parameters in Routine Interface				
	Mean	0.33	0.33	0.38
	Median	0.00	0.00	0.00
	Standard Deviation	0.82	1.15	0.74
	Maximum	2.00	4.00	2.00
	Minimum	0.00	0.00	0.00
Parameters in Routine Call				
	Mean	0.00	1.58	0.25
	Median	0.00	0.00	0.00
	Standard Deviation	0.00	3.87	0.71
	Maximum	0.00	13.00	2.00
	Minimum	0.00	0.00	0.00
Routine Attributes				
	Mean	2.50	0.92	2.88
	Median	1.50	1.00	3.00
	Standard Deviation	3.27	1.00	2.53
	Maximum	9.00	3.00	7.00
	Minimum	0.00	0.00	0.00
Routine Calls				
	Mean	2.83	6.42	5.75
	Median	0.00	4.00	2.00
	Standard Deviation	6.01	6.65	7.32
	Maximum	15.00	21.00	18.00

Type of Error	Statistic	Control Group	Rationale Group	Rationale+Method Group
	Minimum	0.00	0.00	0.00
Status/Error Checks				
	Mean	0.50	4.17	1.13
	Median	0.00	3.50	0.00
	Standard Deviation	1.22	5.25	1.64
	Maximum	3.00	19.00	4.00
	Minimum	0.00	0.00	0.00
Status/Error Codes				
	Mean	1.67	2.83	0.63
	Median	1.50	2.50	0.00
	Standard Deviation	1.86	3.64	1.77
	Maximum	4.00	13.00	5.00
	Minimum	0.00	0.00	0.00
Commission				
Number of Component Attributes				
	Mean	0.17	0.75	0.00
	Median	0.00	0.00	0.00
	Standard Deviation	0.41	2.60	0.00
	Maximum	1.00	9.00	0.00
	Minimum	0.00	0.00	0.00
Required Operations				
	Mean	1.50	2.00	0.75
	Median	1.00	1.00	0.50

Type of Error	Statistic	Control Group	Rationale Group	Rationale+Method Group
	Standard Deviation	1.76	3.30	0.89
	Maximum	5.00	11.00	2.00
	Minimum	0.00	0.00	0.00
Parameters in Routine Interface				
	Mean	0.17	0.17	0.00
	Median	0.00	0.00	0.00
	Standard Deviation	0.41	0.58	0.00
	Maximum	1.00	2.00	0.00
	Minimum	0.00	0.00	0.00
Parameters in Routine Call				
	Mean	1.83	0.58	0.13
	Median	0.00	0.00	0.00
	Standard Deviation	2.86	1.24	0.35
	Maximum	6.00	4.00	1.00
	Minimum	0.00	0.00	0.00
Routine Attributes				
	Mean	0.83	0.08	0.00
	Median	0.00	0.00	0.00
	Standard Deviation	2.04	0.29	0.00
	Maximum	5.00	1.00	0.00
	Minimum	0.00	0.00	0.00
Routine Calls				
	Mean	5.50	1.42	3.75

Type of Error	Statistic	Control Group	Rationale Group	Rationale+Method Group
	Median	0.50	1.00	0.50
	Standard Deviation	9.27	1.88	8.63
	Maximum	23.00	6.00	25.00
	Minimum	0.00	0.00	0.00
Status/Error Checks				
	Mean	1.00	1.00	0.50
	Median	0.00	0.00	0.00
	Standard Deviation	2.00	1.35	1.41
	Maximum	5.00	3.00	4.00
	Minimum	0.00	0.00	0.00
Status/Error Codes				
	Mean	0.00	0.33	0.00
	Median	0.00	0.00	0.00
	Standard Deviation	0.00	1.15	0.00
	Maximum	0.00	4.00	0.00
	Minimum	0.00	0.00	0.00
***Total Errors* (Number of Omission + Number of Commission)**				
	Mean	32.17	35.83	35.50
	Median	32.00	29.50	26.50
	Standard Deviation	15.42	21.11	21.33
	Maximum	58.00	89.00	71.00
	Minimum	10.00	9.00	9.00

7.5.6 Analysis of Variance for Total Number of Errors Detected by Experimenter

This subsection contains the analysis of variance calculations and results for the total number of errors detected by the experimenter for each treatment group.

Analysis of variance using *unequal* sample sizes and *unweighted means analysis*:

Table 7.38 Experiment 1 ANOVA: Total Number of Errors Detected by Experimenter, Unequal Sample Sizes, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=6, n_2=12, n_3=8

Table 7.39 Experiment 1 ANOVA: Total Number of Errors Detected by Experimenter, Unequal Sample Sizes, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g-1 = 2$	$MST = \frac{SST}{df_{MST}} = 32.89$
Error or Unexplained	SSE	$df_{MSE} = g\sum_{i=1}^{g}\langle n_i - 1\rangle = 23$	$MSE = \frac{SSE}{df_{MSE}} = 397.15$

Table 7.40 Experiment 1 ANOVA: Total Number of Errors Detected by Experimenter, Unequal Sample Sizes, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.08$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,23) = 3.39$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance using *unequal* sample sizes and *weighted means analysis*:

Table 7.41 Experiment 1 ANOVA: Total Number of Errors Detected by Experimenter, Unequal Sample Sizes, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=6, n_2=12, n_3=8

Table 7.42 Experiment 1 ANOVA: Total Number of Errors Detected by Experimenter, Unequal Sample Sizes, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 31.54$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 23$	$MSE = \frac{SSE}{df_{MSE}} = 397.15$

Table 7.43 Experiment 1 ANOVA: Total Number of Errors Detected by Experimenter, Unequal Sample Sizes, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.08$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,23) = 3.39$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

7.6 Experiment 2: Change Impact

This section presents the summary statistics and analysis of variance for the mean change impact across all changes for the redesigned Recoverable Virtual Memory (RVM) and Kernel-Venus Interface. The reader should note that evaluation of performance, though done by the subjects for the RVM design, is not part of the thesis research and therefore not presented in the dissertation.

The evaluation of change impact for the second experiment differs in the following ways from the first experiment.

- Using a prescribed sizing method, the subjects evaluated their own designs for the size of the change impact.

  In the first experiment, the experimenter evaluated the designs.
- The size of the change impact for a particular change is the sum of the sizes of the solution elements impacted by that change.

  In the first experiment, the change impact was expressed as a ratio which normalized the size of the impacted part by the size of the total system. This was necessary since the total system size differed across the designs produced by different subjects. In the second experiment, the total system size is estimated to be the size of the Coda Client software given to all subjects.
- Subjects evaluated the size of the change impact for each of the following types of solution elements, as listed below in order of increasing size.
  - Data attributes (e.g. variables defined within methods or at the class level)

- Methods
- Classes
- Files

The experimenter observed that the Rationale+Method group, in general, did not apply the research approach correctly. The exception was the approach to partitioning control flow, which was successfully applied by some of the subjects in this group. Therefore, the experimenter evaluated the designs with respect to the changes concerning control flow.

The experimenter found that for those subjects in the Rationale+Method Group who correctly applied the research approach, the mean change impact is:

- Approximately **0.10** of the mean change impact for the other treatment groups or approximately **10 times better**, across routines.
- Approximately **0.024** of the mean change impact for the other treatment groups or approximately **41** times better, across files.

Given the fact that the Rationale+Method group did not apply the research approach correctly, the experimenter was not surprised to find that the analysis of variance showed no significant differences between the mean change impact across all changes and all treatment groups. This was true for both the redesign of RVM and redesign of the Kernel-Venus Interface and for all types of solution elements except files.

In the case of files, differences in the mean change impact may or may not be due to differences in treatments as described below.

- For the RVM design, the analysis of variance with unweighted means analysis recommends that H_0 should be accepted ($F_{calculated}$ ~ 2.50 and $F_{0.05}$ = 3.26); while the analysis of variance with weighted means analysis recommends that H_0 be rejected ($F_{calculated}$ ~ 3.72 and $F_{0.05}$ = 3.26).

  If one rejects H_0, the lower mean change impact for the Rationale Group (4,675.25) and the Rationale+Method Group (3,975.93), in comparison to the Control Group (7,086.78) are due to the differences in treatment.

- For the Kernel-Venus Interface design, the analysis of variance with unweighted means analysis recommends that H_0 should be accepted ($F_{calculated}$ ~ 2.52 and $F_{0.05}$ = 3.28); while the analysis of variance with weighted means analysis recommends that H_0 be rejected ($F_{calculated}$ ~ 3.31 and $F_{0.05}$ = 3.28).

  If one rejects H_0, the lower mean change impact for the Rationale Group (3,437.88) and the Rationale+Method Group (3,872.36), in comparison to the Control Group (5,676.69) are due to the differences in treatment.

7.6.1 Summary Statistics for the Change Impact with New Recoverable Virtual Memory (RVM) Design

Table 7.44 presents the summary statistics for the size of the impacted solution elements (by type) in the new RVM designs across all types of expected change and for all treatment groups.

Table 7.44 Summary statistics for the size of the impacted solution elements for the new RVM design across all types of expected change.

Type of Solution Element	Statistic	Control Group	Rationale Group	Rationale+Method Group
Data				
	Mean	15.61	13.50	12.07
	Median	8.50	14.50	10.00
	Standard Deviation	24.42	10.89	12.99
	Maximum	80.00	0.00	42.00
	Minimum	0.00	0.00	0.00
Methods				
	Mean	535.11	248.25	210.29
	Median	306.00	27.50	34.50
	Standard Deviation	695.58	500.42	355.02
	Maximum	2,337.00	1,466.00	1,299.00
	Minimum	22.00	14.00	0.00
Classes				
	Mean	517.89	985.88	445.14
	Median	462.00	351.00	470.00
	Standard Deviation	362.78	1,610.96	158.00
	Maximum	1,420.00	4,926.00	685.00
	Minimum	0.00	85.00	186.00
Files				
	Mean	7,086.78	4,675.25	3,975.93
	Median	6,289.50	3,554.00	3,748.50
	Standard Deviation	4,564.58	2,220.94	2,013.82
	Maximum	21,862.00	8,048.00	9,264.00
	Minimum	3,166.00	1,907.00	556.00

7.6.2 Analysis of Variance for the Change Impact with the New RVM Design

The calculations and results of the analysis of variance for the size of the change impact on solution elements (by type) in the new RVM designs appear in Appendix W.

7.6.3 Summary Statistics for the Change Impact with the new Kernel-Venus Interface Design

Table 7.45 presents the summary statistics for the size of the impacted solution elements (by type) in the new Kernel-Venus Interface designs across all types of expected change and for all treatment groups.

Table 7.45 Summary statistics for the size of the impacted solution elements for the new Kernel-Venus Interface Design across all types of expected change.

Type of Solution Element	Statistic	Control Group	Rationale Group	Rationale+Method Group
Data				
	Mean	169.56	99.25	168.43
	Median	107.50	23.50	141.00
	Standard Deviation	307.58	136.79	223.79
	Maximum	1,283.00	367.00	889.00
	Minimum	1.00	3.00	0.00
Methods				
	Mean	872.81	1,117.63	861.50
	Median	588.50	1,420.50	771.50
	Standard Deviation	799.19	757.85	563.78
	Maximum	2,851.00	1,966.00	1,771.00
	Minimum	84.00	153.00	86.00
Classes				
	Mean	552.88	1,256.38	479.21
	Median	250.50	338.50	404.50
	Standard Deviation	710.05	1,832.00	332.47
	Maximum	2,770.00	4,779.00	1,133.00
	Minimum	20.00	202.00	88.00
Files				
	Mean	5,676.69	3,437.88	3,872.36
	Median	4,529.00	3,006.00	3668.00
	Standard Deviation	3,763.34	1,828.96	1,446.68
	Maximum	15,548.00	6,762.00	6,765.00

Type of Solution Element	Statistic	Control Group	Rationale Group	Rationale+Method Group
	Minimum	2,422.00	1,545.00	2,281.00

7.6.4 Analysis of Variance for the Change Impact with the New Kernel-Venus Interface Design

The calculations and results of the analysis of variance for the size of the change impact on solution elements (by type) in the new Kernel-Venus Interface designs appear in Appendix X.

7.7 Experiment 2: Design Effort

This section contains summary statistics for the time spent by the subjects on the design activities and for the number of errors detected by the subjects in the second experiment. Subsection 7.7.1 displays the summary statistics for time. Subsection 7.7.3 presents the summary statistics for the number of errors (by type) detected by the subjects.

The analysis of variance for the mean total time appears in Subsection 7.7.2. The experimenter did not conduct an analysis of variance for the mean total number of errors detected by the subjects because the summary statistics show no significant variance between the treatment groups. The subjects in all treatment groups reported a very small number of errors. The subjects evaluated their own designs according to prescribed directions, while the experimenter evaluated the designs with respect to one type of change as described in Subsection 7.6.4.

The analyses of variance for the total time shows:

Differences in the mean total time between the treatment groups for any of the four project assignments (redesign of RVM, redesign of Kernel-Venus Interface, evaluation of new RVM design, evaluation of new Kernel-Venus Interface) are **not** due to differences in the treatments.

With respect to the total number of errors reported by the subjects:

Due to few reported errors (by most subjects), there is only a small difference between the means reported for the treatment groups.

7.7.1 Summary Statistics for Time

Table 7.46, Table 7.47, Table 7.48, and Table 7.49 contain the summary statistics for the total time recorded by the subjects in each treatment group for each of the four project assignments.

Table 7.46 Summary statistics for time (minutes) spent on the redesign of RVM for each treatment group.

Type of Activity	Statistic	Control Group	Rationale Group	Rationale+Method Group
Redesign of the RVM Facility				
Review the Coda Client requirements specification.				
	Mean	281.25	135.50	162.00
	Median	90.00	75.00	105.00
	Standard Deviation	413.99	149.23	173.13
	Max	1500.00	480.00	600.00
	Minimum	5.00	15.00	0.00
Think about the current design of the Coda client RVM facility.				
	Mean	256.10	246.00	398.75
	Median	195.00	195.00	270.00
	Standard Deviation	218.66	179.33	295.88
	Maximum	900.00	600.00	1,200.00
	Minimum	42.00	60.00	140.00
Determine a new design for the RVM facility.				
	Mean	134.90	186.00	216.19
	Median	120.00	180.00	180.00
	Standard Deviation	100.68	90.33	140.16
	Maximum	360.00	300.00	600.00
	Minimum	28.00	60.00	30.00
Document the new design for the RVM facility.				
	Mean	124.95	141.00	135.31
	Median	82.50	120.00	120.00
	Standard Deviation	110.96	84.91	106.39
	Maximum	450.00	360.00	360.00
	Minimum	0.00	60.00	10.00
Document the rationale for the new design of the RVM facility.				
	Mean	96.60	106.50	140.06
	Median	60.00	60.00	60.00

Type of Activity	Statistic	Control Group	Rationale Group	Rationale+Method Group
	Standard Deviation	105.70	98.09	164.27
	Maximum	450.00	360.00	600.00
	Minimum	20.00	45.00	20.00
Review deliverables.				
	Mean	30.40	41.00	52.44
	Median	25.00	30.00	20.00
	Standard Deviation	28.08	32.81	75.63
	Maximum	120.00	120.00	300.00
	Minimum	5.00	0.00	1.00
Total for All Activities				
	Mean	924.20	856.00	1,104.75
	Median	630.00	940.00	975.00
	Standard Deviation	648.91	357.80	622.17
	Maximum	2,590.00	1,380.00	2,700.00
	Minimum	178.00	345.00	255.00

Table 7.47 Summary statistics for time (minutes) spent on the redesign of the Kernel-Venus Interface for each treatment group.

Type of Activity	Statistic	Control Group	Rationale Group	Rationale+Method Group
Redesign of the Kernel-Venus Interface				
Review the Coda Client requirements specification.				
	Mean	85.50	175.50	99.06
	Median	60.00	135.00	90.00
	Standard Deviation	81.27	201.00	73.63
	Maximum	300.00	720.00	300.00
	Minimum	0.00	15.00	15.00
Think about the current design of the Kernel-Venus Interface.				
	Mean	232.50	240.00	316.25

Type of Activity	Statistic	Control Group	Rationale Group	Rationale+Method Group
	Median	180.00	210.00	300.00
	Standard Deviation	182.75	135.65	160.58
	Maximum	810.00	480.00	600.00
	Minimum	10.00	60.00	120.00
Determine a new design for the Kernel-Venus Interface.				
	Mean	179.60	216.00	256.25
	Median	120.00	150.00	210.00
	Standard Deviation	143.97	177.09	198.56
	Maximum	600.00	600.00	780.00
	Minimum	20.00	60.00	20.00
Document the new design for the Kernel-Venus Interface.				
	Mean	186.70	213.00	230.63
	Median	180.00	180.00	180.00
	Standard Deviation	109.02	147.95	141.87
	Maximum	480.00	480.00	480.00
	Minimum	30.00	30.00	60.00
Document the rationale for the new design of the Kernel-Venus Interface.				
	Mean	97.05	90.50	80.75
	Median	60.00	120.00	30.00
	Standard Deviation	101.01	55.70	83.66
	Maximum	480.00	180.00	300.00
	Minimum	30.00	5.00	15.00
Review deliverables.				
	Mean	31.05	58.50	80.88
	Median	29.00	52.50	30.00
	Standard Deviation	27.76	49.22	103.75
	Maximum	120.00	180.00	300.00
	Minimum	0.00	0.00	5.00

Type of Activity	Statistic	Control Group	Rationale Group	Rationale+Method Group
Total for All Activities				
	Mean	812.40	993.50	1,063.81
	Median	795.00	877.50	937.50
	Standard Deviation	284.59	454.78	551.42
	Maximum	1,300.00	1,650.00	1,950.00
	Minimum	390.00	420.00	390.00

Table 7.48 Summary statistics for time (minutes) spent on the evaluation of the new RVM design for each treatment group.

Type of Activity	Statistic	Control Group	Rationale Group	Rationale+Method Group
Evaluation of the New RVM Design				
Evaluate the spatial performance of the RVM design.				
	Mean	269.85	289.50	215.88
	Median	210.00	210.00	195.00
	Standard Deviation	265.03	193.91	122.66
	Maximum	1,200.00	720.00	480.00
	Minimum	42.00	60.00	10.00
Evaluate the temporal performance of the RVM design.				
	Mean	102.70	152.00	109.94
	Median	60.00	120.00	45.00
	Standard Deviation	132.05	118.21	127.54
	Maximum	600.00	420.00	420.00
	Minimum	15.00	20.00	9.00
Evaluate the impact of change on the RVM design.				
	Mean	248.90	225.00	206.56
	Median	210.00	195.00	180.00
	Standard Deviation	140.44	154.43	135.75
	Maximum	600.00	480.00	480.00

Type of Activity	Statistic	Control Group	Rationale Group	Rationale+Method Group
	Minimum	75.00	60.00	60.00
Review deliverables.				
	Mean	115.65	49.50	58.06
	Median	30.00	60.00	30.00
	Standard Deviation	267.56	38.76	66.79
	Maximum	1,200.00	120.00	270.00
	Minimum	5.00	0.00	5.00
Total for All Activities				
	Mean	737.10	716.00	590.44
	Median	577.50	645.00	447.50
	Standard Deviation	727.16	403.44	361.83
	Maximum	3,600.00	1,500.00	1,500.00
	Minimum	170.00	300.00	232.00

Table 7.49 Summary statistics for time (minutes) spent on the evaluation of the new Kernel-Venus design for each treatment group.

Type of Activity	Statistic	Control Group	Rationale Group	Rationale+Method Group
Evaluation of the New Kernel-Venus Interface Design				
Read and think about assignment.				
	Mean	71.75	141.00	76.81
	Median	60.00	120.00	52.50
	Standard Deviation	54.00	69.35	74.31
	Maximum	240.00	300.00	300.00
	Minimum	5.00	60.00	12.00
Evaluate the Kernel-Venus interface (Deliverable 1: Questions 1-4).				
	Mean	133.25	159.00	102.06
	Median	120.00	150.00	81.50
	Standard Deviation	83.45	105.88	61.35

Type of Activity	Statistic	Control Group	Rationale Group	Rationale+Method Group
	Maximum	300.00	420.00	240.00
	Minimum	40.00	30.00	30.00
Evaluate the Kernel-Venus interface (Deliverable 1: Question 5).				
	Mean	143.00	112.50	82.06
	Median	85.00	120.00	75.00
	Standard Deviation	199.40	53.50	42.10
	Maximum	900.00	180.00	180.00
	Minimum	20.00	15.00	20.00
Evaluate the Kernel-Venus organization for ease of change.				
	Mean	176.10	114.00	117.38
	Median	130.00	120.00	120.00
	Standard Deviation	180.78	54.00	57.67
	Maximum	900.00	180.00	240.00
	Minimum	30.00	40.00	30.00
Review deliverables.				
	Mean	34.65	41.50	62.19
	Median	21.50	30.00	20.00
	Standard Deviation	33.69	39.16	130.64
	Maximum	120.00	140.00	540.00
	Minimum	5.00	0.00	5.00
Total for All Activities				
	Mean	558.75	568.00	440.50
	Median	417.50	540.00	348.00
	Standard Deviation	389.47	210.02	248.29
	Maximum	1,885.00	900.00	1,080.00
	Minimum	120.00	230.00	162.00

7.7.2 Analysis of Variance for Total Time

This subsection contains the analysis of variance calculations and results for the total time recorded by the subjects in each treatment group for each of the four project assignments.

Analysis of variance for the redesign of RVM using *unequal* sample sizes and *unweighted means analysis*:

Table 7.50 Experiment 2, Redesign of RVM ANOVA, Total Time, Unequal Size Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=20, n_2=10, n_3=16

Table 7.51 Experiment 2, Redesign of RVM ANOVA, Total Time, Unequal Size Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 233{,}237.89$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 43$	$MSE = \frac{SSE}{df_{MSE}} = 347{,}890.28$

Table 7.52 Experiment 2, Redesign of RVM ANOVA, Total Time, Unequal Size Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.67$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}((2,43) \sim 3.22)$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the redesign of RVM using *unequal* sample sizes and *weighted means analysis*:

Table 7.53 Experiment 2, Redesign of RVM ANOVA, Total Time, Unequal Size Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=20, n_2=10, n_3=16

Table 7.54 Experiment 2, Redesign of RVM ANOVA, Total Time, Unequal Size Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g-1 = 2$	$MST = \frac{SST}{df_{MST}} = 231{,}108.20$
Error or Unexplained	SSE	$df_{MSE} = g\sum_{i=1}^{g}\langle n_i - 1\rangle = 43$	$MSE = \frac{SSE}{df_{MSE}} = 347{,}890.28$

Table 7.55 Experiment 2, Redesign of RVM ANOVA, Total Time, Unequal Size Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.66$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}((2,43) \sim 3.22)$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the redesign of the Kernel-Venus Interface using *unequal* sample sizes and *unweighted means analysis*:

Table 7.56 Experiment 2, Redesign of Kernel-Venus Interface ANOVA, Total Time, Unequal Size Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=20, n_2=10, n_3=16

Table 7.57 Experiment 2, Redesign of Kernel-Venus Interface ANOVA, Total Time, Unequal Size Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 237{,}527.77$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 43$	$MSE = \frac{SSE}{df_{MSE}} = 185{,}143.25$

Table 7.58 Experiment 2, Redesign of Kernel-Venus Interface ANOVA, Total Time, Unequal Size Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 1.28$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}((2,43) \sim 3.22)$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the redesign of the Kernel-Venus Interface using *unequal* sample sizes and *weighted means analysis*:

Table 7.59 Experiment 2, Redesign of Kernel-Venus Interface ANOVA, Total Time, Unequal Size Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=17, n_2=9, n_3=15

Table 7.60 Experiment 2, Redesign of Kernel-Venus Interface ANOVA, Total Time, Unequal Size Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 299{,}751.04$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 43$	$MSE = \frac{SSE}{df_{MSE}} = 185{,}143.25$

Table 7.61 Experiment 2, Redesign of Kernel-Venus Interface ANOVA, Total Time, Unequal Size Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 1.62$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,43) = 3.22$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance for the evaluation of the new RVM design using *unequal* sample sizes and *unweighted means analysis*:

Table 7.62 Experiment 2, Evaluation of New RVM Design ANOVA, Total Time, Unequal Size Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=20, n_2=10, n_3=16

Table 7.63 Experiment 2, Evaluation of New RVM Design ANOVA, Total Time, Unequal Size Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 88{,}755.39$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{y} \langle n_i - 1 \rangle = 43$	$MSE = \frac{SSE}{df_{MSE}} = 313{,}374.46$

Table 7.64 Experiment 2, Evaluation of New RVM Design ANOVA, Total Time, Unequal Size Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.28$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,43) \sim 3.22$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance for the evaluation of the new RVM design using *unequal* sample sizes and *weighted means analysis*:

Table 7.65 Experiment 2, Evaluation of New RVM Design ANOVA, Total Time, Unequal Size Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=20, n_2=10, n_3=16

Table 7.66 Experiment 2, Evaluation of New RVM Design ANOVA, Total Time, Unequal Size Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 103{,}203.88$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 43$	$MSE = \frac{SSE}{df_{MSE}} = 313{,}374.46$

Table 7.67 Experiment 2, Evaluation of New RVM Design ANOVA, Total Time, Unequal Size Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.33$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}((2{,}43) \sim 3.22)$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the evaluation of the new Kernel-Venus Interface design using *unequal* sample sizes and *unweighted means analysis*:

Table 7.68 Experiment 2, Evaluation of New Kernel-Venus Interface Design ANOVA, Total Time, Unequal Size Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=20, n_2=10, n_3=16

Table 7.69 Experiment 2, Evaluation of New Kernel-Venus Interface Design ANOVA, Total Time, Unequal Size Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 71{,}352.65$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 43$	$MSE = \frac{SSE}{df_{MSE}} = 97{,}760.46$

Table 7.70 Experiment 2, Evaluation of New Kernel-Venus Interface Design ANOVA, Total Time, Unequal Size Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.73$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}((2,43) \sim 3.22)$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the evaluation of the new Kernel-Venus Interface design using *unequal* sample sizes and *weighted means analysis*:

Table 7.71 Experiment 2, Evaluation of New Kernel-Venus Interface Design ANOVA, Total Time, Unequal Size Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=20, n_2=10, n_3=16

Table 7.72 Experiment 2, Evaluation of New Kernel-Venus Interface Design ANOVA, Total Time, Unequal Size Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 77{,}094.48$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 43$	$MSE = \frac{SSE}{df_{MSE}} = 97{,}760.46$

Table 7.73 Experiment 2, Evaluation of New Kernel-Venus Interface Design ANOVA, Total Time, Unequal Size Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.79$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}((2,43) \sim 3.22)$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

7.7.3 Summary Statistics for Number of Errors Detected by Subjects

Table 7.74, Table 7.75, Table 7.76, Table 7.77 contain the summary statistics for the number of errors detected by the subjects in each treatment group in each of the four project assignments.

Table 7.74 Summary statistics for the total number of errors detected by the subjects in the redesign of RVM.

Statistic	Control Group	Rationale Group	Rationale+Method Group
Mean	0.55	3.90	0.75
Median	0.00	0.00	0.00
Standard Deviation	1.57	10.12	2.02
Maximum	6.00	32.00	8.00
Minimum	0.00	0.00	0.00

Table 7.75 Summary statistics for the total number of errors detected by the subjects in the redesign of the Kernel-Venus Interface.

Statistic	Control Group	Rationale Group	Rationale+Method Group
Mean	3.80	1.40	0.38
Median	0.00	0.00	0.00
Standard Deviation	9.93	3.50	1.50
Maximum	42.00	11.00	6.00
Minimum	0.00	0.00	0.00

Table 7.76 Summary statistics for the total number of errors detected by the subjects in the evaluation of the new RVM design.

Statistic	Control Group	Rationale Group	Rationale+Method Group
Mean	0.50	0.20	1.06
Median	0.00	0.00	0.00
Standard Deviation	1.82	0.63	2.35
Maximum	8.00	2.00	7.00
Minimum	0.00	0.00	0.00

Table 7.77 Summary statistics for the total number of errors detected by the subjects in the evaluation of the new Kernel-Venus Interface design.

Statistic	Control Group	Rationale Group	Rationale+Method Group
Mean	0.30	0.10	0.75
Median	0.00	0.00	0.00
Standard Deviation	1.34	0.32	2.74
Maximum	6.00	1.00	11.00
Minimum	0.00	0.00	0.00

7.8 Final Observations and Conclusions about the Experimental Results

The following results for Experiment 1 best reflect the potential benefit of the research approach.

In general,

- At all design levels and *across all changes*, the benchmark mean (median) change impact *outperforms* the comparable mean (median) for any treatment group, by as much as:
  - 3 (5) times at the routine level.
  - 5 (12) times at the component level.
  - 4 (7) times at the routine level with comparative sizing.
- At all design levels and *for most individual changes*, the benchmark mean (median) change impact *outperforms* the comparable mean (median) for any treatment group, by as much as:
  - 44 (45) times at the routine level.
  - 73 (73) times at the component level.
  - 41 (41) times at the routine level with comparative sizing.

More specifically,

- The benchmark mean or median change impact value is almost always better than the related mean or median values of the experimental groups, often substantially so. At the routine level for individual changes, for instance, it is better 72 times out of 84. The other 12 times, it is equal to the experimental groups' mean and median values. Example: For the PSRC change at the routine level, the benchmark value is 4.40%; the experimental groups' mean (median) values are: 18.36% (15.42%), 24.93% (27.62%), and 30.93% (23.26%).

  Conclusion: The research approach, when rigorously understood and applied, outperforms any approach these beginner-level designers might have known prior to the experiment or to any common-sense approaches they might have adopted during the experiment.

- For the experimental groups, the *median* change impact value is frequently and sometimes substantially better than the related *mean* change impact value.

  At the routine level for individual changes, for instance, the median values are better than the mean values 25 times out of 42. In the experimental groups, it was common for at least one subject to create a far-from-optimal design. Example: Regarding the TIMER change, the median value for the Rationale+Method Group was 9.73%, while the mean value was 22.14%. The maximum value was 64.95%.

  Conclusion: Even in the relatively simple problem of the microwave, it is easy to make substantial design mistakes.

- Often the minimum value within an experimental group is better than the benchmark value. At the routine level for individual changes, for instance, the experimental group minima are better 22 times out of 42. Example: For the HBSQ change at the routine level, the experimental group minima are 4.12%, 1.95%, and 8.77%; while the benchmark value is 11.26%.

  Conclusion: Not unexpectedly, the research approach does not automatically lead to the very best design. It is not a complete replacement for human creativity and insight.

- The experimental groups' mean (median) results at the routine level are better compared to the benchmark than their results at the component level. At the routine level across all types of change, the benchmark is approximately 3 (3-5) times better and at the component level 5 (7-12) times.

  Conclusion: Design is more difficult at the "big picture" level, the architectural design level. It is more difficult to apply existing knowledge, common sense approaches, or a newly learned method at this level than in the more local situation of optimizing a routine.

- The most obvious difference between the experimental group taught the research approach and the other groups occurred for the FDBL change, a change that is local to a routine. The Rationale+Method Group mean value at the routine level is 13.70%, while the mean values for the Control and Rationale groups are 21.17% and 21.43%, respectively.

  Conclusion: There is an indirect correlation between the scope of the solution to which the research approach is applied and the ease by which it is applied. This reinforces the previous conclusion about architectural versus routine-level design.

Four structural complexity measures indicate moderate to strong correlation with change impact as summarized in Table 7.78. The correlation coefficient values range across the experimental groups.

Conclusion: The structural complexity measures shown in Table 7.78 may serve as predictors of change complexity during the creation of a design.

Table 7.78 Structural complexity measures with moderate to strong correlation to change impact.

Type of Structural Complexity Measure	Structural Complexity Measure	Correlation Coefficient
Coupling	Mean number of calls to other routines per routine	0.77 to 0.97
Coupling	Mean fan-in per component	-0.76 to -0.95
Partitioning	Number of routines in the system	-0.75 to -0.98
Partitioning	Number of components in the system	-0.80 to -0.94

The results from Experiment 2 particularly demonstrate the effectiveness of the control flow technique when it is applied correctly. Designs produced by subjects who were taught and applied the technique reduced the impact of change by as much as:

- 10 times across routines.
- 41 times across files.

  Conclusion: The impact of poor partitioning decisions is greater for higher-level design components (e.g. files).

The next section presents anecdotal information about the experiments that may help others who plan to design empirical studies in software engineering.

7.9 Anecdotal Information

In addition to the differences in the innate capabilities of the subjects, a high attrition rate contributed to experimental error.

Observations from the first experiment help to explain the attrition and indicate that software design is difficult for beginning software developers.

- The number of completed designs was low compared to the attendance at the lectures.
  - Other class work and activities competed for the participants' time.
  - The design task required more effort than the participants expected.
  - The design task was too difficult for many of the participants.
  - The design task was optional and not required (according to organizational policy).
  - Specific explanations from students were the following.

    "I'm sorry that I cannot finish the experiment by tomorrow. I'm overwhelmed by homeworks, quizzes, and interviews."

    "I am very sorry to tell you that I couldn't participate in the research study due to my heavy workload this week."

    "Through my own fault and lack of time in the past week and a half, I have been unable to commit the time to the project. I apologize deeply that I am backing out, especially at so late a point in time. I hope your project is successful and thank you for offering such an opportunity to students."

- The attendance to the Question-Answer sessions was low.

    - Some participants asked questions via email.
    - Undergraduate students sleep on Saturday (explained by several subjects who did not attend the Saturday session).
    - The participants started the task a day or two before the deadline and therefore did not have questions for the sessions (even though the last session was held on the night before the task was to be submitted).
    - Other class work and activities compete for the participants' time.

- Specific comments from subjects indicate that some of them do not understand design, in general, or design using an object-oriented approach.

> "I cannot do this because I don't plan things out before I write the actual code."
>
> "Where do I start?"
>
> "Why should I decompose classes?"
>
> "Can this be done in just one class microwave? What is the general rule for deciding when we should use another class? Because there could be more than one power source and sensor, does that mean I have to be make each in a class of its own? I find the fewer the classes the shorter the program."

Table 7.79 indicates the attrition that occurred in the first experiment.

Table 7.79 Attrition in the first experiment.

Activity	Number of People in Control Group	Number of People in Rationale Group	Number of People in Rationale+Method Group
Registration	20	19	21
Attendance at Lecture	17	20	18
Completion of Design	7	13	10
Completion of Usable Design	6	12	8

Table 7.80 indicates the attrition that occurred in the second experiment.

Table 7.80 Attrition in the second experiment.

Activity	Number of People in Control Group	Number of People in Rationale Group	Number of People in Rationale+Method Group
Registration	21	17	18
Attendance at Lecture(s)	21	17	18
Completion of All Assignments	20	10	16
Completion of Usable RVM Evaluation	18	8	14
Completion of Usable Kernel-Venus Evaluation	16	8	14

In addition to the treatment lecture and practice examples, the experimenter conducted help sessions and answered questions via e-mail and private meetings. The experimenter forwarded a question and answer only to the subjects in the same treatment group as the person who asked the question. This helped to eliminate any potential cross exposure of treatment across the treatment groups.

The traditional classroom instruction with supplemental help was not sufficient to transfer the research approach to the Rationale+Method Group. Of the 8 usable designs produced by the Rationale+Method Group in Experiment 1, only 2 indicated that their designers correctly applied the research approach. As discussed in the next chapter, further research is needed to better understand how to effectively teach the research approach to designers of varying skill levels.

The next chapter summarizes the contents of the dissertation and outlines the contributions of the thesis research.

8 Summary

In summary, this dissertation presented a research approach for the systematic generation of an evolvable software design and analyzed it theoretically as well as experimentally.

The thesis contributes four types of theoretical results illustrated in detail by practical examples:

1. A research approach consisting of precise steps to decompose a solution into parts that ease future changes, which is a commonly pursued goal of software engineering.
   a. An analytical method that partitions data with operations into components with the goal of reducing the scope and size of the system that must evolve to satisfy new requirements.
   b. Analytical methods for grouping control instructions with the same goal of reducing the scope and size of the system that must evolve.
      (1) Optimal but time-consuming partitioning.
      (2) Heuristical but time-efficient partitioning to achieve near optimal partitions.
2. A research approach that combines, in a novel way, steps that require designer judgment with steps that use analytical methods.
3. A theoretical analysis of the research approach in mathematical terms including a proof of the control flow heuristic being near optimal.
4. A theoretical analysis of the research approach with respect to automation.

The thesis contributes three types of experimental results:

1. Evidence of the effectiveness of the research approach and its analytical methods: a benchmark design produced by closely following the approach outperformed the mean (median) results of the groups by factors of 3-5 (5-12) across all design levels and types of change.

   While the experimental groups consisted largely of beginning designers, this does not detract from the value of the approach. In practice, many software developers have only rudimentary formal design education, if any.
2. Results about the difficulty of transferring a design approach into practice by traditional classroom teaching.

   Overall, the experimental groups taught the approach did not perform any better than the experimental control groups. Teaching the approach did not have a statistically relevant effect. This was true for beginners (Experiment 1) as well as for developers with some experience (Experiment 2).
3. Results about the correlation between the impact of software changes and well-known complexity metrics.

The thesis has value beyond its individual contributions. It demonstrates a comprehensive plan for exploring new software engineering approaches in the laboratory and for validating them in practice. The thesis integrates theory, practical illustration, and experimentation.

Additionally, the thesis contributes ideas on how to address the practical difficulties of experiments in software engineering. For example, the experimenter created the benchmark design to bring the experimental subjects' designs to a consistent level of detail and completeness, in particular to make sizes of changes comparable.

The next chapter concludes the dissertation with a discussion of future research directions and final remarks.

9 Future Research Directions and Final Remarks

There are two major areas for future research, each of which can be subdivided into subordinate directions:

1. Theory

   a. Analytical design methods: The design approach defined in this thesis demonstrates that one can formalize the derivation of partial optimal solutions (such as for grouping control instructions).

      (1) Future research should identify other design areas for which optimal solutions can be produced analytically. Distributing software across processors and grouping data in database tables are candidate areas in which substantial work has already been done [34,42]. A further research step is finding ways to combine the partial analytical methods to create a more unified design space. The goals include, in particular, the resolution of conflicting partial optimizations into a balanced overall optimization.

      (2) Algorithms developed by researchers working in the areas of data analysis and clustering theory may help software researchers to model and automate additional features of the design process. The reader should see [43] for an extensive collection of data clustering algorithms and [123] for a review of the literature on clustering theory.

      (3) Finally, one can ask how the rich knowledge about design currently available can be organized to make its use easier for practitioners who are not deep experts. How can one combine traditional methods like object orientation, analytical design methods, ideas from the patterns and architectural styles community, and others?

   b. Automation: The design approach defined in this thesis still requires judgment in many of its steps. As the experiments demonstrated, beginning designers may not be able to quickly learn how to apply the research approach despite its overall precision and the presence of analytical methods.

      (1) Hence, automating the design work is another research topic. For the analytical methods within the research approach, this is possible, as discussed in this dissertation.

      Questions that must be resolved include:

      In what form should the information be input to a tool, managed by the tool, and exported to related tools (such as code generators or repositories)?

      How can the deterministic analytical methods be integrated with artificial intelligence or genetic algorithms?

      (2) One can also explore the somewhat "philosophical territory" of computational complexity and explore the limits of automation. Should the main research effort be made in automating design or on the even more ambitious goal of directly converting requirements into code, thus obviating the need for any automated assistance to human designers?

2. Experimental Approach

   a. Confirmation and refinement of conclusions about human factors: This thesis has produced data, observations, and conclusions about the application of a research design approach by people. These results merit further attention. For example, three questions that this thesis was unable to pursue concerned the broader effects of teaching a design approach.

      (1) If the same experimental groups are exposed to their treatments (instruction) over several problems, will they draw greater value from it over time?

      (2) Do experimental groups with different compositions (e.g., subjects from a non-

academic setting) react differently to the research approach?

(3) Is there a way to have the same subjects receive the different treatments (within-subject experiment) to reduce the experimental error due to differences in hard-to-measure innate capabilities?

b. Choice of data to collect: A difficulty of current design experiments is the lack of a commonly accepted, minimal set of factors that are indicative of design quality.

(1) Further experimentation and analysis of correlations can help simplify the task of the experimenter by identifying a basic set of factors from which others can be inferred.

(2) The validity of any metric depends upon how well it measures the targeted behavior. A study of the type of cognitive tasks that programmers perform in maintaining and evolving software systems may also help in the development of more comprehensive metrics [157,152].

c. Choice of application domain for the task: Jackson and Jackson suggest that hierarchical decompositions yield reusable software modules in numerical applications but not necessarily so in other domains [76]. Does the research approach yield evolvable components in less mathematically-oriented applications such as multi-media and e-commerce? There is good reason to think that this would be so since the steps in the approach do not depend on features specific to a particular class of applications.

d. Experiment logistics: Providing software engineering researchers with different conditions in which to conduct experiments merits attention.

For this research, the experimenter depended on the goodwill of instructors and students at Carnegie Mellon University, who allowed the experimenter to use their courses for the experiments. This imposed conditions on the experiment, in terms of duration, effort, and choice of topic. Software engineering is a discipline affected by many human factors. Researchers in software engineering need to explore ways to learn more about this crucial aspect.

Final remarks:

Though the idea of applying information-theoretical models to the analysis of software structural complexity is more than twenty years old [23], the systematic application of information theory to software design is an open area of study. There is a need for further research and development of methods to systematically and semi-automatically synthesize designs that satisfy system design constraints such as evolvability and adaptability to changing application objectives and available system resources [45].

Bibliography

[1] Adams, J. and D. Thomas, “Design Automation for Mixed Hardware-Software Systems,” *Electronic Design*, Vol. 45, No. 5, 1997, pp. 64-66 and pp. 71-72.

[2] Albrecht, A., “Measuring Application Development Productivity,” In *Proceedings Joint SHARE/ GUIDE/IBM Application Development Symposium*, Oct. 1979, pp. 34-43.

[3] Alexander, C., *Notes on the Synthesis of Form*, Harvard University Press, Cambridge, MA, 1964.

[4] Alexander, C., *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, New York, NY, 1977.

[5] Alexander, C., *The Timeless Way of Building*, Oxford University Press, New York, NY, 1979.

[6] Allen, R., *A Formal Approach to Architecture*, Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1998.

[7] Allen, R. and D. Garlan, “A Formal Basis for Architectural Connection,” *ACM Transactions on Software Engineering and Methodology*, Jul. 1997.

[8] Altmeyer, J., B. Schürmann, M. Schütze, “Generating ECAD Framework Code from Abstract Models,” In *Proceedings of the Thirty-Second Design Automation Conference*, San Francisco, CA, Jun. 1995.

[9] Arbib, M., *Brains, Machines, and Mathematics*, 2nd ed., Springer-Verlag, New York, NY, 1987.

[10] Arnold, R., Ed., *Tutorial on Software Restructuring*, IEEE Computer Society Press, Washington, D.C., 1986.

[11] Arnold, R., “An Introduction to Software Restructuring,” In *Tutorial on Software Restructuring*, R. Arnold, Ed., IEEE Computer Society Press, Washington, D.C., 1986, pp. 1-11.

[12] Asada, T., R. Swonger, N. Bounds, and P. Duerig, *The Quantified Design Space: A Tool for the Quantitative Analysis of Designs*, Technical Report CMU-CS-92-213, Carnegie Mellon University, Pittsburgh, PA, Nov. 1992.

[13] Asada, T., R. Swonger, N. Bounds, and P. Duerig, “The Quantified Design Space,” Section 5.2 in *Software Architecture: Perspectives on an Emerging Discipline* by M. Shaw and D. Garlan, Prentice-Hall, Upper Saddle River, NJ, 1996, pp. 116-127.

[14] Bachman, F., L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, *Volume II: Technical Concepts of Component-Based Software Engineering*, Technical Report CMU/ SEI-2000-TR-008, Carnegie Mellon University, Pittsburgh, PA, Jul. 2000.

[15] Ballista Project, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/edrc-ballista/www/ as of Jan. 2001.

[16] Basili, V., L. Briand, and W. Melo, “A Validation of Object-Oriented Design Metrics as Quality Indicators,” *IEEE Transactions on Software Engineering*, Vol. 22, No. 10, Oct. 1996, pp. 751-761.

[17] Basili, V. et al., “Final Report: NSF Workshop on a Software Research Program For the 21st Century,” *ACM Software Engineering Notes*, Vol. 24, No. 3, May 1999, pp. 37-44.

[18] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, Reading, MA, 1998, pp. 32-33.

[19] Bass, L., P. Clements, and R. Kazman, "A-7E: A Case Study in Utilizing Architectural Structures," Chapter 3 in *Software Architecture in Practice*, Addison-Wesley, Reading, MA, 1998, pp. 45-71.

[20] Bass, L., P. Clements, and R. Kazman, "Analyzing Development Qualities at the Architectural Level," Chapter 9 in *Software Architecture in Practice*, Addison-Wesley, Reading, MA, 1998, pp. 189-220.

[21] Batory, D. and B. Geraci, "Composition Validation and Subjectivity in GenVoca Generators," *IEEE Transactions on Software Engineering*, Special Issue on Software Reuse, Feb. 1997, pp. 67-82.

[22] Beam, W., *Systems Engineering, Architecture and Design*, McGraw-Hill, New York, NY, 1990.

[23] Belady, L., "Complexity of Large Systems," Chapter 13 in *Software Metrics: An Analysis and Evaluation*, A. Perlis, F. Sayward, and M. Shaw, Eds., MIT Press, Cambridge, MA, 1981, pp. 225-233.

[24] Bell, C. and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, NY, 1971.

[25] Birmingham, W., A. Gupta, and D. Siewiorek, *Automating the Design of Computer Systems: The MICON Project*, Jones and Bartlett, Boston, MA, 1992.

[26] Boehm, B., "The High Cost of Software," Keynote Address in *Proceedings of a Symposium on the High Cost of Software*, J. Goldberg, Ed., Naval Postgraduate School, Monterey, CA, Sep. 17-19, Stanford Research Institute, Menlo Park, CA, 1973.

[27] Booch, G., *Object-Oriented Analysis and Design: With Applications*, 2nd ed., Benjamin/Cummings, Redwood City, CA, 1994.

[28] Bredemeyer Consulting, *Software Architecture Links*, http://www.bredemeyer.com/links.htm as of Jan. 2001.

[29] Brilliant, S. and J. Knight, "NSF Workshop on Empirical Research in Software Engineering," *ACM Software Engineering Notes*, Vol. 24, No. 3, May 1999, pp. 45-52.

[30] Britton, K. and D. Parnas, *A-7E Software Module Guide*, NRL Memorandum Report 4702, Dec. 1981.

[31] Brooks, F., *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition, Addison-Wesley, Reading, MA, 1995.

[32] Burns, A. and A. Wellings, "Introduction to Real-Time Systems," Chapter 1 in *Real-Time Systems and Their Programming Languages*, Addison-Wesley, Wokingham, England, 1990, pp. 1-14.

[33] Burns, A. and A. Wellings, "Designing Real-Time Systems," Chapter 2 in *Real-Time Systems and Their Programming Languages*, Addison-Wesley, Wokingham, England, 1990, pp. 15-39.

[34] Burns, A. and A. Wellings, "Distributed Systems," Chapter 13 in *Real-Time Systems and Their Programming Languages*, Addison-Wesley, Wokingham, England, 1990, pp. 369-429.

[35] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, Chichester, England, 1996.

[36] Castaneda, W., “Software Complexity Analysis on Department of Defense Real-Time Systems,” In *Proceedings of the Second IEEE High-Assurance Systems Engineering Workshop*, Aug. 11-12, IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 130-131.

[37] Chidamber, S. and C. Kemerer, “Towards a Metric Suite for Object-Oriented Design,” In *Proceedings of OOPSLA*, *Sigplan Notices*, Vol. 11, No. 26, 1991, pp. 197-211.

[38] Clements, P., “From Subroutines to Subsystems: Component-Based Software Development,” *The American Programmer*, Vol. 8, No. 11, Nov. 1995.

[39] Coda and Odyssey Projects, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, http://www.cs.cmu.edu/afs/cs/project/coda/Web/coda.html as of Jan. 2001.

[40] Conte, S., H. Dunsmore, and V. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings, Menlo Park, CA, 1986.

[41] Cormen, T., C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[42] Date, C., *An Introduction to Database Systems*, 6th ed., Addison-Wesley, Reading, MA, 1994.

[43] Diday, E., *New Approaches in Classification and Data Analysis*, Springer-Verlag, Berlin, Germany, 1994.

[44] Ebert, C., “Classification Techniques for Metric-Based Software Development,” *Software Quality Journal*, Vol. 5, No. 4, Dec. 1996, pp. 255-272.

[45] Fayad, M. and M. Cline, “Aspects of Software Adaptability”, *Communications of the ACM*, Vol. 39, No. 10, Oct. 1996, pp. 58-59.

[46] Fenton, N., *Software Metrics: A Rigorous Approach*, Chapman & Hall, London, 1991.

[47] Florac, W. and A. Carleton, *Measuring the Software Process: Statistical Process Control for Software Process Improvement*, Addison-Wesley, Reading, MA, 1999.

[48] Frakes, W. and S. Isoda, “Success Factors for Systematic Reuse,” *IEEE Software*, Sep. 1994, pp. 14-19.

[49] Gagliardi, M., R. Rajkumar, and L. Sha, “Designing for Evolvability: Building Blocks for Evolvable Real-Time Systems,” In *Proceedings of the Real-Time Technology and Applications Symposium*, Jun. 10-12, IEEE Computer Society, Los Alamitos, CA, 1996.

[50] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[51] Garlan, D., R. Allen, and J. Ockerbloom, “Architectural Mismatch, or, Why it’s hard to build systems out of existing parts,” In *Proceedings of the 17th International Conference on Software Engineering* , Apr. 1995.

[52] Garman, J., “The ‘Bug’ Heard ‘Round the World,” *ACM SIGSOFT: Software Engineering Notes*, Vol. 6, No. 5, Oct. 1981.

[53] Goldberg, D., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

[54] Grady, R., *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[55] Gupta, A. and D. Siewiorek, “M1: A Small Computer System Synthesis Tool,” Technical Report CMUCAD-90-8, Carnegie Mellon University, Pittsburgh, PA, Feb. 1990.

[56] Halang, W., “Real-Time Systems: Another Perspective,” K.M. Kavi, Ed., *Real-Time Systems: Abstractions, Languages, and Design Methodologies*, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 11-18.

[57] Halstead, M., *Elements of Software Science*, Elsevier/North-Holland, New York, NY, 1977.

[58] Hecht, H., “What are the Most Critical Challenges to Integrating High Assurance Systems?,” In *Proceedings of the Fourth IEEE High-Assurance Systems Engineering Workshop*, Nov. 17-19, IEEE Computer Society Press, Los Alamitos, CA, 1999, p. 227.

[59] Henderson-Sellers, B., *Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall, Upper Saddle River, NJ, 1996, p. 84.

[60] Ibid., p. 124.

[61] Ibid., pp. 128-129.

[62] Ibid., pp. 150-156.

[63] Hofstadter, D., *Gödel, Escher, Bach: An Eternal Golden Braid*, Vintage Books, New York, NY, 1980.

[64] Hoover, C., “TAP-D: A Model for Developing Specialization Tracks in a Graduate Software Engineering Curriculum,” *Annals of Software Engineering*, Vol. 6, Baltzer Science Publ., Bussum, The Netherlands, 1998, pp. 253-279.

[65] Hoover, C. and P. Khosla, “An Analytical Approach to Change for the Design of Reusable Real-Time Software,” In *Proceedings of the Second Workshop on Object-Oriented Real-Time Dependable Systems*, Feb. 1-2, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 144-151.

[66] Hoover, C. and P. Khosla, “Analytical Design of Evolutionary Control Flow Components,” In *Proceedings of the Second IEEE High-Assurance Systems Engineering Workshop*, Aug. 11-12, IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 48-55.

[67] Hoover, C. and P. Khosla, “Analytical Partition of Software Components for Evolvable and Reliable MEMS Design Tools,” In *Proceedings of the Third IEEE High-Assurance Systems Engineering Symposium*, Nov. 13-14, IEEE Computer Society Press, Los Alamitos, CA, 1998, pp. 188-199. A revised version published in the *International Journal of Software Engineering and Knowledge Engineering*, World Scientific Press, Singapore, 1999.

[68] Hoover, C., P. Khosla, and D. Siewiorek, “Analytical Design of Reusable Software Components for Evolvable, Embedded Applications,” In *Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering & Technology*, Mar. 24-27, IEEE Computer Society Press, Los Alamitos, CA, 1999, pp. 70-77.

[69] Hudak, J., B. Suh, D. Siewiorek, and Z. Segall, “Evaluation & Comparison of Fault-Tolerant Software Techniques,” *IEEE Transactions on Reliability*, Vol. 42, No. 2, Jun. 1993, pp. 190-204.

[70] Humphrey, W., *A Discipline for Software Engineering,* Addison-Wesley, Reading, MA, 1995.

[71] Humphrey, W., *Managing Technical People: Innovation, Teamwork, and the Software Process,* Addison-Wesley, Reading, MA, 1996.

[72] *IEEE International Conference on Software Maintenance,* proceedings from IEEE Computer Society Press, Los Alamitos, CA. In particular, a Keynote Address presented at the conference in 2000 addressed the issues and challenges for software maintenance in the new millennium, http://www.computer.org/proceedings/icsm/0753/0753toc.htm as of Jan. 2001.

[73] *IEEE Software,* Special Issue on Systematic Reuse, Vol. 11, No. 5, Sep. 1994.

[74] *IEEE Software,* Special Issue on Legacy Systems, Vol. 12, No. 1, Jan. 1995.

[75] *IEEE Transactions on Software Engineering,* Special Section on Software Reuse, Vol. 23, No. 2, Feb. 1997.

[76] Jackson, D. and M. Jackson, "Problem Decomposition for Reuse," Technical Report CMU-CS-95-108, Carnegie Mellon University, Pittsburgh, PA, 1995.

[77] Jones, T. Capers, "Reusability in Programming: A Survey of the State of the Art," *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, Sep. 1984, pp. 488-494.

[78] Kan, S., *Metrics and Models in Software Quality Engineering*, Addison-Wesley, Reading, MA, 1995.

[79] Kazman, R., "Tool Support for Architecture Analysis and Design," In *Proceedings of the Second Software Architecture Workshop*, San Francisco, CA, Oct. 1996, pp. 94-97.

[80] Kazman, M. Klein, and P. Clements, ATAM: A Method for Architecture Evaluation, Technical Report SEIR 00-4, Carnegie Mellon University, Pittsburgh, PA, Aug. 2000.

[81] Kazman, R., B. Barbacci, M. Klein, and S. Carriere, "Experience with Performing Architecture Tradeoff Analysis," In *Proceedings of the International Conference on Software Engineering*, May 16-22, Los Angeles, CA, ACM 1999, pp. 54-63.

[82] Keppel, G., *Design and Analysis: A Researcher's Handbook,* 3rd ed., Prentice-Hall, Englewood Cliffs, NJ, 1991.

[83] Ibid., pp. 27-32.

[84] Ibid., pp. 45-56.

[85] Ibid., pp. 279-288.

[86] Ibid., pp. 301-317.

[87] Khoshgoftaar, T. and E. Allen, "Predicting Fault-Prone Software Modules in Embedded Systems with Classification Trees," In *Proceedings of the Fourth IEEE High-Assurance Systems Engineering Symposium,* IEEE Computer Society Press, Los Alamitos, CA, Nov. 17-19, 1999, pp. 105-112.

[88] Kirkpatrick, S., C. Gelatt, Jr., and M.Vecchi, "Optimization by Simulated Annealing", *Science,* Vol. 220, No. 4598, May 13, 1983, pp. 45-54.

[89] Kitchenham, B., “Evaluating Software Engineering Methods and Tools,” Series of 12 articles in select issues of *ACM Software Engineering Notes*, Vols. 21-23, 1996-1998.

[90] Klein, M. and R. Kazman, *Attribute-Based Architectural Styles*, Technical Report SEIR 99-22, Carnegie Mellon University, Pittsburgh, PA, Dec. 1999.

[91] Knuth, D., *The Art of Computer Programming*, Vols. 1-3, rev. ed., Addison-Wesley, Reading, MA, 1998.

[92] Koopman, P. and J. DeVale, “The Exception Handling Effectiveness of POSIX Operating Systems,” *IEEE Transactions on Software Engineering*, Vol. 26, No. 9, Sep. 2000, pp. 837-848.

[93] Lane, T., Studying Software Architecture through Design Spaces and Rules, Technical Report SEIR 90-18, Carnegie Mellon University, Pittsburgh, PA, Oct. 1990.

[94] Lane, T., A Design Space and Design Rules for User Interface Software Architecture, Technical Report SEIR 90-22, Carnegie Mellon University, Pittsburgh, PA, Oct. 1990.

[95] Lane, T., “Guidance for User-Interface Architectures,” Section 5.1 in *Software Architecture: Perspectives on an Emerging Discipline* by M. Shaw and D. Garlan, Prentice-Hall, Upper Saddle River, NJ, 1996, pp. 97-115.

[96] Lapin, L., *Probability and Statistics for Modern Engineering*, Brooks/Cole Publ. Co. (Wadsworth), Belmont, CA, 1983.

[97] Ibid., pp. 28-31.

[98] Ibid., pp. 343-347.

[99] Ibid., pp. 476-478.

[100] Ibid., pp. 478-481.

[101] Ibid., pp. 481-485.

[102] Lawrence, T., “The Quality of Service Model and High Assurance,” In *Proceedings of the Second IEEE High-Assurance Systems Engineering Workshop*, Aug. 11-12, IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 38-39.

[103] Leach, R., *Software Reuse: Methods, Models, and Cost*; McGraw-Hill Co., New York, NY, 1997.

[104] Lee, M., B. Barta, and P. Juliff, Eds., *Software Quality and Productivity: Theory, Practice, Education, and Training*, Chapman & Hall, London, UK, 1995.

[105] Leveson, N., *Safeware: System Safety and Computers*, Addison-Wesley, Reading, MA, 1995.

[106] Li, W. and S. Henry, “Object-Oriented Metrics That Predict Maintainability,” *Journal of System Software*, Vol. 23, 1993, pp. 111-122.

[107] Luckham, D., J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann, “Specification and Analysis of System Architecture Using Rapide,” *IEEE Transactions on Software Engineering*, Special Issue on Software Architecture, Vol. 21, No. 4, Apr. 1995, pp. 336-355.

[108] Maxion, R. and R. Olszewski, "Eliminating Exception Handling Errors with Dependability Cases: A Comparative, Empirical Study," *IEEE Transactions on Software Engineering*, Vol. 26, No. 9, Sep. 2000, pp. 888-906.

[109] McCabe, T., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 4, No. 2, 1976, pp. 308-320.

[110] McCall, J., P. Richards, and G. Walters, *Factors in Software Quality*, Vols. I, II,III, US Rome Air Development Center Reports NTIS AD/A-049 014, 015, 055, 1977.

[111] Meyer, B., "The Significance of Components," *Software Development Online*, Nov. 1999. The home web page for *Software Development Online* is http://www.sdmagazine.com as of Jan. 2001.

[112] Meyer, B., "What to Compose?," *Software Development Online*, Mar. 2000. The home web page for *Software Development Online* is http://www.sdmagazine.com as of Jan. 2001.

[113] Mili, H., F. Mili, and A. Mili, "Reusing Software: Issues and Research Directions," *IEEE Transactions on Software Engineering*, Vol. 21, No. 6, Jun. 1995, pp. 528-562.

[114] Mok, A., "Towards Mechanization of Real-Time System Design," A. van Tilborg and G. Koob, Eds., *Foundations of Real-Time Computing: Formal Specifications and Methods*, Kluwer Academic Publ., Boston, MA, 1991, pp. 1-37.

[115] Oman, P. and S. Pfleeger, Eds., *Applying Software Metrics*, IEEE Computer Society Press, Los Alamitos, CA, 1997.

[116] Parnas, D., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12, 1972, pp. 1053-1058.

[117] Parnas, D., P. Clements, and D. Weiss, "The Modular Structure of Complex Systems," In *Proceedings of the Seventh International Conference on Software Engineering*, Mar. 1984, pp. 408-417. Reprinted in *IEEE Transactions on Software Engineering*, SE-11, 1985, pp. 259-266.

[118] Paulk, M., C. Weber, and B. Curtis, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, Reading, MA, 1995.

[119] Pfleeger, S., "Assessing Measurement," *IEEE Software*, Vol. 14, No. 2, Mar./Apr. 1997, pp. 25-26.

[120] Ramachandran, M. and W. Fleisher, "Design for Large Scale Software Reuse: An Industrial Case Study," In *Proceedings of the Fourth International Conference on Software Reuse*, Apr. 23-26, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 104-111.

[121] Rechtin, E., "The Art of Systems Architecting," *IEEE Spectrum*, Oct. 1992.

[122] Rechtin, E., *Systems Architecting: Creating and Building Complex Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[123] Reinke, R., *Symbolic Clustering*, Ph.D. Dissertation, Report No. UIUCDCS-R-91-1704, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, 1991.

[124] Rowe, P., *Design Theory*, MIT Press, Cambridge, MA, 1987.

[125] Rubinstein, M., *Patterns of Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ, 1975.

[126] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall , Englewood Cliffs, NJ, 1988.

[127] Schneidewind, N., “Software Metrics Validation: Space Shuttle Flight Software Example,” *Annals of Software Engineering*, Vol. 1, 1995, pp. 287-309.

[128] Selic, B., G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, NY, 1994.

[129] Sha, L., R. Rajkumar, and M. Gagliardi, “Evolving Dependable Real Time Systems,” In *Proceedings of the IEEE Aerospace Applications Conference*, Aspen, CO, Feb. 3-10, IEEE Computer Society Press, New York, NY, 1996, pp. 335-346.

[130] Shaw, M., R. DeLine, and G. Zelesnik, “Abstractions and Implementations for Architectural Connections,” In *Proceedings of the Third International Conference on Configurable Distributed Systems*, Annapolis, MD, May 1996.

[131] Shaw, M. and D. Garlan, *Characteristics of Higher-Level Languages for Software Architecture*, Technical Report SEIR 94-23, Carnegie Mellon University, Pittsburgh, PA, Dec. 1994.

[132] Shaw, M. and D. Garlan, “What is Software Architecture,” Section 1.1 in *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Upper Saddle River, NJ, 1996, pp. 1-5.

[133] Shaw, M. and D. Garlan, “Architectural Styles,” Chapter 2 in *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Upper Saddle River, NJ, 1996, pp. 19-32.

[134] Shaw, M. and D. Garlan, “UniCon: A Universal Connector Language,” Section 8.1 in *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Upper Saddle River, NJ, 1996, pp. 183-190.

[135] Shaw, M., D. Garlan, and J. Galmes, *Experience with a Course on Architectures for Software Systems Part II: Educational Materials*, Technical Report SEIR 94-20, Carnegie Mellon University, Pittsburgh, PA, Aug. 1994.

[136] Shepperd, M., *Software Engineering Metrics Volume I: Measures and Validations*, McGraw-Hill Book Company Europe, Berkshire, England, 1993.

[137] Sitaraman, M., “Performance-Parameterized Reusable Software Components,” *International Journal of Software Engineering and Knowledge Engineering*, Vol. 2, No. 4, 1992, pp. 567-587.

[138] Sitaraman, M. and B. Weide, Eds., “Component-Based Software Engineering Using RESOLVE,” *ACM SIGSOFT Software Engineering Notes*, Vol. 19, No. 4, Oct. 1994, pp. 21-67.

[139] Sitaraman, M., L. Welch, and D. Harms, “On Specification of Reusable Software Components,” *International Journal of Software Engineering and Knowledge Engineering*, Special issue on Reusable Software Components, Vol. 3, No. 2, Jun. 1993, pp. 207-229.

[140] Software Engineering Institute, *Software Architecture and the Architecture Tradeoff Analysis Initiative*, Carnegie Mellon University, http://www.sei.cmu.edu/ata/ata_init.html as of Jan. 2001.

[141] Software Engineering Institute, *SEI Product Line Practice Publications*, Carnegie Mellon University, http://www.sei.cmu.edu/plp/plp_publications.html as of Jan. 2001.

[142] Software Engineering Institute, *Simplex Architecture*, Carnegie Mellon University, http://www.sei.cmu.edu/activities/str/descriptions/simplex_body.html as of Jan. 2001.

[143] Stankovic, J., “Good System Structure Features: Their Complexity and Execution Time Cost,” *Tutorial on Software Restructuring*, R. Arnold, Ed., IEEE Computer Society Press, Washington, D.C., 1986, pp. 36-48. Reprinted from *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, Jul. 1982, pp. 306-318.

[144] Stewart, D., *Real-Time Software Design and Analysis of Reconfigurable Multi-Sensor Based Systems*, Ph.D. Dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1994.

[145] Stewart, D. and P. Khosla, “The Chimera Methodology: Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects,” In *Proceedings of the IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, Dana Point, CA, Oct. 1994.

[146] Stewart, D., D. Schmitz, and P. Khosla, “The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications,” *IEEE Transactions on Systems, Man, and Cybernetics*, Nov./Dec. 1992, pp. 1282-1295.

[147] Szyperski, C., “Components and Objects Together,” *Software Development Online*, May 1999. The home web page for *Software Development Online* is http://www.sdmagazine.com as of Jan. 2001.

[148] Thomas, D., *The Design and Analysis of an Automated Design Style Selector*, Ph.D. Dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1977.

[149] Thompson, C., *Workshop on Compositional Software Architectures*, Workshop Report, Monterey, CA, Jan. 6-8, 1998. Representatives from the Object Management Group, the Defense Advanced Research Projects Agency, Microelectronics and Computer Technology Corporation, Object Services and Consulting, Inc., and academe met to discuss advances and challenges for the future in the development of component software architectures, non-functional system-wide properties, and web/ORB integration architectures. The home web page for the workshop is http://www.objs.com/workshops/ws9801/index.html as of Jan. 2001.

[150] Tomayko, J., “Forging a Discipline: An Outline History of Software Engineering Education,” *Annals of Software Engineering*, Vol. 6, Baltzer Science Publ., Bussum, The Netherlands, 1998, pp. 3-18.

[151] VanHilst, M. and D. Notkin, “Decoupling Change from Design”, *Software Engineering Notes*, Vol. 21, No. 6, Nov. 1996, pp. 58-69.

[152] von Mayrhauser, A. and A. Vans, “Program Comprehension During Software Maintenance and Evolution,” *Computer*, Vol. 28, No. 8, Aug. 1995, pp. 44-55.

[153] Ward, P. and S. Mellor, *Structured Development for Real-Time Systems*, Vol. 2: Essential Modeling Techniques, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[154] *WEBSTER’S II: New Riverside University Dictionary*, Riverside Publ. Co. (Houghton Mifflin), Boston, MA, 1984.

[155] Weide, B., W. Ogden, and M. Sitaraman, “Recasting Algorithms to Encourage Reuse,” *IEEE Software*, Sep. 1994, pp. 80-88.

[156] Weinberg, G., *Rethinking Systems Analysis and Design*, Dorset House, New York, NY, 1988.

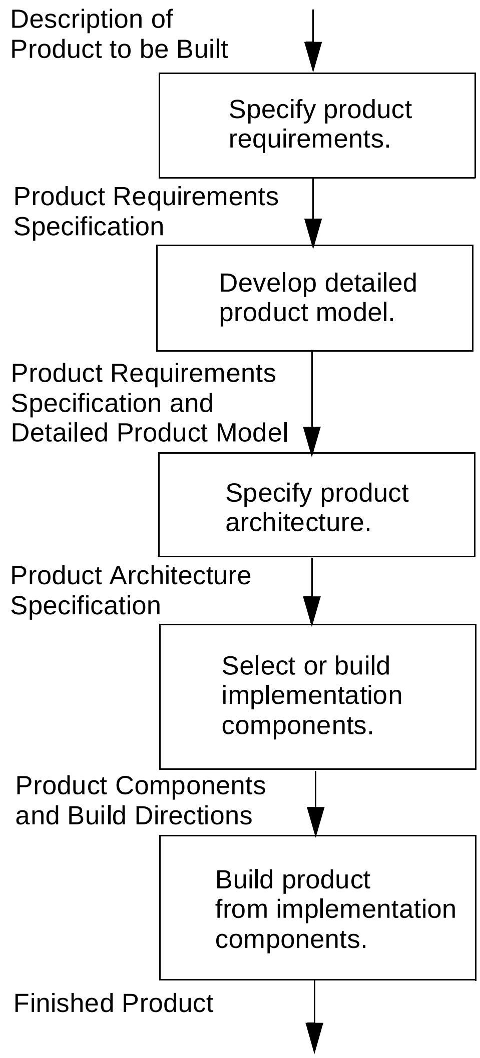
[157] Weiser, M. and B. Shneiderman, "Human Factors of Software Design and Development," *Tutorial on Software Restructuring*, R. Arnold, Ed., IEEE Computer Society Press, Washington, D.C., 1986, pp. 67-81.

[158] Westerberg, K., "Development of Software for Solving Systems of Linear Equations," Technical Report EDRC-05-35-89, Carnegie Mellon University, Pittsburgh, PA, 1989.

[159] Wortmann, J., *Object-Oriented Analysis for Advanced Flight Data Management*, Report No. 96-43, Technical University of Berlin, Germany, 1996.

[160] Yourdon, E. and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Yourdon Press (Prentice-Hall), Englewood Cliffs, NJ, 1979.

Appendix A Generic Process for Component-Based Product Development



Appendix B Statement of Work for the Microwave Oven Software

Your job is to design the software necessary to operate a microwave oven. Use an object-oriented style for your design, which should include the definition of the files, classes and methods, as well as pseudo-code for the methods. The pseudo-code can be at a higher level than actual code, but it should be precise enough for a reviewer to understand how much code would have to be written for each method and which functionality is assigned to each method.

The remainder of the statement of work provides background on the microwave oven's general structure, an analysis of the requirements that its software must fulfill, and design directives. (Note that the information does not necessarily correspond to an actual appliance. Do not assume that microwave ovens work precisely as described here.)

Microwave Oven: Structure

The microwave oven consists of a container in which food can be heated by penetrating it with microwave radiation. The radiation comes from one or more power sources. There are two primary controls for the oven: (1) a timer, which limits the duration of the radiation, and (2) a power sensor, which checks the level of radiation actually generated.

Users program the oven through a display and a keyboard, by which they can set the characteristics of the radiation (e.g., power and duration). Since radiation is harmful to people, a door sensor monitors whether the oven door is closed. If it is not closed, the microwave oven must shut off the radiation immediately.

Microwave Oven: Software

Electronics control the behavior of the oven (e.g., power sources). Software in turn controls the electronics.

The software should accomplish the following four functions, which build upon each other.

1. Drive the electronics: This part of the software is closest to the electronics. It gives instructions to the hardware and obtains status in return. For example, it tells the power source to increase power by one notch. (How this happens exactly does not have to concern you. From a high level viewpoint, the drivers use electronic connections between the processing unit that runs the software and the rest of the hardware.)
2. Control the electronics: This part of the software puts together the instructions to the hardware and passes them to the device drivers. For example, it decides when to send a command to the power source or to the timer.
3. Schedule a heating operation: This part of the software translates the user input into instructions that match the hardware's capabilities. For example, it calculates the time and power it takes to reheat two servings of pizza, if that is what the user has entered.
4. Manage the user interface: This part of the software directly receives user input through the keyboard and outputs messages on the display. For example, it understands that the button pushes 2,3,0 represent a duration of 2 minutes and 30 seconds. Button pushes 1,0,0,0 represent 10 minutes.

In your design, you must specify which files, classes, data and methods will be needed to fulfill each of these four functions. Table B.1 is an analysis of the specific requirements for each function. The analysis has a data part and a functionality part. It tells you the type of information is needed for each function and how the information is to be manipulated. Text enclosed by brackets provides design directives.

Table B.1 Analysis of the requirements for each basic function of the microwave oven software.

Basic Function	Functionality	Data
Manage User Interface	[When you design this part of the software, assume that you have at your disposal built-in methods called GET_KEYBOARD and PUT_DISPLAY to read the next button push from the keyboard and write text to the display. GET_KEYBOARD and PUT_DISPLAY are part of the SYSTEM class residing in the system.h file.] The user can request heating by the key sequences below. • Straight heating: The POWER button followed by a single digit to set the power level. Then the TIMER button followed by up to four digits setting a time between 0 seconds and 10 minutes 0 seconds. Programming concludes by the START button. The user interface must display an error message "ERROR" if the user attempts to set a power higher than 9, a duration longer than 10 minutes, or a number of seconds higher than 59. • Defrosting: The DEFROST button followed by up to four digits setting a weight between 0 ounces and 10 pounds, 0 ounces. Programming concludes by the START button. The user interface must display an error message "ERROR" if the user attempts to set a weight higher than 10 pounds or a number of ounces higher than 15. • Reheating: REHEAT button followed by up to two digits setting a number of servings between 0 and 25. Pushing the START button a first time continues the programming. It is followed by up to four digits setting a weight between 0 ounces and 10 pounds, 0 ounces. Programming concludes by pushing the START button a second time. The user interface must display an error message "ERROR" if the user attempts to request more than 25 servings, a total weight higher than 10 pounds, or a number of ounces higher than 15. The user interface should also enable the following functions. • The user can request the immediate halt to operation by hitting the STOP button. • The user interface must display a status message "DONE" whenever a heating or stop request completes normally. • The user interface must display an error message "FAIL" whenever a heating or stop request completes abnormally. In general: • The user interface must display an error message "ERROR" if the user attempts any meaningless key sequence (e.g., POWER followed by DEFROST). • Any error, STOP request, or opening of the door cancels all current programming. If those events occur, the oven cancels any partially completed requests; and the user must start over.	[The keyboard has buttons for the 10 digits, as well as START, STOP, DEFROST, REHEAT, POWER and TIMER buttons. The display has a length of 5 characters.] • Power Level: 0 through 9. • Duration: 0 minutes, 0 seconds through 10 minutes, 0 seconds. • Weight: 0 pounds, 0 ounces through 10 pounds, 0 ounces. • Number of Servings: 0 through 25. • Weight per Serving: 0 through 10 pounds, 0 ounces. • Heating Type: heat, defrost, or reheat.

Basic Function	Functionality	Data
Schedule Heating Operation	**Heat Request:** The user has requested a Heat function. Convert the button pushes into a Heat Operation consisting only of a power level and duration. The type of heat may be heat, defrost, or reheat. • If Heat Type is *heat*, pass on power level and duration to Heat Operation. Convert duration from minutes/seconds to ticks (1 second = 1 tick). • If Heat Type is *defrost*, use a DEFROST formula to convert Weight into a power level and duration. [Use the name DEFROST whenever you want to refer to this calculation in your design.] • If Heat Type is *reheat*, use a REHEAT formula to convert Servings and Weight per Serving into a power level and duration. [Use the name REHEAT whenever you want to refer to this calculation in your design.] If the user requests Stop, invoke the Stop Operation. Pass on to the user interface the status or error code received from the Heat or Stop Operations.	• Power level received through Heat Request and passed on to Heat Operation: 0 through 9. • Duration received through Heat Request: 0 minutes, 0 seconds through 10 minutes, 0 seconds. • Duration passed on to Heat Operation: 0 through 600 ticks. • Weight received through Heat Request: 0 pounds, 0 ounces through 10 pounds, 0 ounces. • Servings received through Heat Request: 0 through 25. • Weight per serving received through Heat Request: 0 through 10 pounds, 0 ounces. • Heat Type received through Heat Request: heat, defrost, or reheat.
Control Electronics	**Heat Operation:** Move the power up to the desired level and keep it there for the desired duration. Note that the electronics are very primitive. The algorithm is correspondingly simplistic. *Set timer to desired duration.* Then go through a feedback loop that works as described below. You can assume that each pass through the loop takes only fractions of a second. Initiate door status. Initiate power source status. Initiate power sensor read. Read power sensor to check actual power achieved. If power under desired level, increase power by 3 notches on all sources. If power over or at desired level, reduce power by 3 notches on all sources. Check timer - If expired, stop all power sources and return status code. Read status of all electronics - If malfunction or door open, stop all power sources and return an error code. Start over at the beginning of the loop. **Stop Operation:** Shut off all power sources immediately.	• Power level received by Heat Operation: 0 through 9. • Duration received by Heat Operation, provided to timer, or returned by timer: 0 through 600 ticks, where each tick presents 1 second. • Status code returned by Heat Operation and Stop Operation: blank or Heating Complete. • Error code returned by Heat Operation and Stop Operation: blank or Malfunction. • 4 identical power sources (numbered 1 through 4). • 1 power sensor. • 1 door sensor. • 1 timer. • Status returned by power source: fail or functioning. • Power level returned by power sensors: 0 through 9. • Status returned by power sensor: fail or functioning. • Status returned by door sensor: fail, open, or closed.

Basic Function	Functionality	Data
Drive Electronics	• Increase power by one notch for power source • Decrease power by one notch for power source • Shut off power source • Initiate power source status • Read status of power source • Initiate power sensor read • Read level on power sensor • Read status of power sensor • Initiate door status • Read status of door • Read status of door sensor • Set timer • Check timer status [When you design this part of the software, assume that you have at your disposal a built-in method called CALL_HARDWARE to request the functionality above. The CALL_HARDWARE method is part of the SYSTEM class residing in the system.h file. For example: • To increase power by one notch on power source 1, write the pseudo-code CALL_HARDWARE (power source 1, increase power by one notch). • To set the timer to 120 seconds, write CALL_HARDWARE (timer, set, 120). Also, you can assume that the electronics are automatically initialized when the microwave oven is connected to the power source.]	• Status returned by power source: fail or functioning. • Power level returned by power sensors: 0 through 9. • Status returned by power sensor: fail or functioning.. • Status returned by door sensor: fail, open, or closed. • Duration used by timer: 0 through 600 ticks.

Microwave Oven: Software - Additional Requirements for Product Evolution

We want to design the software so that it is not only suitable for this particular microwave but can be easily adapted for our entire microwave product line. Table B.2 lists the additional considerations to keep in mind for the product line.

Table B.2 Additional requirements for product evolution.

Functionality	Data
1. The product line will contain more sophisticated microwaves that allow not just the POWER-TIMER button sequence to program the heating but also a TIMER followed by POWER sequence. 2. Depending on the type of microwave oven, the defrost and reheat formulas may change. 3. Microwaves might be built with power sources whose behavior has to be controlled in a more sophisticated way than up or down three notches (e.g., the increase or decrease might have to be relative to how high the power already is or the power level might not stabilize). 4. Microwaves whose electronics respond faster would allow for different feedback loops. The sequences could be: • Initiate door status. Initiate power source status. Initiate power sensor read. Read power sensor to check actual power achieved. Adjust power if necessary. Read status of all electronics. Check timer for expiration. • Initiate door status. Initiate power source status. Initiate power sensor read. Read status of all electronics. Read power sensor to check actual power achieved. Adjust power if necessary. Check timer for expiration.	5. Some microwaves in the product line will be for the international market. Error and status messages, as well as weight measures, could be different in content and format (e.g., changes in the length and characters used). 6. More powerful microwaves will be part of the product line. They will have higher limits for weight, servings, and weight per serving. 7. Microwaves could be built with parts from different suppliers and thereby could require different configurations (e.g., 10 power sources and 4 power sensors). You can assume that these ovens would include a built-in function to determine an overall power level for different levels detected by multiple power sensors. 8. Microwaves could be built with a different type of power source. The parameters for CALL_HARDWARE could change format (e.g., instead of just specifying an increase or decrease by one notch of the power level, the driver software might have to specify an angle for the radiation). 9. Similar possibilities exist for the power sensor, door sensor, and timer.

Appendix C Change Complexity Values Across Alternative Sequences

Partition of Required Sequence <B,C,D,E,F,G,H,I>	Change Complexity for Alternative Sequence 1 <B,C,D,E,F,G,I,H >	Change Complexity for Alternative Sequence 2 <B,C,D,I,E,F,G,H>	Change Complexity Across All Alternative Sequences
1. <B><C><D><E><F><G><H><I>	8	8	16
2. <B><C><D><E><F><G><H,I>	2	9	11
3. <B><C><D><E><F><G,H><I>	9	7	16
4. <B><C><D><E><F><G,H,I>	3	9	12
5. <B><C><D><E><F,G><H><I>	7	7	14
6. <B><C><D><E><F,G><H,I>	2	8	10
7. <B><C><D><E><F,G,H><I>	9	6	15
8. <B><C><D><E><F,G,H,I>	4	9	13
9. <B><C><D><E,F><G><H><I>	7	7	14
10. <B><C><D><E,F><G><H,I>	2	8	10
11. <B><C><D><E,F><G,H><I>	8	6	14
12. <B><C><D><E,F><G,H,I>	3	5	8
13. <B><C><D><E,F,G><H><I>	6	6	12
14. <B><C><D><E,F,G><H,I>	2	5	7
15. <B><C><D><E,F,G,H><I>	9	5	14
16. <B><C><D><E,F,G,H,I>	5	5	10
17. <B><C><D,E><F><G><H><I>	7	9	16
18. <B><C><D,E><F><G><H,I>	2	10	12
19. <B><C><D,E><F><G,H><I>	8	8	16
20. <B><C><D,E><F><G,H,I>	3	5	8
21. <B><C><D,E><F,G><H><I>	6	8	14
22. <B><C><D,E><F,G><H,I>	2	6	8
23. <B><C><D,E><F,G,H><I>	8	7	15
24. <B><C><D,E><F,G,H,I>	4	6	10
25. <B><C><D,E,F><G><H><I>	6	9	15
26. <B><C><D,E,F><G><H,I>	2	10	12
27. <B><C><D,E,F><G,H><I>	7	8	15
28. <B><C><D,E,F><G,H,I>	3	6	9
29. <B><C><D,E,F,G><H><I>	5	9	14

Partition of Required Sequence <B,C,D,E,F,G,H,I>	Change Complexity for Alternative Sequence 1 <B,C,D,E,F,G,I,H >	Change Complexity for Alternative Sequence 2 <B,C,D,I,E,F,G,H>	Change Complexity Across All Alternative Sequences
30. <B><C><D,E,F,G><H,I>	2	6	8
31. <B><C><D,E,F,G,H><I>	9	9	18
32. <B><C><D,E,F,G,H,I>	6	6	12
33. <B><C,D><E><F><G><H><I>	7	7	14
34. <B><C,D><E><F><G><H,I>	2	8	10
35. <B><C,D><E><F><G,H><I>	8	6	14
36. <B><C,D><E><F><G,H,I>	3	5	8
37. <B><C,D><E><F,G><H><I>	6	6	12
38. <B><C,D><E><F,G><H,I>	2	7	9
39. <B><C,D><E><F,G,H><I>	8	5	13
40. <B><C,D><E><F,G,H,I>	4	6	10
41. <B><C,D><E,F><G><H><I>	6	6	12
42. <B><C,D><E,F><G><H,I>	2	7	9
43. <B><C,D><E,F><G,H><I>	7	5	12
44. <B><C,D><E,F><G,H,I>	3	5	8
45. <B><C,D><E,F,G><H><I>	5	5	10
46. <B><C,D><E,F,G><H,I>	2	5	7
47. <B><C,D><E,F,G,H><I>	8	4	12
48. <B><C,D><E,F,G,H,I>	5	5	10
49. <B><C,D,E><F><G><H><I>	6	9	15
50. <B><C,D,E><F><G><H,I>	2	10	12
51. <B><C,D,E><F><G,H><I>	7	8	15
52. <B><C,D,E><F><G,H,I>	3	6	9
53. <B><C,D,E><F,G><H><I>	5	8	13
54. <B><C,D,E><F,G><H,I>	2	7	9
55. <B><C,D,E><F,G,H><I>	7	7	14
56. <B><C,D,E><F,G,H,I>	4	7	11
57. <B><C,D,E,F><G><H><I>	5	9	14
58. <B><C,D,E,F><G><H,I>	2	10	12
59. <B><C,D,E,F><G,H><I>	6	8	14
60. <B><C,D,E,F><G,H,I>	3	7	10

Partition of Required Sequence ***<B,C,D,E,F,G,H,I>***	Change Complexity for Alternative Sequence 1 ***<B,C,D,E,F,G,I,H >***	Change Complexity for Alternative Sequence 2 ***<B,C,D,I,E,F,G,H>***	Change Complexity Across All Alternative Sequences
61. *<B><C,D,E,F,G><H><I>*	4	9	13
62. *<B><C,D,E,F,G><H,I>*	2	7	9
63. *<B><C,D,E,F,G,H><I>*	9	9	18
64. *<B><C,D,E,F,G,H,I>*	7	7	14
65. *<B,C><D><E><F><G><H><I>*	7	7	14
66. *<B,C><D><E><F><G><H,I>*	2	8	10
67. *<B,C><D><E><F><G,H><I>*	8	6	14
68. *<B,C><D><E><F><G,H,I>*	3	8	11
69. *<B,C><D><E><F,G><H><I>*	6	6	12
70. *<B,C><D><E><F,G><H,I>*	2	7	9
71. *<B,C><D><E><F,G,H><I>*	8	5	13
72. *<B,C><D><E><F,G,H,I>*	4	8	12
73. *<B,C><D><E,F><G><H><I>*	6	6	12
74. *<B,C><D><E,F><G><H,I>*	2	7	9
75. *<B,C><D><E,F><G,H><I>*	7	5	12
76. *<B,C><D><E,F><G,H,I>*	3	5	8
77. *<B,C><D><E,F,G><H><I>*	5	5	10
78. *<B,C><D><E,F,G><H,I>*	2	5	7
79. *<B,C><D><E,F,G,H><I>*	8	4	12
80. *<B,C><D><E,F,G,H,I>*	5	5	10
81. *<B,C><D,E><F><G><H><I>*	6	8	14
82. *<B,C><D,E><F><G><H,I>*	2	9	11
83. *<B,C><D,E><F><G,H><I>*	7	7	14
84. *<B,C><D,E><F><G,H,I>*	3	5	8
85. *<B,C><D,E><F,G><H><I>*	5	7	12
86. *<B,C><D,E><F,G><H,I>*	2	6	8
87. *<B,C><D,E><F,G,H><I>*	7	6	13
88. *<B,C><D,E><F,G,H,I>*	4	6	10
89. *<B,C><D,E,F><G><H><I>*	5	8	13
90. *<B,C><D,E,F><G><H,I>*	2	9	11
91. *<B,C><D,E,F><G,H><I>*	6	7	13

Partition of Required Sequence <B,C,D,E,F,G,H,I>	Change Complexity for Alternative Sequence 1 <B,C,D,E,F,G,I,H >	Change Complexity for Alternative Sequence 2 <B,C,D,I,E,F,G,H>	Change Complexity Across All Alternative Sequences
92. <B,C><D,E,F><G,H,I>	3	6	9
93. <B,C><D,E,F,G><H><I>	4	8	12
94. <B,C><D,E,F,G><H,I>	2	6	8
95. <B,C><D,E,F,G,H><I>	8	8	16
96. <B,C><D,E,F,G,H,I>	6	6	12
97. <B,C,D><E><F><G><H><I>	6	6	12
98. <B,C,D><E><F><G><H,I>	2	7	9
99. <B,C,D><E><F><G,H><I>	7	5	12
100. <B,C,D><E><F><G,H,I>	3	6	9
101. <B,C,D><E><F,G><H><I>	5	5	10
102. <B,C,D><E><F,G><H,I>	2	6	8
103. <B,C,D><E><F,G,H><I>	7	4	11
104. <B,C,D><E><F,G,H,I>	4	7	11
105. <B,C,D><E,F><G><H><I>	5	5	10
106. <B,C,D><E,F><G><H,I>	2	6	8
107. <B,C,D><E,F><G,H><I>	6	4	10
108. <B,C,D><E,F><G,H,I>	3	5	8
109. <B,C,D><E,F,G><H><I>	4	4	8
110. <B,C,D><E,F,G><H,I>	2	5	7
111. <B,C,D><E,F,G,H><I>	7	3	10
112. <B,C,D><E,F,G,H,I>	5	5	10
113. <B,C,D,E><F><G><H><I>	5	9	14
114. <B,C,D,E><F><G><H,I>	2	10	12
115. <B,C,D,E><F><G,H><I>	6	8	14
116. <B,C,D,E><F><G,H,I>	3	7	10
117. <B,C,D,E><F,G><H><I>	4	8	12
118. <B,C,D,E><F,G><H,I>	2	8	10
119. <B,C,D,E><F,G,H><I>	6	7	13
120. <B,C,D,E><F,G,H,I>	4	8	12
121. <B,C,D,E,F><G><H><I>	4	9	13
122. <B,C,D,E,F><G><H,I>	2	10	12

Partition of Required Sequence <B,C,D,E,F,G,H,I>	**Change Complexity for Alternative Sequence 1 <B,C,D,E,F,G,I,H >**	**Change Complexity for Alternative Sequence 2 <B,C,D,I,E,F,G,H>**	**Change Complexity Across All Alternative Sequences**
123. <B,C,D,E,F><G,H><I>	5	8	13
124. <B,C,D,E,F><G,H,I>	3	8	11
125. <B,C,D,E,F,G><H><I>	3	9	12
126. <B,C,D,E,F,G><H,I>	2	8	10
127. <B,C,D,E,F,G,H><I>	9	9	18
128. <B,C,D,E,F,G,H,I>	8	8	16

Appendix D Human Subjects Clearance Request

Carnegie Mellon University

Human Subjects Clearance Request

Date: 17 September 1999

CMU Protocol No. _____________
New Request __X__ Renewal ____

I. Overview of the Proposal for Experimentation

A. Project and Investigators

Principal investigators:

Daniel P. Siewiorek, Director
Human-Computer Interaction Institute
Buhl Professor of Electrical & Computer Engineering and Computer Science
Carnegie Mellon University
412-268-5228
dps@cs.cmu.edu

Carol L. Hoover, Doctoral Candidate
Dept. of Electrical & Computer Engineering
Carnegie Mellon University
412-268-6480
clh@cs.cmu.edu

Project Title: Analysis of an Experimental Approach for the Design of High-Assurance Software

Project Dates: From 1 October 1999 To 1 May 2000

Name of Experimenter: Carol L. Hoover

Brief Description of Research: To study the affects of using experimental methods for designing software on the resulting software artifacts.

B. Subjects, Benefits, and Risk Assessment

1. How many subjects will be used in this experiment? About 150-200

2. From what source do you plan to obtain subjects?

   Undergraduate and graduate students taking programming or software engineering courses offered by organizations such as the School of Computer Science (for example, 15-127, Introduction to Programming and Computer Science) and the Carnegie Institute of Technology (for example, 12-741, Advanced Programming Concepts in Computer-Aided Engineering).

3. Is there any benefit gained by the subject for participating?

   Pay and opportunity to learn innovative techniques useful for the development of well-structured software.

4.	Will the subjects include any of the following?	No
	Fetuses	Mentally Retarded
	Hospitalized Patients	Minors
	Institutionalized Patients	Pregnant Women
	Mentally Disabled	Prisoners
5.	Degree of physical risk to subjects:	Negligible
6.	Degree of psychological risk to subjects:	Negligible

II. Abstract of the Proposed Experimentation

The purpose of the proposed experimentation is to determine the affects of applying an experimental software design approach on the quality of the resulting software artifacts. The experimental approach consists of a way of thinking (rationale) about software quality properties as well as precise and step-wise directions (design methods) to guide the human designer in the development of a software design that will achieve the desired quality properties. Example software quality properties are ease of change and adaptability.

The proposed experimentation will validate the effectiveness of the experimental approach. The experimenters will measure effectiveness by whether or not and to what extent the new approach helps software designers to more effectively, in comparison to software design without using the experimental approach, to develop software designs that achieve target properties. Preliminary investigation has shown that the new approach when applied by the experimenter resulted in software designs that achieved target quality properties. This experimentation will determine the usefulness of the experimental approach across many more software designers and across factors that, in addition to the design approach, may also affect the human designer’s capability to produce a design that achieves target quality properties

III. Experimental Treatment

We plan a series of three to four experiments with variation in the level of knowledge and skill as well as task complexity (software design assignment) across the experiments. Within each experiment, we will hold constant as much as is feasible the level of knowledge and skill of the subjects as well as the task complexity. The goal is to determine not only if the experimental approach positively affects the ability of the subjects to produce “good” software designs but also if the affect varies depending on the software design expertise of the subjects and the complexity of the task.

We will recruit subjects from undergraduate and graduate students taking programming or software engineering courses offered by organizations such as the School of Computer and the Carnegie Institute of Technology. To control the experimental factors that we discussed above, we will recruit participants from one course per experiment. Participation will be voluntary. To manage the time spent on the assignment while maximizing the benefit to the participants, we will prepare assignments that are appropriate for the educational objectives of the target courses. Though all students in these courses will complete the assignments that we are using for the experiments, participation in the experiments will be voluntary. We will design the experimental treatment so that is does not significantly increase the time that the participants need to complete their assignments.

Within each experiment, we will organize the pool of subjects into control and experimental groups. Both groups will perform the same software design assignment and will receive the standard course instruction for the assignment. In addition, the experimental groups will receive instruction on how to apply the experimental approach that is designed to complement the course instruction. The experimental groups will use the experimental approach along with their standard course instruction to complete the assignment. Control subjects will have the opportunity to learn about the technique after the completion of the experiment.

The experimenter will evaluate the software artifacts (software designs and code) that both the control and experimental subjects produce. The experimenter will develop and apply metrics for measuring the quality of these artifacts. The course instructors whose students are participating in the experiments may review the experimental results but will not use the results to evaluate student performance for the assignment.

IV. Consent Form

The experimenter will give copies of the attached description of the experiment (Call for Participation in a Research Study) to potential participants in the experiment. Participation is optional. The instructors of courses from which subjects are recruited will not penalize those students who chose not to participant. The students who chose to participate in the experiments will read, comprehend and sign copies of the attached consent form. Both the Call for Participation in a Research Study and consent forms explain that the subject may stop the experiment at any time if he/she is uncomfortable with his/her participation.

V. Confidentiality

The experimenter will evaluate the software designs produced by the subjects with respect to the test metrics. Only the experimenter and the instructors whose students are participating in the experiment will have access to the software designs that the student subjects produce and the evaluation of each individual design. The investigators will report or publish only group data. They will not report or publish the names of the subjects and will discard the names of the subjects when they are no longer needed.

VI. Risk/Benefit Analysis

The risks to the subjects are negligible and no more than that expected for a programming class assignment. The subjects will have 1-4 weeks to perform the tasks during time intervals set by the subjects. The tasks (software design and programming assignments) will take no more than the amount of time required for a similar type of class assignment. The subjects may choose to discontinue their participation at any time.

In addition to a payment (*type and amount currently under consideration by the principal investigators*[1]), the subjects will have the opportunity to learn about a new software design approach that complements and enhances techniques used widely by professional software developers. Control subjects will have the opportunity to learn about the technique after the completion of the experiment. The experimental approach may improve widely used software design techniques. Therefore, participants have the opportunity to contribute to the progress of their field of study.

[1]The principal investigators will determine an appropriate payment and will indicate this on the actual consent forms and Call for Participation in a Research Study.

Appendix E Call for Participation in a Research Study

Analysis of an Experimental Approach for the Design of High-Assurance Software

You have the unique opportunity to learn about a new design approach by participating in a research study. The purpose of the design approach is to help the software developer design “good” software. In addition to working correctly, good software has other properties that increase its value not only to the user but also to the producer of the software. In the software engineering community, we designate such properties as quality attributes. High-assurance computing requires that the software perform reliably as well as correctly on demand, often in critical environments such as air traffic control and nuclear power plant management. The design of good software is especially important for these types of applications. Carol L. Hoover, a doctoral student in the Department of Electrical and Computer Engineering, has developed the new design approach and has tested it on a small scale. She is planning an experiment to test the effectiveness of her experimental approach across a larger group of software developers and is looking for students to participate in her experiment.

As a participant in this study, you will complete a software design (*and programming for some experiments*[1]) assignment. The amount of time required for the assignment will be similar to that required for other programming assignments in your class. Likewise, the difficulty of the assignment will be similar to your current class assignments. Your instructor will count the software design that you develop as a regular class assignment, while the experimenter will examine your design (*and program for some experiments*[1]) only for experimental purposes. Participation is voluntary, but students who do not participate in the study will do the same assignment because it is important to the educational objectives for your course. Non-participants will not submit their software artifacts to the experimenter. Your course instructor will not use the experimental data to evaluate your performance on the assignment.

For your participation in the experiment, the investigators will give you (*payment type and amount still being considered by the principal investigators*[2]) as well as the opportunity to learn a new way to design software. The experiment poses minimal risk, no more than that which you may incur in completing your class assignments. If you are uncomfortable with the experiment for whatever reason, you may end your participation at any time without penalty. The investigators will not use your name or reveal your identity in any description or publication of the research.

At the end of the experiment, you will be able to obtain a full description of the study, including a discussion of its scientific purpose and results. You can address any questions that you may have about this research now or in the future to:

Carol L. Hoover Professor	Daniel P. Siewiorek, Director
Dept. of Electrical & Computer Engineering	Human-Computer Interaction Institute
Carnegie Mellon University	Carnegie Mellon University
clh@cs.cmu.edu	dps@cs.cmu.edu
412-268-6480	412-268-5228

Participation in the experiment is not only a unique learning opportunity but is also a way to contribute to the progress of your field of study. In addition, applying the new technique may actually be fun!

We hope that you will volunteer!
Carol L. Hoover and Professor Daniel P. Siewiorek

[1]Some experiments will involve software design tasks and others programming tasks. The Call for Participation form will state the type of task to be done for the experiment.

[2]The principal investigators will determine an appropriate payment and will replace the italicized text with the type of payment.

Appendix F Consent to Participate in a Research Study

Department of Electrical and Computer Engineering

Carnegie Mellon University

Title of Study: Analysis of an Experimental Approach for the Design of High-Assurance Software

The purpose of this research is to study the affects of using experimental methods for designing software on the resulting software artifacts (software designs and code).

I agree to participate in the above named research study. Carol L. Hoover will explain to me the procedures of the related experiment.

For my participation in the experiment, the investigators will give me (*payment type and amount still being considered by the principal investigator*[1]) as well as the opportunity to learn a new way to design software. I understand that the experiment poses minimal risk. I am aware that as a participant in this study I will complete a software design (*or programming*[2]) task and that the time to perform the task will be similar to that needed for a class assignment. I realize that my participation is voluntary and that I may end my participation at any time without penalty. I am aware that only the experimenter and my course instructor will have access to information that I provide for the experiment. I understand that my course instructor will not use the experimental results to evaluate my performance in the course. The investigators will not use my name or reveal my identity in any description or publication of the research.

At the end of the experiment, I will be able to obtain a full description of the study, including a discussion of its scientific purpose and results. I can address any future questions that I may have about this research to:

Carol L. Hoover
Dept. of Electrical & Computer Engineering
Carnegie Mellon University
clh@cs.cmu.edu
412-268-6480

Professor Daniel P. Siewiorek, Director
Human-Computer Interaction Institute
Carnegie Mellon University
dps@cs.cmu.edu
412-268-5228

I can direct any questions that I may have about my rights as a research participant to:

Susan Burkett
Associate Provost
Warner Hall 402
Carnegie Mellon University
412-268-8746

By signing this form, I agree to participate in this study. I acknowledge that a copy of this form has been given to me.

____

Printed Name

____________________________________ ____________________________

Signature Date

[1]The principal investigators will determine an appropriate payment and will replace the italicized text with the type of payment.

[2]Some experiments will involve software design tasks and others programming tasks. The consent form will state the type of task to be done for the experiment.

Appendix G Receipt of Compensation Form

Carnegie Mellon University

Receipt of Compensation for Participation in Experiment

Analysis of an Experimental Approach for the Design of High-Assurance Software

CMU Protocol Number: HS99-119

I verify that I received cash in the amount of $ _______________ in payment for my participation in the above named experiment.

____

Participant's Name (Please print.) Participant's Signature

____

Social Security Number Date

Appendix H Benchmark Design for the Microwave Oven Software

The benchmark design consists of the following parts.

- Conventions for defining the elements of the design.
- Definition of the design elements with reference to the corresponding functional requirements.

Conventions for Defining Design Elements:

To show how the research approach applies to different design styles, the author defines each design element with respect to object-oriented and structured designs, two popular design approaches. To eliminate confusion, the names of structured design elements appear in parentheses. The author defines the names of the design elements according to the following convention. The reader may prefer another convention.

- Class, file, or directory names consist of one or more English words whose first letters are capitalized. Underscores connect the words (e.g. Power_Source).
- Method or routine names consist of lowercase English words connected by underscores. A set of parentheses follow each method or routine name. The parentheses may enclose interface parameters (e.g. initiate_power_source_status(powerSource)).

  The exception are the names of predefined methods or routines which consist of English words that contain capital letters (e.g. PUT_DISPLAY() and CALL_HARDWARE()).
- Parameter names consist of lowercase English words whose first letters are capitalized, except for the first word. Each name appears as a string of concatenated words (e.g. powerSource).
- The names of data types should conform to the following rules.
  - Constants or predefined types contain English words that are capitalized and connected by underscores.
  - User-defined types consist of lowercase English words whose first letters are capitalized. Each name appears as a string of concatenated words (e.g. StatusCode).
  - Naming variables follows the convention for parameters.

Definition of the Design Elements:

The design elements (components) which handle the *Drive Electronics* software follow.

- Power_Source class
- Power_Sensor class
- Door_Sensor class
- Timer Class

```
class (file) Power_Source::/* Change signature for component: PSRC (RCG1). */
    SYSTEM sysObj;          /* instance of SYSTEM class */

    method (routine): StatusCode   initiate_power_source_status(powerSource){
        return sysObj.CALL_HARDWARE(powerSource,"initiate status");
    }   /* Return FAIL or FUNCTIONING. */

    method (routine): StatusCode   read_power_source_status(powerSource){
```

```
        return sysObj.CALL_HARDWARE(powerSource,“read status”);
    }   /* Return FAIL or FUNCTIONING. */

    method (routine): StatusCode   increase_power(powerSource){
        return sysObj.CALL_HARDWARE(powerSource,“increase power”);
    }   /* Return FAIL or FUNCTIONING. */

    method (routine): StatusCode   decrease_power(powerSource){
        return sysObj.CALL_HARDWARE(powerSource,“decrease power”);
    }   /* Return FAIL or FUNCTIONING. */

    method (routine):StatusCode   shut_off_power_source(powerSource){
        return sysObj.CALL_HARDWARE(powerSource,“shut off power”);
    }   /* Return FAIL or FUNCTIONING. */
endclass Power_Source

class (file) Power_Sensor:: /* Change signature for component: PSNSR (RCG1). */
    SYSTEM sysObj;          /* instance of SYSTEM class */

    method (routine): StatusCode   initiate_power_sensor_read(powerSensor){
        return sysObj.CALL_HARDWARE(powerSensor,“initiate read”);
    }   /* Return FAIL or FUNCTIONING. */

    method (routine): StatusCode   read_power_sensor_level(powerSensor,powerLevel){
        return sysObj.CALL_HARDWARE(powerSensor,
            “read level”,powerLevel);
    }   /* Return FAIL or FUNCTIONING and power level. */

    method (routine): StatusCode   read_power_sensor_status(powerSensor){
        return sysObj.CALL_HARDWARE(powerSensor,“read status”);
    }   /* Return FAIL or FUNCTIONING. */
endclass Power_Sensor

class (file) Door_Sensor::/* Change signature for component: DSNSR (RCG1). */
    SYSTEM sysObj;          /* instance of SYSTEM class */

    method (routine): StatusCode   initiate_door_status(doorSensor){
        return sysObj.CALL_HARDWARE(doorSensor,“initiate status”);
    }   /* Return FAIL or FUNCTIONING. */

    method (routine): StatusCode   read_door_status(doorSensor){
        return sysObj.CALL_HARDWARE(doorSensor,“read door status”);
    }   /* Return OPEN, CLOSED, or FAIL. */

    method (routine): StatusCode   read_door_sensor_status(doorSensor){
        return sysObj.CALL_HARDWARE(doorSensor,
            “read door sensor status”);
    }   /* Return FAIL or FUNCTIONING. */
endclass Door_Sensor

class (file) Timer::/* Change signature for component: TIMER (RCG1). */
    SYSTEM sysObj;          /* instance of SYSTEM class */

    method (routine): Integer   convert_to_ticks(numberOfMinutes,numberOfSeconds){
```

```
        return numberOfMinutes*60 + numberOfSeconds;
    }   /* Return number of ticks. */

    method (routine):set_timer(numberOfTicks){
        return sysObj.CALL_HARDWARE(timer,"set",numberOfTicks);
    }   /*Return FAIL or FUNCTIONING. */

    method (routine): StatusCode   check_timer_status(){
        return sysObj.CALL_HARDWARE(timer,"check status");
    }   /* Return EXPIRED, FAIL, or FUNCTIONING. */
endclass Timer
```

The design elements (components) which handle the *Control Electronics* software follow.

- Increase_Decrease_Power class
- Stop_All_Power_Sources class
- Initiate_Status_All_Power_Sources class
- Hardware_Configuration class
- Control_Electronics class

class (file) Increase_Decrease_Power:: /* Change signature for component: **CPSRC**.*/

```
    method (routine):StatusCode   increase_all_power_sources(powerSources,powerSourceObj,
        errorCode){
        StatusCode statusCode = FUNCTIONING;
        Integer count1 = 1, count2 = 1;
        errorCode = BLANK;
        while (statusCode is FUNCTIONING) and (count1 <= NUMBER_POWER_SOURCES) do{
            while (statusCode is FUNCTIONING) and (count2 <= NUMBER_OF_NOTCHES) do{
                statusCode = powerSourceObj.increase_power(powerSources[count1]);
                if statusCode is not FUNCTIONING then{
                    errorCode = INCREASE_POWER_SOURCE_FAILURE;
                }
                count2 = count2 + 1;
            }
            count1 = count1 + 1;
        }
        return statusCode;
    }   /* Return status code of FUNCTIONING or FAIL and error code of BLANK or
        INCREASE_POWER_SOURCE_FAIL. */

    method (routine):StatusCode   decrease_all_power_sources(powerSources,powerSourceObj,
        errorCode){
        StatusCode statusCode = FUNCTIONING;
        Integer count1 = 1, count2 = 1;
        errorCode = BLANK;
        while (statusCode is FUNCTIONING) and (count1 <= NUMBER_POWER_SOURCES) do{
            while (statusCode is FUNCTIONING) and (count2 <= NUMER_OF_NOTCHES) do{
                statusCode = powerSourceObj.decrease_power(powerSources[count1]);
                if statusCode is not FUNCTIONING then{
                    errorCode = DECREASE_POWER_SOURCE_FAILURE;
                }
                count2 = count2 + 1;
            }
```

```
            count1 = count1 + 1;
        }
        return statusCode;
    }   /* Return status of FUNCTIONING or FAIL and error code of BLANK or
        DECREASE_POWER_SOURCE_FAIL. */
endclass Increase_Decrease_Power
```

class (file) Stop_All_Power_Sources::

```
    method (routine):StatusCode    stop_all_power_sources(powerSources,powerSourceObj,
        errorCode){
        StatusCode statusCode = FUNCTIONING;
        Integer count = 1;
        errorCode = BLANK;
        while (statusCode is FUNCTIONING) and (count <= NUMBER_POWER_SOURCES) do{
            statusCode = powerSourceObj.shut_off_power_source(powerSources[count]);
            if statusCode is not FUNCTIONING then{
                errorCode = SHUT_OFF_POWER_SOURCE_FAILURE;
            }
            count = count + 1;
        }
        return statusCode;
    }   /* Return status of FUNCTIONING or FAIL and errorCode of BLANK or
        SHUT_OFF_POWER_SOURCE_FAILURE. */
endclass Stop_All_Power_Sources
```

class (file) Initiate_Status_All_Power_Sources::

```
    method (routine):StatusCode    initiate_status_all_power_sources(powerSources,powerSourceObj,
        errorCode){
        StatusCode statusCode = FUNCTIONING;
        Integer count = 1;
        errorCode = BLANK;
        while (statusCode is FUNCTIONING) and (count <= NUMBER_POWER_SOURCES) do{
            statusCode = powerSourceObj.initiate_power_source_status(powerSources[count]);
            if statusCode is not FUNCTIONING then{
                errorCode = INITIATE_POWER_SOURCE_STATUS_FAILURE;
            }
            count = count + 1;
        }
        return statusCode;
    }   /* Return status code of FUNCTIONING or FAIL and error code of BLANK or
        INITIATE_POWER_SOURCE_STATUS_FAILURE. */
endclass Initiate_Status_All_Power_Sources
```

class (file) Hardware_Configuration:: /* Change signature for component: **CHD**, **FDBL**.*/

```
    Power_Source powerSourceObj;      /* instance of Power_Source class */
    Power_Sensor powerSensorObj;      /* instance of Power_Sensor class */
    Door_Sensor doorSensorObj;        /* instance of Door_Sensor class */
    Timer timerObj;                   /* instance of Timer class */
    Stop_All_Power_Sources stopObj;   /* instance of Stop_All_Power_Sources class */

    PowerSources powerSources;
    PowerSensors powerSensors;
    DoorSensors doorSensors;
```

```
method (routine):StatusCode    read_all_devices_status(errorCode){
    SYSTEM sysObj;              /* instance of SYSTEM class */
    StatusCode statusCode = FUNCTIONING;
    Integer count = 1;
    errorCode = BLANK;
    if sysObj.GET_KEYBOARD is STOP then{
        statusCode = EXPIRED;
    }
    while (statusCode is FUNCTIONING) and (count <= NUMBER_POWER_SOURCES) do{
        statusCode = powerSourceObj.read_power_source_status(powerSources[count]);
        if statusCode is not FUNCTIONING then{
            errorCode = POWER_SOURCE_STATUS_FAILURE;
        }
        count = count + 1;
    }
    count = 1;
    while (statusCode is FUNCTIONING) and (count <= NUMBER_POWER_SENSORS) do{
        statusCode = powerSensorObj.read_power_sensor_status(powerSensors[count]);
        if statusCode is not FUNCTIONING then{
            errorCode = POWER_SENSOR_STATUS_FAILURE;
        }
        count = count + 1;
    }
    count = 1;
    while (statusCode is FUNCTIONING) and (count <= NUMBER_DOOR_SENSORS) do{
        statusCode = doorSensorObj.read_door_sensor_status(doorSensors[count]);
        if statusCode is not FUNCTIONING then{
            errorCode = DOOR_SENSOR_STATUS_FAILURE;
        }else{
            statusCode = doorSensorObj.read_door_status(doorSensors[count]);
            if statusCode is OPEN then{
                errorCode = DOOR_OPEN;
            }else if statusCode is CLOSED then{
                statusCode = FUNCTIONING;
            }else{
                errorCode = DOOR_STATUS_FAILURE;
            }
        }
        count = count + 1;
    }
    if statusCode is FUNCTIONING then{
        statusCode = timerObj.check_timer_status();
        if statusCode is not (FUNCTIONING or EXPIRED) then{
            errorCode = TIMER_STATUS_FAIL;
        }
    }
    return statusCode;
}    /* end read_all_devices_status -- Return status code of FUNCTIONING, FAIL, EXPIRED,
    OPEN and error code of BLANK, POWER_SOURCE_STATUS_FAILURE,
    POWER_SENSOR_STATUS_FAILURE, DOOR_SENSOR_STATUS_FAILURE,
    DOOR_OPEN, DOOR_STATUS_FAILURE, or TIMER_STATUS_FAIL. */
```

```
method (routine):StatusCode    control_flow_component_C1(errorCode){
    Initiate_Status_All_Power_Sources powerStatusObj;
                                        /* instance of Initiate_Status_All_Power_Sources class */
    StatusCode statusCode = FUNCTIONING;
    errorCode = BLANK;
    statusCode = doorSensorObj.initiate_door_status(doorSensor[1]);
    if statusCode is not FUNCTIONING then{
        errorCode = INITIATE_DOOR_STATUS_FAILURE;
    }else{
        statusCode = powerStatusObj.initiate_status_all_power_sources(powerSources,
            powerSourceObj,errorCode);
        if statusCode is FUNCTIONING then{
            statusCode = powerSensorObj.initiate_power_sensor_read(powerSensor[1]);
            if statusCode is not FUNCTIONING then{
                errorCode = INITIATE_POWER_SENSOR_READ_FAILURE;
            }
        }
    }
    return statusCode;
}   /* end control_flow_component_C1: Return status code of FUNCTIONING or FAIL and
    error code of BLANK, INITIATE_DOOR_STATUS_FAILURE,
    INITIATE_POWER_SOURCE_STATUS_FAILURE, or
    INITIATE_POWER_SENSOR_READ_FAILURE. */

method (routine):StatusCode    control_flow_component_C2(errorCode){
    Increase_Decrease_Power controlPowerObj;/* instance of Increase_Decrease_Power class */
    StatusCode statusCode = FUNCTIONING;
    PowerLevel powerLevel;
    errorCode = BLANK;
    statusCode = powerSensorObj.read_power_sensor_level(powerSensor[1],powerLevel);
    if statusCode is not FUNCTIONING then{
        errorCode = READ_POWER_SENSOR_LEVEL_FAILURE;
    }else if powerLevel under desired level then{
        statusCode = controlPowerObj.increase_all_power_sources(powerSources,
            powerSourceObj,errorCode);
        if statusCode is FUNCTIONING then{
            if powerLevel over desired level then{
                statusCode = controlPowerObj.decrease_all_power_sources(powerSources,
                    powerSourceObj,errorCode);
            }
        }
    }
    return statusCode;
}   /* end control_flow_component_C2: Return status code of FUNCTIONING or FAIL and
    error code of BLANK, READ_POWER_SENSOR_LEVEL_FAILURE,
    INCREASE_POWER_SOURCE_FAILURE,
    DECREASE_POWER_SOURCE_FAILURE. */

method (routine):StatusCode    control_flow_component_C3(errorCode){
    StatusCode statusCode = FUNCTIONING, statusCode1 = FUNCTIONING;
    ErrorCode errorCode1 = BLANK;
    errorCode1 = BLANK;
    statusCode = timerObj.check_timer_status();
    if statusCode is EXPIRED then{
```

```
            statusCode = stopObj.stop_all_power_sources(powerSources,powerSourceObj,
                errorCode);
        }else if statusCode is FUNCTIONING{
            statusCode = read_all_devices_status(errorCode);
            if statusCode is not FUNCTIONING then{
                statusCode1 = stopObj.stop_all_power_sources(powerSources,powerSourceObj,
                    errorCode2);
            }
        }else{
            errorCode = CHECK_TIMER_STATUS_FAILURE;
        }
        return statusCode;
    }   /* end control_flow_component_C3: Return status code of FUNCTIONING, FAIL, EXPIRED,
        or OPEN and error code of SHUT_OFF_POWER_SOURCE_FAILURE,
        POWER_SOURCE_STATUS_FAILURE, POWER_SENSOR_STATUS_FAILURE,
        DOOR_SENSOR_STATUS_FAILURE, DOOR_OPEN, or DOOR_STATUS_FAILURE,
        and CHECK_TIMER_STATUS_FAILURE. */

    method (routine):StatusCode    heat_operation(powerLevel,duration,errorCode){
        StatusCode statusCode = FUNCTIONING, statusCode1 = FUNCTIONING;
        ErrorCode errorCode1 = BLANK;
        errorCode1 = BLANK;
        statusCode = timerObj.set_timer(duration);
        if statusCode is not FUNCTIONING then{
            errorCode = SET_TIMER_FAILURE;
        }
        while (statusCode is FUNCTIONING) and (errorCode is BLANK){
            statusCode = control_flow_component_C1(errorCode);
            if statusCode is FUNCTIONING then{
                statusCode = control_flow_component_C2(errorCode);
                if statusCode is FUNCTIONING then{
                    statusCode = control_flow_component_C3(errorCode);
                }
            }
        }
        if statusCode is not EXPIRED or OPEN then{
            statusCode1 = stopObj.stop_all_power_sources(powerSources,powerSourceObj,
                errorcode1);
        return statusCode;
    }   /* end heat_operation: Return status code of EXPIRED, OPEN, or FAIL and error code
        of BLANK, INITIATE_DOOR_STATUS_FAILURE,
        INITIATE_POWER_SOURCE_STATUS_FAILURE,
        INITIATE_POWER_SENSOR_READ_FAILURE,
        READ_POWER_SENSOR_LEVEL_FAILURE,
        INCREASE_POWER_SOURCE_FAILURE, DECREASE_POWER_SOURCE_FAILURE,
        SHUT_OFF_POWER_SOURCE_FAILURE, POWER_SOURCE_STATUS_FAILURE,
        POWER_SENSOR_STATUS_FAILURE, DOOR_SENSOR_STATUS_FAILURE,
        DOOR_OPEN, or DOOR_STATUS_FAILURE, or
        CHECK_TIMER_STATUS_FAILURE. */
endclass Hardware_Configuration
```

```
class (file) Control_Electronics::
    method (routine):StatusCode control_electronics(heatRequest,powerLevel,duration,errorCode){
        Hardware_Configuration hardwareObj;  /* instance of Hardware_Configuration class */
        StatusCode statusCode = BLANK;
        errorCode = BLANK;
        if heatRequest is HEAT_OPERATION then{
            statusCode = hardwareObj.heat_operation(powerLevel,duration,errorCode);
            if (statusCode is EXPIRED) and (errorCode is BLANK) then{
                statusCode = HEATING_COMPLETE;
            }else{
                statusCode = BLANK;
                errorCode = MALFUNCTION;
            }
        }
        else if heatRequest is STOP_OPERATION then{
            statusCode = hardwareObj.stopObj.stop_all_power_sources(hardwareObj.powerSources,
                hardwareObj.powerSourceObj,errorCode);
            if (statusCode is FUNCTIONING) and (errorCode is BLANK) then{
                statusCode = HEATING_COMPLETE;
            }else{
                statusCode = BLANK;
                errorCode = MALFUNCTION;
            }
        }
        else{
            errorCode = INCORRECT_HEATING_TYPE;
        }
        return statusCode;
    }   /* Return status code of BLANK or HEATING_COMPLETE and error code of BLANK
        or MALFUNCTION. */
endclass Control_Electronics
```

The design elements which handle the *Manage User Interface* software follow.

- Straight_Heat class
- Defrost class
- Reheat class
- Manage_User_Interface Class

```
class (file) Straight_Heat:: /* Change signature for component: HBSQ. */
    method (routine):StatusCode straight_heat(powerLevel, duration){
        SYSTEM sysObj;                  /* instance of SYSTEM class */
        Timer timerObj;                 /* instance of Timer class */
        StatusCode statusCode = OK;
        KeyType key = NO_KEY_PRESSED;
        Integer value1 = 0, value2 = 0, count = 1;
        powerLevel = 0;
        duration = 0;
        key = sysObj.GET_KEYBOARD();
        if not (is_digit(key)) then{
            statusCode = ERROR;
        }else{
```

```
            sysObj.PUT_DISPLAY(convert_to_text(key));
            powerLevel = convert_to_integer(key);
            if not (MIN_POWER <= powerLevel <= MAX_POWER) then{
                statusCode = ERROR;
            }else{
                key = sysObj.GET_KEYBOARD();
                if not (key is TIMER) then{
                    statusCode = ERROR;
                }else{
                    key = sysObj.GET_KEYBOARD();
                    while (statusCode is OK and key is not START and count <= 4) do{
                        if not (is_digit(key)) then{
                            statusCode = ERROR;
                        }else{
                            sysObj.PUT_DISPLAY(convert_to_text(key));
                            if count <= 2 then{
                                value1 = value1*10 + convert_to_integer(key);
                            }else{
                                value2 = value2*10 + convert_to_integer(key);
                            }
                            key = sysObj.GET_KEYBOARD();
                            count = count + 1;
                        }
                    }
                    if not (key is START) then{
                        statusCode = ERROR;
                    }else{
                        if count <= 3 then{
                            value2 = value1;
                            value1 = 0;
                        }
                        if not (MIN_SECONDS <= value2 <= MAX_SECONDS) then{
                            statusCode = ERROR;
                        }else{
                            duration = timerObj.convert_to_ticks(value1,value2);
                            if not (MIN_TICKS <= duration <= MAX_TICKS) then{
                                statusCode = ERROR;
                            }
                        }
                    }
                }
            }
            return statusCode;
        }   /* end straight_heat: Return status of OK or ERROR, powerLevel, and duration. */
    endclass Straight_Heat
```

class (file) Defrost::

```
    method (routine):StatusCode defrost(powerLevel, duration){
        SYSTEM sysObj;                  /* instance of SYSTEM */
        StatusCode statusCode = OK;
        KeyType key = NO_KEY_PRESSED;
        Integer value1 = 0, value2 = 0, count = 1;
        powerLevel = 0;
        duration = 0;
```

```
        key = sysObj.GET_KEYBOARD();
        while (statusCode is OK and key is not START and count <= 4) do{
            if not (is_digit(key)) then{
                statusCode = ERROR;
            }else{
                sysObj.PUT_DISPLAY(convert_to_text(key));
                if count <= 2 then{
                    value1 = value1*10 + convert_to_integer(key);
                }else{
                    value2 = value2*10 + convert_to_integer(key);
                }
                key = sysObj.GET_KEYBOARD();
                count = count + 1;
            }
        }
        if not (key is START) then{
            statusCode = ERROR;
        }else{
            if count <= 3 then{
                value2 = value1;
                value1 = 0;
            }
            if not (MIN_OUNCES <= value1 <= MAX_OUNCES) then{
                statusCode = ERROR;
            }else{
                /* Convert pounds to ounces. */
                value1 = CONVERT_LARGER_UNITS_TO_SMALLER_UNITS * value1 + value2;
                if not (MIN_WEIGHT <= value1 <= MAX_WEIGHT) then{
                    statusCode = ERROR;
                }else{
                    DEFROST(value1,powerLevel,duration);
                }
            }
        }
        return statusCode;
    }   /* end defrost: Return status of OK or ERROR, powerLevel, and duration. */
endclass Defrost
```

class (file) Reheat::

```
    method (routine):StatusCode reheat(powerLevel, duration){
        SYSTEM sysObj;                /* instance of SYSTEM class */
        StatusCode statusCode = OK;
        KeyType key = NO_KEY_PRESSED;
        Integer servings = 0, value1 = 0, value2 = 0, count = 1;
        powerLevel = 0;
        duration = 0;
        key = sysObj.GET_KEYBOARD();
        while (statusCode is OK and key is not START and count <= 2) do{
            if not (is_digit(key)) then{
                statusCode = ERROR;
            }else{
                sysObj.PUT_DISPLAY(convert_to_text(key));
                servings = servings*10 + convert_to_integer(key);
                key = sysObj.GET_KEYBOARD();
```

```
                count = count + 1;
            }
        }
        if not (key is START) then{
            statusCode = ERROR;
        }else if not (MIN_SERVINGS <= servings <= MAX_SERVINGS) then{
                statusCode = ERROR;
        }else{
            key = sysObj.GET_KEYBOARD();
            count = 1;
            while (statusCode is OK and key is not START and count <= 4) do{
                if not (is_digit(key)) then{
                    statusCode = ERROR;
                }else{
                    sysObj.PUT_DISPLAY(convert_to_text(key));
                    if count <= 2 then{
                        value1 = value1*10 + convert_to_integer(key);
                    }else{
                        value2 = value2*10 + convert_to_integer(key);
                    }
                    key = sysObj.GET_KEYBOARD();
                    count = count + 1;
                }
            }
            if not (key is START) then{
                statusCode = ERROR;
            }else{
                if count <= 3 then{
                    value2 = value1;
                    value1 = 0;
                }
                if not (MIN_OUNCES <= value1 <= MAX_OUNCES) then{
                    statusCode = ERROR;
                }else{
                    /* Convert pounds to ounces. */
                    value1 = CONVERT_LARGER_UNITS_TO_SMALLER_UNITS * value1
                        + value2;
                    if not (MIN_WEIGHT <= value1 <= MAX_WEIGHT) then{
                        statusCode = ERROR;
                    }else{
                        REHEAT(servings,value1,powerLevel,duration);
                    }
                }
            }
        }
        return statusCode;
    }   /* end reheat: Return status of OK or ERROR, powerLevel, and duration. */
endclass Reheat
```

class (file) Manage_User_Interface:: /* Change signature for component: **APO**. */

```
    method (routine)::manage_user_interface(){
        SYSTEM sysObj;                      /* instance of SYSTEM class */
        Control_Electronics controlObj;     /* instance of Control_Electronics class */
        Straight_Heat heatObj;              /* instance of Straight_Heat */
```

```
        Defrost defrostObj;                 /* instance of Defrost */
        Reheat reheatObj;                   /* instance of Reheat */
        StatusCode statusCode = HEATING_COMPLETE;
        ErrorCode errorCode = BLANK;
        HeatingType heatRequest = NO_HEAT_TYPE;
        KeyType key = NO_KEY_PRESSED;
        MessageType statusMessage = NO_MESSAGE;
        Integer powerLevel = 0, duration = 0;
        while (statusCode = HEATING_COMPLETE and errorCode = BLANK) do{
            heatRequest = HEAT;
            key = sysObj.GET_KEYBOARD();
            if key is POWER then{
                statusCode = heatObj.straight_heat(powerLevel,duration);
            }else if key is DEFROST then{
                statusCode = defrostObj.defrost(powerLevel,duration);
            }else if key is REHEAT then{
                statusCode = reheatObj.reheat(powerLevel,duration);
            }else if key is STOP then{
                heatRequest = STOP;
            }else{
                statusCode = ERROR;
            }
            if statusCode is not OK then{
                statusMessage = ERROR_MESSAGE;
                statusCode = BLANK;
            }else{
                statusCode = controlObj.control_electronics(heatRequest,powerLevel,duration,
                    errorCode);
                if (statusCode is HEATING_COMPLETE) and (errorCode is BLANK) then{
                    statusMessage = DONE_MESSAGE;
                }else{
                    statusMessage = FAIL_MESSAGE;
                    statusCode = BLANK;
                }
            }
            sysObj.PUT_DISPLAY(convert_to_text(statusMessage));
        }
    }   /* end manage_user_interface */
endclass Manage_User_Interface
```

Appendix I Data Collection Tables for Design Evaluation

This appendix contains the data collection tables for the evaluation of the designs produced by the subjects who participated in the experiments discussed in Chapter 7. Table I.3 and Table I.2 indicate the types of data that are needed to calculate values of the target change complexity measures. Table I.3 - Table I.4 list the types of data required for the target structural complexity measures as discussed in Chapter 4 and Chapter 6.

Table I.1 Change analysis features at the routine level.

Change	Names of Routines (Methods) Reused With Modification	Size of Routines (Methods) Reused With Modification	Names of Routines (Methods) Reused Without Modification	Size of Routines (Methods) Reused Without Modification

Table I.2 Change analysis features at the component level.

Change	Names of Components (Classes) Reused With Modification	Size of Components (Classes) Reused With Modification	Names of Components (Classes) Reused Without Modification	Size of Components (Classes) Reused Without Modification

Table I.3 Structural complexity features for routines.

Name of Routine (Method)	Number of Routine (Method) Attributes (1)	Size of Routine (Method) Logic (2)	Number of Calls to Other Routines (Methods) (3)	Names of External Components (Classes) Providing Services (4)	Number of Decision Points $V(G_i)$ (5)

Table I.4 Structural complexity features for components.

Component (Class) Name	Number of Component (Class) Level Attributes (6)	Number of Component Routines (Methods) (7)	Component (Class) Size $(\Sigma(2)_{comp} + 6)$ (8)	Fan-In (number of external components calling internal routines or methods) (9)	Fan-Out (number of external components whose routines or methods are called) (10)	Number of External Routine (Method) Calls $(\Sigma(3)_{comp})$ (11)	$V(G_i)$ (across all routines or methods in the compo-nent) $(\Sigma(5)_{comp})$ (12)

Table I.5 Structural complexity features for systems.

Number of System Level Attributes (13)	Number of Components in System (14)	System Size $(\Sigma(8))_{comp} + 13)$ (15)	System V(G) $(\Sigma(12)_{comp})$ (16)

Appendix J Evaluation of Changeability for the Benchmark Design

This appendix contains the data collected to assess the changeability of the benchmark design. Table J.1 contains the data for evaluating changeability at the routine level, and Table J.2 contains the data for evaluating changeability at the component level. For each type of change (as designated in the first column of each table), the evaluator determined the impact on the routines and components specified in the benchmark design. Routines/components which must be modified or replaced to implement a particular change appear in the "Reused With Modification" column of the same row. Routines/components that can reused without modification to implement a particular change appear in the "Reused Without Modification" column of the same row. The third column contains the sum of the sizes of the routines/components reused with modification for the related change. The fifth column contains the sum of the sizes of the routines/components reused without modification for the related change. The reader should recall the relationship between the size of the part reused with modification and the size of the part reused without modification part as shown in Figure J.1. Section 4.5 discussed the method used to size routines, components, component-level attributes, and system-level attributes.

$$\text{Size of the system} = \Sigma(\text{ size}(rwm_i)) + \Sigma(\text{ size}(rwom_i)) + nc + as.$$
$$= \Sigma(\text{ size}(cwm_i)) + \Sigma(\text{ size}(cwom_i)) + ns.$$

where each rwm_i is a routine reused with modification, each $rwom_i$ is a routine reused without modification, each cwm_i is a component reused with modification, each $cwom_i$ is a component reused without modification, nc is the number of component-level attributes, and ns is the number of system-level attributes or logic defined/declared outside of the components.

Figure J.1 Relationship between the sizes of the parts reused with and without modification.

Table J.1 Evaluation of change impact on routines (methods) of the benchmark design.

Change	Name of Routines (Methods) Reused With Modification	Size of Routines (Methods) Reused With Modification (*in system.h)	Names of Routines (Methods) Reused Without Modification	Size of Routines (Method) Reused Without Modification (Total size of 364 does not include 12 class-level attributes.)
HBSQ	straight_heat	41	Microwave Oven Software minus {straight_heat}	323
DFORM	DEFROST	size(DEFROST)*	Microwave Oven Software	364
RFORM	REHEAT	size(REHEAT)*	Microwave Oven Software	364

Change	Name of Routines (Methods) Reused With Modification	Size of Routines (Methods) Reused With Modification (*in system.h)	Names of Routines (Methods) Reused Without Modification	Size of Routines (Method) Reused Without Modification (Total size of 364 does not include 12 class-level attributes.)
CPSRC	increase_all_power_sources, decrease_all_power_sources	30	Microwave Oven Software minus {increase_all_power_sources, decrease_all_power_sources}	334
FDBL	control_flow_component_C2, control_flow_component_C3	28	Microwave Software minus {control_flow_component_C2, control_flow_component_C3}	336
IMSWT	DEFROST, REHEAT	size(DEFROST) + size(REHEAT)*	Microwave Oven Software	364
HLWS	no change impact	0	Microwave Oven Software	364
CHD	heat_operation, control_flow_component_C1, control_flow_component_C2, control_flow_component_C3, read_all_devices_status	92	Microwave Oven Software minus {heat_operation, read_all_devices_status, control_flow_component_C1, control_flow_component_C2, control_flow_component_C3}	272
PSRC	initiate_power_source_status , read_power_source_status, increase_power, decrease_power, shut_off_power_source	16	Microwave Oven Software minus {x: x is a routine (method) of the Power_Source component (class).}	348
PSNSR	initiate_power_sensor_read, read_power_sensor_level, read_power_sensor_status	11	Microwave Oven Software minus {x: x is a routine (method) of the Power_Sensor component (class).}	353
DSNSR	initiate_door_status, read_door_status, read_door_sensor_status	10	Microwave Oven Software minus {x: x is a routine (method) of the Door_Sensor component (class).}	354
TIMER	convert_to_ticks, set_timer, check_timer_status	10	Microwave Oven Software minus {x: x is a routine (method) of the Timer component (class).}	354
APO	manage_user_interface	33	Microwave Oven Software minus {manage_user_interface}	331
EDA	replacement of heat_operation	19	Microwave Oven Software minus {heat_operation}	345

Table J.2 Evaluation of change impact on components (classes) of the benchmark design.

Change	Names of Components (Classes) Reused With Modification	Size of Components (Classes) Reused With Modification	Names of Components (Classes) Reused Without Modification	Size of Components (Classes) Reused Without Modification (Total size of 376 includes 12 class-level attributes.)
HBSQ	Straight_Heat	41	Microwave Oven Software minus {Straight_Heat component}	335
DFORM	SYSTEM	size(SYSTEM)	Microwave Oven Software	376
RFORM	SYSTEM	size(SYSTEM)	Microwave Oven Software	376
CPSRC	Increase_Decrease_Power	30	Microwave Oven Software minus {Increase_Decrease_Power}	346
FDBL	Hardware_Configuration	100	Microwave Oven Software minus {Hardware_Configuration}	276
IMSWT	SYSTEM	size(SYSTEM)	Microwave Oven Software	376
HLWS	no change impact	0	Microwave Oven Software	376
CHD	Hardware_Configuration	100	Microwave Oven Software minus {Hardware_Configuration}	276
PSRC	Power_Source	16	Microwave Oven Software minus {Power_Source}	360
PSNSR	Power_Sensor	11	Microwave Oven Software minus {Power_Sensor}	365
DSNSR	Door_Sensor	10	Microwave Oven Software minus {Door_Sensor}	366
TIMER	Timer	10	Microwave Oven Software minus {Timer}	366
APO	Manage_User_Interface	33	Microwave Oven Software minus {Manage_User_Interface}	343
EDA	replacement of heat_operation in Hardware_Configuration	100	Microwave Oven Software minus {Hardware_Configuration}	276

Appendix K Evaluation of Structural Complexity for the Benchmark Design

This appendix contains the data collected to assess the structural complexity of the benchmark design. Table K.1, Table K.2, and Table K.3 contain the structural complexity features at the routine, component, and system levels, respectively. For the meaning of the table columns labeled 1-16, see Appendix I.

Table K.1 Evaluation of structural complexity features at the routine level.

Component (Class) Name	Routine (Method) Name	1	2	3	4	5
Power_Source	initiate_power_source_status	1	3	1	SYSTEM	0
	read_power_source_status	1	3	1	SYSTEM	0
	increase_power	1	3	1	SYSTEM	0
	decrease_power	1	3	1	SYSTEM	0
	shut_off_power_source	1	3	1	SYSTEM	0
Power_Sensor	initiate_power_sensor_read	1	3	1	SYSTEM	0
	read_power_sensor_level	2	4	1	SYSTEM	0
	read_power_sensor_status	1	3	1	SYSTEM	0
Door_Sensor	initiate_door_status	1	3	1	SYSTEM	0
	read_door_status	1	3	1	SYSTEM	0
	read_door_sensor_status	1	3	1	SYSTEM	0
Timer	convert_to_ticks	2	4	0	none	0
	set_timer	1	3	1	SYSTEM	0
	check_timer_status	0	2	1	SYSTEM	0
Increase_Decrease_Power	increase_all_power_sources	6	15	1	Power_Source	3
	decrease_all_power_sources	6	15	1	Power_Source	3
Stop_All_Power_Sources	stop_all_power_sources	5	12	1	Power_Source	2
Initiate_Status_All_Power_Sources	initiate_status_all_power_sources	5	12	1	Power_Source	2
Hardware_Configuration	read_all_devices_status	4	34	6	Power_Source, Power_Sensor, Door_Sensor, Timer, SYSTEM	11
	control_flow_component_C1	3	13	3	Door_Sensor, Initiate_Status_All_Power_Sources, Power_Sensor	3
	control_flow_component_C2	4	14	3	Increase_Decrease_Power, Power_Sensor	4

Component (Class) Name	Routine (Method) Name	1	2	3	4	5
	control_flow_component_C3	4	14	3	Timer, Stop_All_Power_Sources	3
	heat_operation	6	19	2	Timer, Stop_All_Power_Sources	5
Control_Electronics	control_electronics	6	21	2	Hardware_Configuration, Stop_All_Power_Sources	4
Straight_Heat	straight_heat	8	41	7	SYSTEM, TIMER	10
Defrost	defrost	8	32	4	SYSTEM	7
Reheat	reheat	9	46	7	SYSTEM	11
Manage_User_Interface	manage_user_interface	12	33	6	SYSTEM, Straight_Heat, Defrost, Reheat, Control_Electronics	7

Table K.2 Evaluation of structural complexity features at the component level.

Component (Class) Name	6	7	8 ($\Sigma(2)_{Comp}$ + 6)	9	10	11 ($\Sigma(3)_{comp}$)	12 ($\Sigma(5)_{comp}$)
Power_Source	1	5	16	5	1	5	0
Power_Sensor	1	3	11	3	1	3	0
Door_Sensor	1	3	10	2	1	3	0
Timer	1	3	10	3	1	2	0
Increase_Decrease_Power	0	2	30	1	1	2	6
Stop_All_Power_Sources	0	1	12	2	1	1	2
Initiate_Status_All_Power_Sources	0	1	12	1	1	1	2
Hardware_Configuration	8	5	102	1	8	17	26
Control_Electronics	0	1	21	1	2	2	4
Straight_Heat	0	1	41	1	2	7	10
Defrost	0	1	32	1	1	4	7
Reheat	0	1	46	1	1	7	11
Manage_User_Interface	0	1	33	0	5	6	7

Table K.3 Evaluation of structural complexity features at the system level.

Number of System Level Attributes (13)	Number of Components in System (14)	System Size ($(\Sigma(8))_{comp}$ + 13) (15)	System V(G) ($\Sigma(12)_{comp}$) (16)
0	13	376	75

Appendix L Redesign Software Practice Exercise: Project Assignment 3

Resources:

- Documents describing Coda Client
- Coda lectures
- Design lectures

Task Objectives:

- Redesign the RVM facility of the Coda Client.
- Describe the rationale (reasons) for the new design.

Terminology:

The requirements document for the Coda Client contains a list of terms for understanding the functions of the software.

Deliverables:

To receive full credit for this assignment, you should staple all pages together and include a title page with the assignment name (Project Assignment 3), the course name, the due date, your name, and your e-mail address. Also put a legible version of your name and e-mail address on each page in the packet.

If you are participating in the project experiment, please make a copy of your deliverables (entire packet) and submit both the original and the copy by the due date. The experimenter will receive the copy of your assignments.

1. Redesign of the RVM facility. See the task descriptions in the Task Section.
   - Deliverable 1: Description of the new design for the RVM structures.
   - Deliverable 2: Description of the rationale for your new design.
2. Submit the deliverables produced **before and after** the review. **Please mark the corrections that you made on the pre-review version**. **Label the deliverables as pre-review or post-review**. It is important that the graders and experimenters be able to distinguish the before and after deliverables.
3. Complete the **data log form** as you accomplish each task and include the form at the end of your deliverables packet.

Introduction:

In Project Assignment 2, you analyzed the classes and functions that handle the Recoverable Virtual Memory or RVM features of the Venus Cache Manager (Section 3.2 in the requirements specification). You should now have a better understanding of the RVM organization. From here on the term Venus refers to the Venus Cache Manager of the Coda client.

The maximum Venus cache size is limited by the amount of used RVM. Current RVM usage is approximately 10% of the size of the cache. Venus maps RVM into the virtual address space of the Coda client, which is on Linux 2.2 kernels limited to 2GB. The client's virtual address space also includes reserved areas for program code, data, heap, shared libraries, memory-mapped files, and stack-space. As a result, the virtual address space can fit at most 1 GB of RVM data and less on most platforms.

One of the current problems is scalability: Coda cannot handle big client caches due to the current RVM design. Though stored in RVM, transient data does not need to be stored persistently. Transient data is reinitialized whenever Venus is started. As you can observe in the FSO structure (object), about 50% of the structure contain transient data. Taking the transient data out of the RVM may help to reduce the RVM overhead.

In this assignment, you will suggest ways to redesign the RVM facility to improve its run-time performance and scalability to larger Venus Client caches.

Tasks:

1. Redesign of the RVM facility.

   Consider the guidelines and techniques presented during the design lecture(s) that you attended to help you redesign the RVM facility.

   Reorganize the FSO structures (classes, functions, data definitions, etc.) to improve the RVM performance (reduce memory usage and CPU overhead for RVM transactions). You may want to define new classes or data structures and definitions. You may also want to recommend a different partition of the definitions and declarations across files.

   The FSO class definitions occur in Coda_Client/venus/fso.h. The definition of FSO is in the .cc files whose names start with fso. Transient data is marked with the comment /*T*/. The RVM library function prototypes and data definitions are located in Coda_Client/coda-include/rvm.h. See also rvm_segment.h, rvm_statistics.h, and rvmlib.h in the same directory.

   Use diagram(s) and text to describe your new design for the FSO structures. (**Deliverable 1**)

2. Explanation of rationale for redesign.

   Describe your rationale or reasons for your new design. (**Deliverable 2**)

3. Review of deliverables

   Review your deliverables carefully and make any necessary corrections**. Please mark your corrections on the pre-review versions**. Label the pre-review and post-review versions of your deliverables and submit both of them. You will not be penalized for corrections made to your pre-review design and rationale descriptions.

Appendix M Additional Information for Groups 2 & 3: Project Assignment 3

In Lecture 2, we discussed designing for reuse, change, and flexible performance. All of these features are important for the design of reliable distributed systems that can be evolved to satisfy future as well as current needs. The rationale is to “think ahead of time” during the design phase about how the parts of the solution could be reused across the design as well as across other solutions that are similar though not identical. The designer will also identify those parts of the solution that can or should have flexible performance. The key is to localize (isolate in components) those parts of the solution that are reusable, change together, or should have variant implementations with different levels of performance.

You have already been told about a change that we want to make to CODA now: to improve scalability by reducing the size of the data structure stored in RVM. Only persistent data from the fso object should go to RVM.

To reduce the effort involved in changing the system in the future, we want you to consider during your re-design that the following changes may need to be made later:

- Type and amount of the persistent data may change,
- Type and amount of the transient data may change,
- Operations on the persistent data may become different from the operations on the transient data,
- Activation of the operations on the persistent data may occur in a different order than for the transient data.

Appendix N Redesign of Coda Client Features: Project Assignment 4

Resources:

- Documents describing Coda Client (See http://www.coda.cs.cmu.edu/doc as well as the requirements specification.)
- Coda lectures
- Design lectures

Task Objectives:

- Redesign the Kernel-Venus interface of the Coda Client.
- Describe the rationale (reasons) for your new design.

Terminology:

The requirements document for the Coda Client contains a list of terms for understanding the functions of the software.

Deliverables:

To receive full credit for this assignment, you should staple all pages together and include a title page with the assignment name (Project Assignment 4), the course name, the due date, your name, and your e-mail address. Also put a legible version of your name and e-mail address on each page in the packet.

If you are participating in the project experiment, please make a copy of your deliverables (entire packet) and submit both the original and the copy by the due date. The experimenter will receive the copy of your assignments.

1. Redesign of the Kernel-Venus Interface. See the task descriptions in the Task Section.

   - Deliverable 1: Description of your new design for the Kernel-Venus interface.
   - Deliverable 2: Description of the rationale for your new design.

2. Submit the deliverables produced before and after the review. Please mark the corrections that you made on the pre-review version. Label the deliverables as pre-review or post-review. It is important that the graders and experimenters be able to distinguish the before and after deliverables.

3. Complete the **data log form** as you accomplish each task and include the form at the end of your deliverables packet.

Introduction:

In your analysis of the Coda Client, you should recall that there are three types of file system objects (files, directories, and symbolic links). A 96-bit file identifier (FID) identifies file system objects (FSOs).

The Coda Kernel (Kernel) uses the FIDs to request access to specific FSOs from the Venus Cache Manager (Venus). Venus provides the Kernel with access to a specific FSO by sending the device and inode numbers of the container file in which the FSO is stored on local disk.

Internally, the Kernel code hashes the 96-bit FIDs to 32-bit inode numbers. The problem is that different FSOs can become associated with the same inode number. On a Linux platform, this *inode collision* will make one of the colliding FSOs completely inaccessible. The Kernel code is also more complex than necessary to support FIDs that change during reintegration. For example, the inode number of a mount-point is the inode of the volume mount-point instead of the hashed FID of the volume's root directory.

To simplify maintenance of the Kernel code in the future, we would like to "clean-up" the Kernel-Venus interface. Venus should assign a unique inode number to every FSO so that the Kernel-Venus interface does not require the use of the FID. Venus should keep track of the inode number assigned to each FID associated with an FSO in the file cache. The Kernel will then refer to FSOs via the Venus-generated inode numbers, while Venus will map the inode numbers of the incoming Kernel requests to the corresponding FIDs.

Tasks:

1. Redesign of the Kernel-Venus interface

   Consider the guidelines and techniques presented during the design lecture(s) that you attended to help you redesign the Kernel-Venus interface. (**Deliverable 1**)

   Use diagram(s) and English text to describe your new design for the Kernel-Venus upcalls and downcalls as well as the related Kernel and Venus logic. In this assignment, we expect the upcall and downcall definitions to change because the Kernel and Venus will identify file system objects with unique Venus-generated inode numbers rather than 96-bit FIDs.

   To understand how the Kernel-Venus upcall and downcall interface currently works, you will need to study the following data structures as well as others.

   - vproc – basic thread class that processes the Venus File System (VFS) requests. These include the Kernel-Venus upcalls and downcalls. The vproc class definition appears in vproc.h. Definitions of the VFS operations are in vproc_vfscalls.cc. You may want to examine the venus_cnode structure defined in vproc.h that is used to package information to be passed back to the Kernel from Venus.
   - worker – derived thread class that inherits the vproc class. Worker objects receive the Kernel requests to Venus by waiting for and receiving messages from a message queue. The worker class definition occurs in worker.h, and the definitions of the worker operations are in worker.cc. In particular, notice the "main" function that contains a large switch statement to selectively handle the different types of Kernel to Venus requests.
   - Msgent – message class for creating messages between the Kernel and Venus. The file worker.h contains the definition for this class.
   - Venus process that opens the connection with the Kernel device via a call to the WorkerInit() function. The Venus process also sends messages from the Kernel to be processed by a "worker" thread to the "worker multiplexor" via a call to WorkerMux(). The logic for this process appears in the main function in the file venus.cc.

   Redesigning the Kernel-Venus interface may involve any combination of the following design actions. Apply any or all of these actions to "clean-up" the Kernel-Venus interface as described in the Introduction. You need not apply all of these actions.

   - Reorganize existing data and operations (logic) differently in existing or new files, classes, and methods.
   - Add new data or operations into existing or new files, classes, or methods.
   - Modify existing method or function interfaces. For example, add/delete parameters, change the types of the parameters, or change the type of the method or function.
   - Modify the logic of a method or function whose prototype has changed. For example, change the logic within a method or function whose parameters have changed.
   - Modify the logic of a method or function that calls a method or function whose prototype has changed. For example, change the logic of a method that calls a method whose parameter list has changed.

- Reorder logic within existing methods or functions. For example, reorder calls to other methods or functions.
- Create new directories.
- Organize existing and new files differently within the directories.

The specification of your new design should include the following types of information.

- Description of the directories and files that would contain the implementation of your design
- (Describe the parts of the design to be contained in each header file or other source code file. Describe the files to be included in each directory.)
- Data type declarations
- Definition of classes and methods

Pseudo-code descriptions of the logic to be performed by each method or function

Carefully describe your design with English prose in addition to diagrams. For instance, if you use diagrams to illustrate the classes in your design, discuss the purpose of the classes as well as the type and purpose of each data and method encapsulated in the classes.

Please remember that the specification of your design should be complete and well organized so that someone else could use your specification to help them code a new and correct implementation for the Kernel-Venus Interface and related logic in the Coda Client.

2. Explanation of rationale for redesign

   Discuss your rationale or reasons for your new design. For example, you might explain how and why you used the redesign activities listed above. Likewise, you should try to apply the concepts presented in the design lectures. In this section of your assignment, you should discuss how you applied these concepts in the derivation of your design. (**Deliverable 2**)

3. Review of deliverables

   Review your deliverables carefully and make any necessary corrections. Ask yourself the following questions:

   a. Does the new interface consistently use unique inode numbers generated by the Venus Cache Manager to identify file system objects?

   b. Is the Kernel consistently using the Venus-generated inode numbers in place of the hashed versions of the FIDs?

   c. Does your design explain where and how Venus generates the unique inode number for each FSO associated with an FID?

   d. Does Venus properly map the inode numbers that it generates back to the actual FIDs when processing Kernel requests?

   e. Does my design specification provide enough information so that I could implement the new Kernel-Venus interface and related logic correctly?

   Please mark your corrections on the pre-review versions. Label the pre-review and post-review versions of your deliverables and submit both of them. You will not be penalized for corrections made to your pre-review design and rationale descriptions.

Help Questions:

Some questions that you should try to answer before redesigning the Kernel-Venus interface follow.

1. What objects stay in existence while the Kernel is active? Is there a “Kernel” object?
2. What objects stay in existence while Venus is active? Is there a “Venus” object?
3. What objects are involved in creating and posting a Kernel request to Venus? What methods or functions are involved?
4. What objects are involved in receiving and processing a Kernel request to Venus? What methods or functions are involved?
5. Can you trace the sequence of method or function calls (object interactions) that occur to send a request from the Kernel to Venus and for Venus to process the request and send any necessary information back to the Kernel?
6. What object hashes the 96-bit FIDs to 32-bit inode numbers used by the Kernel? Is this done via a function/method call or in-line logic (e.g. MACRO expansion)?
7. Where in the Kernel code are the 32-bit inode numbers used? What parts (if any) of the Kernel design would need to change if the Kernel uses a unique inode number generated by Venus to identify FSOs?
8. What parts (if any) of the Venus design would need to change if Venus generates a unique inode number for each FSO and uses these numbers to identify FSOs in communications with the Kernel? Where and how should Venus generate these unique numbers? How should Venus keep track of the unique inode assigned to each FID (FSO identifier)?

Appendix O Additional Information for Groups 2 & 3: Project Assignment 4

In assignment 4, you are redesigning the Coda Kernel-Venus interface to simplify the Kernel logic. When redesigning Venus to hide information about FIDs from the Kernel, we also want to design it to enable evolution of the Coda Client in other related areas.

A Coda Client can currently only communicate with servers in a single administrative domain called a *realm*. We would like to redesign the Coda client so that adding support in the future for multiple realms would have minimal impact. Support for realms depends at least on the Kernel not knowing about FIDs because different realms will use the same FIDs to represent different objects. In other words, FIDs would be unique within a realm but not unique across realms. Therefore, to refer to a unique object, Venus would need to know the realm in which the object resides as well as its FID. In the client, a new persistent "realm" class/structure will most likely be associated with volumes in a particular realm.

Venus will also need to support multiple tokens for a single user, one per realm. Each token would allow a user to access file system objects within a specific realm for a period of time. The authentication daemons in a realm would allocate persistent unique user ids (uids) for foreign users (user outside a realm) to avoid changing the client-server protocols.

When you redesign Venus for Assignment 4, you should organize your design so that the following types of changes could be made in the future with minimal change to Venus or to the server.

1. Allow users to access file system objects from multiple realms.
    a. Associate each FSO with a realm as well as with a unique FID within a realm.
    b. Maintain multiple tokens (one per realm) for a single user.
2. Change the order of the initialization activities in the main Venus process. (See main function in venus.cc.) The original order of the operations is as shown below. A letter in the margin identifies each primary operation. Some related operations are grouped together. For example, the letter A represents all of the logic from the line on which the letter is positioned to the line preceding the line on which B is located.

    Some feasible variations to this order are:

    a. A B C D E F G H J I L K M N O P Q R S T V U X W Y Z AA BB
    b. A B C D F E G H I J K L M N O P Q R S T U V X W Z Y AA BB
    c. A B C D E F G H K I J L M N O P Q R S T U V W X Y Z AA BB

```
     int main(int argc, char **argv) {

     /* Print to the console -- important during reboot. */

     #if ! defined(__CYGWIN32__) && ! defined(DJGPP)
A.         freopen("/dev/console", "w", stderr);
     #endif
           fprintf(stderr, "Coda Venus, version %d.%d.%d\n\r",
           VenusMajorVersion, VenusMinorVersion, VenusReleaseVersion);
           fflush(stderr);
           coda_assert_action = CODA_ASSERT_SLEEP;
           coda_assert_cleanup = VFSUnmount;
           ParseCmdline(argc, argv);
           DefaultCmdlineParms();/* read vstab and /etc/coda/venus.conf */
```

```
        /* open the console file and print vital info */

            freopen(consoleFile, "w", stderr);
            fprintf(stderr, "Coda Venus, version %d.%d.%d\n",
            VenusMajorVersion, VenusMinorVersion,
            VenusReleaseVersion);
            fflush(stderr);
            CdToCacheDir();
            CheckInitFile();

            #if ! defined(__CYGWIN32__) && ! defined(DJGPP)
                SetRlimits();
            #endif

            /* Initialize. N.B. order of execution is very important here! */
            /* RecovInit < VSGInit < VolInit < FSOInit < HDB_Init */

            #ifdef DJGPP
                /* disable debug messages */

                __djgpp_set_quiet_socket(1);
            #endif

             /* test mismatch with kernel before doing real work */

B.          testKernDevice();

        /*
         * VprocInit MUST precede LogInit. Log messages are stamped
         * with the id of the vproc that writes them, so log messages
         * can't be properly stamped until the vproc class is initialized.
         *
         * The logging routines return without doing anything if LogInit
         * hasn't yet been called.
         */

C.          VprocInit();        /* init LWP/IOMGR support */
D.          LogInit();          /* move old Venus log and create a new one */
E.          LWP_SetLog(logFile, lwp_debug);
F.          RPC2_SetLog(logFile, RPC2_DebugLevel);
G.          SpoolInit();        /* make sure the spooling directory exists */
H.          DaemonInit();       /* before any Daemons initialize and after LogInit */
I.          ProfInit();
J.          StatsInit();
K.          SigInit();          /* set up signal handlers */
L.          DIR_Init(DIR_DATA_IN_RVM);
M.          RecovInit();        /* set up RVM and recov daemon */
N.          CommInit();         /* set up RPC2, {connection,server,mgroup} lists, probe daemon */
O.          UserInit();         /* fire up user daemon */
P.          VSGInit();          /* first alloc of recoverable vm, init VSGDB and daemon */
Q.          VolInit();          /* init VDB, daemon */
R.          FSOInit();          /* allocate FSDB if necessary, recover FSOs, start FSO daemon */
S.          HDB_Init();         /* allocate HDB if necessary, scan entries, start the HDB daemon */
T.          VmonInit();         /* set up Vmon and start Vmon daemon */
```

```
U.        MarinerInit();      /* set up mariner socket */
V.        WorkerInit();       /* open kernel device */
W.        CallBackInit();     /* set up callback subsystem and create callback server threads */
X.        WritebackInit();    /* set up writeback subsystem */
Y.        AdviceInit();       /* set up AdSrv and start the advice daemon */
Z.        LRInit();           /* set up local-repair database */
          //    VFSMount();

          /* Get the Root Volume. */

AA.       eprint("Getting Root Volume information...");
          while (!GetRootVolume()) {
              ServerProbe();
              struct timeval tv;
              tv.tv_sec = 15;
              tv.tv_usec = 0;
              VprocSleep(&tv);
          }
          VFSMount();

      #ifdef DJGPP
          k_Purge();
      #endif

          UnsetInitFile();
          eprint("Venus starting...");

          /* Act as message-multiplexor/daemon-dispatcher. */

BB.       for (;;) {
              /* Wait for a message or daemon expiry. */

              int rdfds = (KernelMask | MarinerMask);
              if (VprocSelect(NFDS, &rdfds, 0, 0, &DaemonExpiry) > 0) {

                  /* Handle mariner request(s). */
                  if (rdfds & MarinerMask) MarinerMux(rdfds);

                  /* Handle worker request. */
                  if (rdfds & KernelMask) WorkerMux(rdfds);
              }

              /* set in sighand.cc whenever we want to perform a clean shutdown */
              if (TerminateVenus) break;

              /* Fire daemons that are ready to run. */
              DispatchDaemons();
          }
          LOG(0, ("Venus exiting"));
          VDB->FlushVolume();
          RecovFlush(1);
          RecovTerminate();
          VFSUnmount();
          (void)CheckAllocs("TERM");
```

```
        fflush(logFile);
        fflush(stderr);
        LWP_TerminateProcessSupport();
        exit(0);}
/* end main */
```

Please Note:

You do not need to incorporate these changes into the current design. Rather, you should design Venus so that the changes could easily be made in the future. For example, you do not need to create a new realm class. Rather, you should organize the existing class structures so that the addition of realms could be made more easily in the future. The next section discusses some of the related data structures.

Relevant Data Structures:

In addition to the data structures discussed in the Project Assignment 4 handout, you may find it helpful to study the following data structures.

- userent – class that represents a user logged onto a machine running a Coda Client. The definition of the user class is in user.h. The definition of the user class methods are in user.cc.
- UserDaemon – a process that continually checks the time remaining for each user's authentication. The UserDaemon() function executes within a lightweight process initiated by USERD_Init(). The definition of both are in user.cc.
- SecretToken, ClearToken – structures that contain information regarding the authentication of users by servers. The definitions of these structures and related prototypes for authentication appear in auth2.h. Other server related prototypes appear in admon.h and adsrv.h.
- adviceconn – class that represents a Venus Advice Monitor. The adviceconn class definition is in adviceconn.h. The definition of the adviceconn methods TokensAcquired() and TokensExpired() are in advice.cc.

Help Questions:

Here are some questions that you should answer before redesigning Venus to make the changes listed above easier to implement in the future.

1. Which Venus data structures (e.g. classes or structures) contain information about an FSO's FID?
2. Where might the Venus keep information about an FSO's realm? Can this be done to minimize the impact on the existing Venus? Should some part of Venus be restructured to make the addition of realms with respect to the identification of FSOs easier?
3. Which Venus data structures (e.g. classes or structures) contain information about server authentication and tokens?
4. Where might Venus keep information about a user's token for each realm in which files are to be accessed? Can this be done to minimize the impact the existing Venus? Should some part of Venus be restructured to make the addition of realms with respect to user authentication easier?
5. How might the main Venus process be redesigned to simplify reordering the operations that it performs in the future?

Appendix P Design Evaluation Practice Exercise: Project Assignment 5

Resources:

- Calculator
- Coda lectures
- Design lectures
- Documents describing Coda Client
- Assignment 3 deliverables

Task Objectives:

- Evaluate new Coda Client RVM Facility for performance improvements.
- Evaluate new Coda Client RVM Facility for ease of adding features.

Terminology:

The requirements document for the Coda Client contains a list of terms for understanding the functions of the software. From here on, Venus and RVM refer to the Venus Cache Manager and the Recoverable Virtual Memory facility of the Coda Client, respectively. The term FSO refers to file system object.

Deliverables:

To receive full credit for this assignment, you should staple all pages together and include a title page with the assignment name (Project Assignment 5), the course name, the due date, your name, and your e-mail address. Also put a legible version of your name and e-mail address on each page in the packet.

If you are participating in the project experiment, please make a copy of your deliverables (entire packet) and submit both the original and the copy by the due date. The experimenter will receive the copy of your assignments.

1. Redesign of the RVM facility. See the task descriptions in the Task Section.

   - Deliverable 1: Evaluation of new Coda Client RVM Facility for performance improvements.
   - Deliverable 2: Evaluation of new Coda Client RVM Facility for ease of adding features.

2. Submit the deliverables produced **before and after** the review. **Please mark the corrections that you made on the pre-review version**. **Label the deliverables as pre-review or post-review**. It is important that the graders and experimenters be able to distinguish the before and after deliverables.

3. Complete the **data log form** as you accomplish each task and include the form at the end of your deliverables packet.

Introduction:

In Assignment 3, you learned that the maximum Venus cache size is limited by the amount of virtual memory used for RVM and that the Coda Client is, therefore, not scalable to large Venus client caches. Your task for this assignment was to suggest ways to redesign the RVM facility to improve its run-time performance and scalability to larger client caches.

In this assignment, you will evaluate the capability of your new design to satisfy the objectives of Assignment 3. In other words, you will determine how your new design would improve the Coda Client performance. You will also evaluate the impact of changing your new design to handle other features that the Coda research team may want to make in the future.

A software design team can use this type of analysis to determine if the proposed design satisfies the desired performance objectives. Likewise, impact evaluation is useful when planning the effort (time and skills) needed to modify a current software product to satisfy new product needs. The basic idea is to identify the features of the current product that would need to change and to estimate the effort needed to correctly change each feature.

Tasks:

Analyze the original Coda client design as well as your new design to perform the following evaluations.

A. Evaluation of your new design of the Coda Client RVM for performance improvements.

Assume a Linux System 2.2 Kernel and a compatible C++ compiler to determine the size of basic data types such as integers. For example, an integer (*int*) is 32 bits. Answer the following questions about the current design as well as your new design of the Coda Client RVM. **Please show your calculations**. (**Deliverable 1**)

Spatial Analysis of Performance

First, analyze the savings in RVM that should result from your new design of the RVM facility.

1. How many bytes db_o of memory for persistent and transient data are in the original *fsdb* data structure? Be careful to count the size of each field of a struct.
2. How many bytes db_n of memory are in your new data structure(s) for storing the persistent data from the old *fsdb*? Be careful to count the size of each field of a struct.
3. What is the percent reduction in RVM needed for the database metadata?

   [percent reduction = ((1 - db_n/db_o) * 100)]
4. How many bytes m_o of memory for persistent and transient data are in the original *fsobj* data structure? Be careful to count the size of each field of a struct.
5. How many bytes m_n of memory are in your new data structure(s) for storing the persistent data from the old *fsobj*? Be careful to count the size of each field of a struct.
6. What is the percent reduction in RVM needed for an FSO's metadata?

   [percent reduction = ((1 - m_n/m_o) * 100)]
7. Assume a limit of 1 GB of virtual memory for RVM and db_o and m_o to be the size of the original *fsdb* and *fsobj* data structures, respectively. What is the maximum number f_o of FSO*s* that may be allocated in the current Linux 2.2 system at any one time? [f_o = (1GB- db_o)/m_o]
8. Assume a limit of 1 GB of virtual memory for RVM and db_n and m_n to be the size of your new data structures to store persistent data. What is the maximum number f_n of FSOs that may be allocated in a Linux 2.2 system that uses your new design for persistent memory?

   [f_n = (1GB- db_n)/m_n]
9. What is the percent increase in the maximum number of FSOs that can be allocated in the system?

   [percent increase = ((f_n/f_o - 1) * 100)]
10. The size of RVM is currently about 10% of the Venus cache size. If the maximum size of RVM is

about 1 GB, what is the approximate size of the current Venus cache?

11. Assume that the size of the Venus cache is directly related to the maximum number of FSOs that can be allocated in the system at any one time. **Derive a formula** that shows the size of the Venus cache with respect to the maximum number of FSOs in the system. Now, **use the formula** and the results from the previous questions to approximate the maximum size of the Venus cache using your new design for persistent data.

12. Determine the percent increase in the maximum size of the Venus cache.

Temporal Analysis of Performance

Next, analyze the savings in execution time that may result from your new design of the RVM facility. Assume that many RVM operations (method calls) consist of the following sub-operations.

a. CTM – Call to RVM method (e.g. save current register values, push parameters onto stack, save return address, change program counter to address of new method, etc.).

b. PRE – Pre-operations for transfer of persistent data (e.g. prepare for atomic transaction).

c. TRANS – Transaction involving persistent data.

d. POST – Post-operations from transfer of persistent data (e.g. end atomic transaction).

Now assume that each sub-operation requires the following time to execute. Constants are shown in **bold** and variables in *italics*. Use the names of the constants in your calculations.

CTM – **ctm**

PRE – **pre**

TRANS – **trans** * *m*, where *m* is the length of the persistent data.

POST – **post**

RTC – **rtc**

The total time to perform an RVM operation follows.

total time = **ctm** + **pre** + **trans** * m + **post** + **rtc**

13. Calculate the total time t_o needed to transfer metadata data about an FSO. (Use the value of m_o from question 4.)

14. Calculate the total time t_n needed to transfer the persistent data about an FSO using your new design. (Use the value of m_n from question 5.)

15. Now generate a mathematical expression that indicates the percent reduction in execution time needed for RVM operations.

    Hint: (See the expression used in question 6 and use the values of t_o and t_n as determined in questions 13 and 14.)

B. Explanation of your new design of the Coda Client RVM for ease of adding features.

Enumerated below are changes that the Coda research team might like to make in the future. For each type of change, perform the impact analysis steps completely and accurately. Create an impact analysis **table for each change** to graphically display the results of applying the impact analysis steps.

The grade assigned to your answers will depend upon completeness as well as accuracy. Use your new design as well as the related parts of the Coda Client code for the size estimates. (**Deliverable 2**)

Types of changes:

1. Add two new persistent data fields to the cached file-system database metadata (current *fsdb* structure).
2. Change the type of the transient data fields "readers," "writers," "openers," "Writers," "Execers," and "refcnt" from short to int (currently in the *fsdb* structure).
3. Create a new class called CacheStats that encapsulates the transient data DirAttrStats, DirDataStats, FileAttrStats, and FileDataStats. An object of type CacheStats may reside in the object that contains the transient *fsdb* data. This new class will contain operations such as UpdateCacheStats() and PrintCacheStats(). The type of these operations will be changed to *int*. The file fso0.cc currently contains the definitions for these functions.
4. Move the operation ResetTransient() from the *fsdb* class to a class that encapsulates the transient data that was originally stored in *fsdb*. The declaration of ResetTransient() in the class *fsdb* occurs in fso.h, and the definition of ResetTransient() occurs in fso0.cc.

Impact Analysis Steps:

1. List the program elements (other than classes, methods, or functions) **that will need to be modified** to handle the change. Indicate the container file for each program element.
2. List the classes **that will need to be modified** to handle the change. The needed modification may involve data definitions, method definitions or declarations, or references to methods of other classes. Indicate the container file for each class definition.
3. List the methods or functions **that will need to be modified** to handle the change. The needed modification may involve local variable declarations or program logic including references to other methods or functions. Indicate the container class if the definition occurs within the class definition. Indicate the container file for each method or function definition or implementation.
4. Estimate the size of each program element listed in Step 1.
5. Estimate the size of each class listed in Step 2.
6. Estimate the size of each method or function listed in Step 3.
7. Determine the size of each file that contains the declaration or definition of a data structure that must change or a reference to the data structure that must change (those files listed in Steps 1-3). Add the sizes of each include statement, pre-compiler directive, data definition, class definition, method definition, and function definition contained in the file.

Estimating the Size of Program Elements and Files:

Type	**Size**
include statement	1
pre-compiler directive	1
data definition	Count each semi-colon, ";", in the definition. (e.g. The definition for a struct that contains 3 semi-colons would have a count of 3.
program statements	Count each semi-colon, ";", in the statements.method or function declaration 1. (e.g. void ResetTransient(); in the class *fsdb* would have a count of 1.)

method or function definition	Count the local data definitions and program statements in the method or function. Count local data definitions and program statements as described above. (e.g. A method containing local variable definitions using 2 semi-colons and programming statements using 10 semi-colons would have a size of 12.)
class definition	Count each data definition and method declaration or definition in the class as described above. (e.g. A class containing data definitions using 5 semi-colons and 3 method declarations would have a size of 5+3 or 8.) (e.g. A class containing data definitions of size 3, 4 method declarations, and 1 method definition of size 10, would have a size of 3+4+10 = 17.) Please note that the implementations of most class methods appear in .cc files separate from the class definitions that occur in .h files.
file	Count each include statement, pre-compiler directive, data definition, method or function definition, and each class definition that occur in the file. Use the above guidelines for sizing each element of the file.

Example:

Table P.1 Impact analysis for change type "a."

Container File	File Size	Container Class	Class Size	Program Element	Data Size	Method or Function Definition	Method or Function Size
foo.h	50	NA		struct fab	4		
foo.h	50	class foo	15				
foo.h	50	class goo	20			giddle()	7
foo.cc	34	NA		enum fit	1		
foo.cc	34	NA				fiddle()	12
foo.cc	34	NA				faddle()	14
foo.cc	34	NA				fuddle()	7
foo2.cc	21	NA				fy()	9

File foo.h contains the definition of a struct called fab that will need to be modified to handle change type "a." This file also contains the definitions for classes foo and goo that must also be modified to handle the change type "a." The size of foo.h is greater than the sum of the sizes of fab, foo, and goo (50 > 4+15+20) because foo.h also contains the definitions of program elements that are not involved in the change type "a". The definition of fab does not occur within a class definition. Therefore, no container class is applicable (NA). The method giddle must change to accommodate change type "a." The definition for giddle() occurs within the class goo. Therefore, goo is the container class for giddle(). The size of goo includes the size of giddle().

File foo.cc contains definitions for the enumeration fit as well as for the methods fiddle, faddle, and fuddle. **Each of these definitions requires modification** to handle change type "a." The file does not contain any other data structure definitions or program elements. Therefore, the size of foo.cc is the sum of the sizes of enum fit, fiddle(), faddle(), and fuddle() or 34. The definitions of fit, fiddle, faddle, and fuddle occur outside of a class definition: there is no applicable container class for these program elements.

The file foo2.cc contains the definition for a function called fy that must change to handle change type “a.” Other program elements also reside in foo2.cc.

C. Review of deliverables

Review your deliverables carefully and make any necessary corrections**. Please mark your corrections on the pre-review versions**. Label the pre-review and post-review versions of your deliverables and submit both of them. You will not be penalized for corrections made to your pre-review design and rationale descriptions.

Appendix Q Evaluation of New Kernel-Venus Organization: Project Assignment 6

Resources:

- Calculator
- Coda lectures
- Design lectures
- Documents describing Coda Client
- Assignment 4 deliverables

Task Objectives:

- Evaluate new Kernel-Venus interface.
- Evaluate new Kernel and Venus organization for ease of change.

Terminology:

The requirements document for the Coda Client contains a list of terms for understanding the functions of the software. From here on, Venus and Kernel refer to the Venus Cache Manager and the Coda Client Kernel, respectively. The term FID refers to the identifier for a file system object. The term FSO refers to a file system object.

Deliverables:

To receive full credit for this assignment, you should staple all pages together and include a title page with the assignment name (Project Assignment 6), the course name, the due date, your name, and your e-mail address. Also put a legible version of your name and e-mail address on each page in the packet.

If you are participating in the project experiment, please make a copy of your deliverables (entire packet) and submit both the original and the copy by the due date. The experimenter will receive the copy of your assignments.

1. Redesign of the Kernel-Venus interface. See the task descriptions in the Task Section.

    - Deliverable 1: Evaluation of the new Kernel-Venus interface.
    - Deliverable 2: Evaluation of the new Kernel and Venus organization for ease of change.

2. Submit the deliverables produced **before and after** the review. **Please mark the corrections that you made on the pre-review version**. **Label the deliverables as pre-review or post-review**. It is important that the graders and experimenters be able to distinguish the before and after deliverables.

3. Complete the **data log form** as you accomplish each task and include the form at the end of your deliverables packet.

Introduction:

In Assignment 4, you “cleaned-up” the Kernel-Venus interface so that maintenance on the Kernel and Venus would be simplified. In this assignment, you will evaluate the capability of your new design to satisfy the objectives of Assignment 4. You will determine how well you isolated the knowledge of the 96-bit FIDs from the Kernel. You will also analyze the impact of changing your Assignment 4 design to handle other features, such as realms and multiple user authentication tokens, that the Coda research team may want to make in the future.

A Coda Client currently communicates with servers in a single administrative domain called a realm. In the future, we would like to change the Coda Client to support multiple realms. Support for realms depends at least on the Kernel not knowing about FIDs because different realms will use the same FIDs to represent different objects. In other words, FIDs would be unique within a realm but not unique across realms. Therefore, to refer to a unique object, Venus would need to know the realm in which the object resides as well as its FID. In the client, a new persistent "realm" class/structure will most likely be associated with volumes in a particular realm.

Venus will also need to support multiple tokens for a single user, one per realm. Each token would allow a user to access file system objects within a specific realm for a period of time. The authentication daemons in a realm would allocate persistent unique user ids (uids) for foreign users (user outside a realm) to avoid changing the client-server protocols.

Tasks:

To accomplish the following tasks, use your design from Assignment 4 as well as the original Coda Client code.

A. Evaluation of the new Kernel-Venus interface.

Suppose that you have been consulted to determine the effort needed to "clean-up" the Kernel-Venus interface. As described in Assignment 4, we would like Venus to convert the 96-bit FID assigned to an FSO into a unique 32-bit inode number. Venus should use these unique inode numbers to identify FSOs when communicating with the Kernel. Answer the following questions to evaluate your redesign of the Kernel-Venus interface from Assignment 4. The grade assigned to your answers will depend upon completeness as well as accuracy. You will receive credit for finding ways to improve your Assignment 4 design (no penalty for mistakes in that you find in your Assignment 4 design). (**Deliverable 1**)

1. In the original Coda Client, which Kernel data structures (e.g. classes or structs) contain information about an FSO's FID?

   List these data structures along with the names of the files that contain them.

2. Does your Assignment 4 design hide the Kernel data structures listed above from the original 96-bit FID for an FSO?

   For each data structure listed in the answer to Question 1, **briefly** explain how your Assignment 4 design **does** or **does not** hide the data structure from the 96-bit FID. For each data structure that still uses the 96-bit FID, explain why or explain how the design could be improved.

3. In the original Coda Client, which Venus data structures (e.g. classes or structs) contain information about an FSO's FID?

   List these data structures along with the names of the files that contain them.

4. Does your Assignment 4 design properly handle the conversion from 96-bit FIDs to 32-bit inode numbers to uniquely identify each FSO?

   a. List the data structure(s) in your Assignment 4 design that handles the conversion from 96-bit FIDs to 32-bit unique inode numbers.

   b. List the data structures in your Assignment 4 design that correspond to the data structures listed in the answer to Question 3. Identify those that still use the 96-bit FID and explain why. Identify those that use the new 32-bit unique inode number and explain why.

5. Now apply the impact analysis steps from Assignment 5 (listed near the end of this document) to

estimate the size of the changes needed to "clean-up" the original Kernel-Venus interface. Estimate the size of the program elements, class, methods or functions, and files in the original Coda Client that must be modified to "clean-up" the Kernel-Venus interface. One way to do this is to use your design from Assignment 4 to identify the old program features that would need to be modified.

B. Evaluation of the new Kernel and Venus organization for ease of change.

Enumerated below are changes that the Coda research team might like to make in the future. For each type of change, perform the impact analysis steps completely and accurately. Create an impact analysis **table for each change** to graphically display the results of applying the impact analysis steps. These steps are the same as those that you used in Assignment 5. The types of changes are, of course, different from those in Assignment 5. Estimate the changes needed to a Coda Client implemented according to your Assignment 4 design. If your design does not include enough detail, use the related parts of the Coda Client code to help with the size estimates. The grade assigned to your analysis will depend upon completeness as well as accuracy. (**Deliverable 2**)

NOTE: This is somewhat different from part 5 of Deliverable 1. In part 5, you are estimating the effort to change the original Coda Client so that it would conform to your Assignment 4 design or to a similar design if there were flaws in your Assignment 4 design. In this part, you are estimating the effort to make additional changes to a Coda Client already implemented according to your Assignment 4 design. The idea is to determine how well your Assignment 4 design facilitates these new changes. There is no penalty if your Assignment 4 design does not easily accommodate the changes.

Types of changes:

1. Allow users to access file system objects from multiple realms. Associate each FSO with a realm as well as with a unique FID within a realm.
2. Maintain multiple tokens (one per realm) for a single user.
3. Change the order of the initialization activities in the main Venus process to the following new order: A B C D F E G H I J K L M N O P Q R S T U V X W Z Y AA BB.

   The main Venus process is in venus.cc. The code segment that follows shows the original order (A B C D E F G H I J K L M N O P Q R S T U V W X Y Z AA BB) of the operations in the Venus process. A letter in the margin identifies each primary operation. Some related operations are grouped together. For example, the letter A represents all of the logic from the line on which the letter is positioned to the line preceding the line on which B is located.

```
        int main(int argc, char **argv) {

        /* Print to the console -- important during reboot. */

        #if ! defined(__CYGWIN32__) && ! defined(DJGPP)
A.              freopen("/dev/console", "w", stderr);
        #endif
                fprintf(stderr, "Coda Venus, version %d.%d.%d\n\r",
                VenusMajorVersion, VenusMinorVersion, VenusReleaseVersion);
                fflush(stderr);
                coda_assert_action = CODA_ASSERT_SLEEP;
                coda_assert_cleanup = VFSUnmount;
                ParseCmdline(argc, argv);
                DefaultCmdlineParms();/* read vstab and /etc/coda/venus.conf */
```

```
    /* open the console file and print vital info */

        freopen(consoleFile, "w", stderr);
        fprintf(stderr, "Coda Venus, version %d.%d.%d\n",
        VenusMajorVersion, VenusMinorVersion,
        VenusReleaseVersion);
        fflush(stderr);
        CdToCacheDir();
        CheckInitFile();

        #if ! defined(__CYGWIN32__) && ! defined(DJGPP)
            SetRlimits();
        #endif

        /* Initialize. N.B. order of execution is very important here! */
        /* RecovInit < VSGInit < VolInit < FSOInit < HDB_Init */

        #ifdef DJGPP
            /* disable debug messages */

            __djgpp_set_quiet_socket(1);
        #endif

         /* test mismatch with kernel before doing real work */

B.      testKernDevice();

    /*
     * VprocInit MUST precede LogInit. Log messages are stamped
     * with the id of the vproc that writes them, so log messages
     * can't be properly stamped until the vproc class is initialized.
     *
     * The logging routines return without doing anything if LogInit
     * hasn't yet been called.
     */

C.      VprocInit();      /* init LWP/IOMGR support */
D.      LogInit();        /* move old Venus log and create a new one */
E.      LWP_SetLog(logFile, lwp_debug);
F.      RPC2_SetLog(logFile, RPC2_DebugLevel);
G.      SpoolInit();      /* make sure the spooling directory exists */
H.      DaemonInit();     /* before any Daemons initialize and after LogInit */
I.      ProfInit();
J.      StatsInit();
K.      SigInit();        /* set up signal handlers */
L.      DIR_Init(DIR_DATA_IN_RVM);
M.      RecovInit();      /* set up RVM and recov daemon */
N.      CommInit();       /* set up RPC2, {connection,server,mgroup} lists, probe daemon */
O.      UserInit();       /* fire up user daemon */
P.      VSGInit();        /* first alloc of recoverable vm, init VSGDB and daemon */
Q.      VolInit();        /* init VDB, daemon */
R.      FSOInit();        /* allocate FSDB if necessary, recover FSOs, start FSO daemon */
S.      HDB_Init();       /* allocate HDB if necessary, scan entries, start the HDB daemon */
T.      VmonInit();       /* set up Vmon and start Vmon daemon */
```

```
U.        MarinerInit();      /* set up mariner socket */
V.        WorkerInit();       /* open kernel device */
W.        CallBackInit();     /* set up callback subsystem and create callback server threads */
X.        WritebackInit();    /* set up writeback subsystem */
Y.        AdviceInit();       /* set up AdSrv and start the advice daemon */
Z.        LRInit();           /* set up local-repair database */
          //    VFSMount();

          /* Get the Root Volume. */

AA.       eprint("Getting Root Volume information...");
          while (!GetRootVolume()) {
              ServerProbe();
              struct timeval tv;
              tv.tv_sec = 15;
              tv.tv_usec = 0;
              VprocSleep(&tv);
          }
          VFSMount();

      #ifdef DJGPP
          k_Purge();
      #endif

          UnsetInitFile();
          eprint("Venus starting...");

          /* Act as message-multiplexor/daemon-dispatcher. */

BB.       for (;;) {
              /* Wait for a message or daemon expiry. */

              int rdfds = (KernelMask | MarinerMask);
              if (VprocSelect(NFDS, &rdfds, 0, 0, &DaemonExpiry) > 0) {

                  /* Handle mariner request(s). */
                  if (rdfds & MarinerMask) MarinerMux(rdfds);

                  /* Handle worker request. */
                  if (rdfds & KernelMask) WorkerMux(rdfds);
              }

              /* set in sighand.cc whenever we want to perform a clean shutdown */
              if (TerminateVenus) break;

              /* Fire daemons that are ready to run. */
              DispatchDaemons();
          }
          LOG(0, ("Venus exiting"));
          VDB->FlushVolume();
          RecovFlush(1);
          RecovTerminate();
          VFSUnmount();
          (void)CheckAllocs("TERM");
```

```
        fflush(logFile);
        fflush(stderr);
        LWP_TerminateProcessSupport();
        exit(0);}
    /* end main */
```

Impact Analysis Steps:

1. List the program elements (other than classes, methods, or functions) **that will need to be modified** to handle the change. Indicate the container file for each program element.
2. List the classes **that will need to be modified** to handle the change. The needed modification may involve data definitions, method definitions or declarations, or references to methods of other classes. Indicate the container file for each class definition.
3. List the methods or functions **that will need to be modified** to handle the change. The needed modification may involve local variable declarations or program logic including references to other methods or functions. Indicate the container class if the definition occurs within the class definition. Indicate the container file for each method or function definition or implementation.
4. Estimate the size of each program element listed in Step 1.
5. Estimate the size of each class listed in Step 2.
6. Estimate the size of each method or function listed in Step 3.
7. Estimate the size of each file that contains the declaration or definition of a data structure that must change or a reference to the data structure that must change (those files listed in Steps 1-3). Add the sizes of each include statement, pre-compiler directive, data definition, class definition, method definition, and function definition contained in the file.

Estimating the Size of Program Elements and Files

Type	**Size**
include statement	1
pre-compiler directive	1
data definition	Count each semi-colon, ";", in the definition. (e.g. The definition for a struct that contains 3 semi-colons would have a count of 3.)
program statements	Count each semi-colon, ";", in the statements.
method or function declaration	1 (e.g. void ResetTransient(); in the class *fsdb* would have a count of 1.)
method or function definition	Count the local data definitions and program statements in the method or function. Count local data definitions and program statements as described above. (e.g. A method containing local variable definitions using 2 semi-colons and programming statements using 10 semi-colons would have a size of 12.)
class definition	Count each data definition and method declaration or definition in the class as described above. (e.g. A class containing data definitions using 5 semi-colons and 3 method declarations would have a size of 5+3 or 8.) (e.g. A class containing data definitions of size 3, 4 method declarations, and 1 method definition of size 10, would have a size of 3+4+10 = 17.)

	Please note that the implementations of most class methods appear in .cc files separate from the class definitions that occur in .h files.
file	Count each include statement, pre-compiler directive, data definition, method or function definition, and each class definition that occur in the file. Use the above guidelines for sizing each element of the file.

C. Review of deliverables

Review your deliverables carefully and make any necessary corrections**. Please mark your corrections on the pre-review versions**. Label the pre-review and post-review versions of your deliverables and submit both of them. You will not be penalized for corrections made to your pre-review design and rationale descriptions.

Relevant Data Structures:

To complete the tasks for this assignment, you may find it helpful to study the following data structures.

userent – class that represents a user logged onto a machine running a Coda Client. The definition of the user class is in user.h. The definition of the user class methods are in user.cc.

UserDaemon – a process that continually checks the time remaining for each user's authentication. The UserDaemon() function executes within a lightweight process initiated by USERD_Init(). The definition of both are in user.cc.

SecretToken, ClearToken – structures that contain information regarding the authentication of users by servers. The definitions of these structures and related prototypes for authentication appear in auth2.h. Other server related prototypes appear in admon.h and adsrv.h.

adviceconn – class that represents a Venus Advice Monitor. The adviceconn class definition is in adviceconn.h. The definition of the adviceconn methods TokensAcquired() and TokensExpired() are in advice.cc.

Help Questions:

Here are some questions that you should answer before identifying the parts of the Coda Client that would need to change to add realms and multiple user authentication tokens.

1. Where might the Venus keep information about an FSO's realm? Can this be done to minimize the impact on the existing Venus?
2. Which Venus data structures (e.g. classes or structs) contain information about server authentication and tokens?
3. Where might Venus keep information about a user's token for each realm in which files are to be accessed? Can this be done to minimize the impact the existing Venus?

Appendix R Change Impact for Each Expected or Feasible Change

Table R.1 Change signatures for the expected or feasible changes to the microwave oven software.

Expected or Feasible Evolution of the Microwave Oven Software	Change Signature
POWER-TIMER as well as TIMER-POWER button sequences to program straight heating.	HBSQ
Different defrost formula.	DFORM
Different reheat formula.	RFORM
More sophisticated control of power sources than up or down three notches.	CPSRC
Different feedback loops to allow for electronics which respond faster.	FDBL
Error and status messages, as well as weight measures, for the international market.	IMSWT
More powerful microwaves with higher limits for weight, servings, and weight per serving.	HLWS
Different configurations of hardware devices.	CHD
Different type of power source with new parameters for the CALL_HARDWARE interface.	PSRC
Different type of power sensor with new parameters for the CALL_HARDWARE interface.	PSNSR
Different type of door sensor with new parameters for the CALL_HARDWARE interface.	DSNSR
Different type of timer with new parameters for the CALL_HARDWARE interface.	TIMER
Addition of programmed operations to heat specific foods such as bacon, popcorn, or vegetables.	APO
Event-driven approach to controlling the hardware.	EDA

Change Impact at the Routine Level:

Table R.2 Change impact at the routine level for change HBSQ.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	17.32%	19.05%	21.43%	11.26%
Median	14.65%	16.48%	15.07%	11.26%
Standard Deviation	13.38%	13.80%	16.55%	NA
Maximum	38.46%	44.44%	59.79%	11.26%
Minimum	4.12%	1.95%	8.77%	11.26%

Table R.3 Change impact at the routine level for change DFORM.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	0.00%	0.00%	0.00%	0.00%
Median	0.00%	0.00%	0.00%	0.00%
Standard Deviation	0.00%	0.00%	0.00%	NA
Maximum	0.00%	0.00%	0.00%	0.00%
Minimum	0.00%	0.00%	0.00%	0.00%

Table R.4 Change impact at the routine level for change RFORM.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	0.00%	0.00%	0.00%	0.00%
Median	0.00%	0.00%	0.00%	0.00%
Standard Deviation	0.00%	0.00%	0.00%	NA
Maximum	0.00%	0.00%	0.00%	0.00%
Minimum	0.00%	0.00%	0.00%	0.00%

Table R.5 Change impact at the routine level for change CPSRC.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	15.33%	17.69%	14.79%	8.24%
Median	9.40%	17.45%	10.82%	8.24%
Standard Deviation	12.27%	9.71%	9.27%	NA
Maximum	37.18%	34.74%	34.67%	8.24%
Minimum	5.24%	3.45%	7.89%	8.24%

Table R.6 Change impact at the routine level for change FDBL.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	21.17%	21.43%	13.70%	7.69%
Median	22.25%	20.77%	9.62%	7.69%
Standard Deviation	10.95%	7.83%	11.04%	NA
Maximum	39.74%	34.74%	34.67%	7.69%
Minimum	8.24%	8.33%	3.91%	7.69%

Table R.7 Change impact at the routine level for change IMSWT.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	36.54%	43.87%	38.36%	0.00%
Median	35.90%	36.55%	44.58%	0.00%
Standard Deviation	16.79%	26.51%	26.82%	NA
Maximum	54.95%	97.30%	73.68%	0.00%
Minimum	9.68%	12.34%	0.00%	0.00%

Table R.8 Change impact at the routine level for change HLWS.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	22.59%	22.83%	21.92%	0.00%
Median	23.35%	20.11%	20.14%	0.00%
Standard Deviation	13.54%	12.01%	19.67%	NA
Maximum	38.46%	44.44%	59.79%	0.00%
Minimum	0.00%	2.97%	0.00%	0.00%

Table R.9 Change impact at the routine level for change CHD.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	26.27%	30.54%	34.25%	25.27%
Median	26.40%	29.54%	36.41%	25.27%
Standard Deviation	11.69%	6.81%	9.35%	NA
Maximum	39.74%	43.51%	47.56%	25.27%
Minimum	8.24%	20.27%	21.05%	25.27%

Table R.10 Change impact at the routine level for change PSRC.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	18.36%	24.93%	30.93%	4.40%
Median	15.42%	27.62%	23.26%	4.40%
Standard Deviation	11.12%	12.31%	24.75%	NA
Maximum	37.18%	50.65%	83.51%	4.40%
Minimum	7.50%	6.03%	3.51%	4.40%

Table R.11 Change impact at the routine level for change PSNSR.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	12.45%	20.23%	14.35%	3.02%
Median	5.73%	19.51%	11.78%	3.02%
Standard Deviation	14.46%	10.80%	11.06%	NA
Maximum	37.18%	35.14%	35.37%	3.02%
Minimum	1.05%	2.59%	1.75%	3.02%

Table R.12 Change impact at the routine level for change DSNSR.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	11.84%	16.24%	14.01%	2.75%
Median	3.38%	19.35%	10.87%	2.75%
Standard Deviation	14.72%	12.04%	11.27%	NA
Maximum	37.18%	35.14%	35.37%	2.75%
Minimum	2.09%	1.58%	0.88%	2.75%

Table R.13 Change impact at the routine level for change TIMER.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	20.39%	21.01%	22.14%	2.75%
Median	6.68%	22.36%	9.73%	2.75%
Standard Deviation	24.40%	16.66%	21.04%	NA
Maximum	53.85%	48.65%	64.95%	2.75%
Minimum	2.64%	1.05%	7.02%	2.75%

Table R.14 Change impact at the routine level for change ADO.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	17.12%	18.19%	20.20%	9.07%
Median	11.62%	12.12%	16.48%	9.07%
Standard Deviation	11.57%	14.50%	17.65%	NA
Maximum	38.46%	44.44%	59.79%	9.07%
Minimum	8.82%	2.92%	5.95%	9.07%

Table R.15 Change impact at the routine level for change EDA.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	22.20%	22.25%	14.35%	5.22%
Median	22.51%	22.83%	9.62%	5.22%
Standard Deviation	11.13%	7.57%	10.44%	NA
Maximum	39.74%	34.74%	34.67%	5.22%
Minimum	8.82%	8.33%	5.36%	5.22%

Change Impact at the Component Level:

Table R.16 Change impact at the component level for change HBSQ.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	51.52%	52.78%	51.52%	10.90%
Median	47.30%	57.65%	43.53%	10.90%
Standard Deviation	30.37%	32.47%	26.54%	NA
Maximum	96.30%	97.47%	95.65%	10.90%
Minimum	21.60%	3.47%	21.84%	10.90%

Table R.17 Change impact at the component level for change DFORM.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	0.00%	0.00%	0.00%	0.00%
Median	0.00%	0.00%	0.00%	0.00%
Standard Deviation	0.00%	0.00%	0.00%	NA
Maximum	0.00%	0.00%	0.00%	0.00%
Minimum	0.00%	0.00%	0.00%	0.00%

Table R.18 Change impact at the component level for change RFORM.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	0.00%	0.00%	0.00%	0.00%
Median	0.00%	0.00%	0.00%	0.00%
Standard Deviation	0.00%	0.00%	0.00%	NA
Maximum	0.00%	0.00%	0.00%	0.00%
Minimum	0.00%	0.00%	0.00%	0.00%

Table R.19 Change impact at the component level for change CPSRC.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	38.82%	51.98%	44.59%	7.98%
Median	27.11%	45.87%	31.00%	7.98%
Standard Deviation	29.68%	28.06%	31.28%	NA
Maximum	96.30%	97.47%	95.65%	7.98%
Minimum	13.93%	13.49%	12.71%	7.98%

Table R.20 Change impact at the component level for change FDBL.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	37.42%	51.85%	34.14%	26.60%
Median	25.69%	45.87%	29.92%	26.60%
Standard Deviation	30.58%	28.04%	27.43%	NA
Maximum	96.30%	97.47%	95.65%	26.60%
Minimum	11.94%	18.25%	8.96%	26.60%

Table R.21 Change impact at the component level for change IMSWT.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	60.15%	73.22%	59.01%	0.00%
Median	61.21%	72.26%	56.39%	0.00%
Standard Deviation	35.27%	20.93%	32.27%	NA
Maximum	100.00%	97.47%	95.65%	0.00%
Minimum	9.40%	38.17%	0.00%	0.00%

Table R.22 Change impact at the component level for change HLWS.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	50.40%	59.37%	40.53%	0.00%
Median	49.78%	57.65%	36.68%	0.00%
Standard Deviation	33.89%	24.99%	36.27%	NA
Maximum	96.30%	97.47%	95.65%	0.00%
Minimum	0.00%	26.83%	0.00%	0.00%

Table R.23 Change impact at the component level for change CHD.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	41.26%	52.83%	53.52%	26.60%
Median	32.31%	45.87%	46.62%	26.60%
Standard Deviation	29.84%	26.91%	24.88%	NA
Maximum	96.30%	97.47%	95.65%	26.60%
Minimum	11.94%	23.81%	31.03%	26.60%

Table R.24 Change impact at the component level for change PSRC.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	36.95%	41.89%	47.57%	4.26%
Median	25.69%	31.46%	41.96%	4.26%
Standard Deviation	30.83%	30.33%	30.84%	NA
Maximum	96.30%	97.47%	95.65%	4.26%
Minimum	13.93%	10.48%	6.72%	4.26%

Table R.25 Change impact at the component level for change PSNSR.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	36.95%	41.37%	47.03%	2.93%
Median	25.69%	31.46%	40.36%	2.93%
Standard Deviation	30.83%	30.96%	30.86%	NA
Maximum	96.30%	97.47%	95.65%	2.93%
Minimum	13.93%	4.29%	6.72%	2.93%

Table R.26 Change impact at the component level for change DSNSR.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	36.95%	40.43%	45.26%	2.66%
Median	25.69%	31.46%	38.80%	2.66%
Standard Deviation	30.83%	32.11%	31.81%	NA
Maximum	96.30%	97.47%	95.65%	2.66%
Minimum	13.93%	1.79%	6.72%	2.66%

Table R.27 Change impact at the component level for change TIMER.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	58.14%	54.39%	66.82%	2.66%
Median	50.01%	45.87%	72.30%	2.66%
Standard Deviation	32.79%	36.53%	24.51%	NA
Maximum	100.00%	100.00%	95.65%	2.66%
Minimum	25.14%	1.43%	28.81%	2.66%

Table R.28 Change impact at the component level for change ADO.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	47.44%	57.88%	51.52%	8.78%
Median	41.78%	66.56%	43.53%	8.78%
Standard Deviation	34.57%	28.85%	26.54%	NA
Maximum	96.30%	97.47%	95.65%	8.78%
Minimum	9.40%	6.62%	21.84%	8.78%

Table R.29 Change impact at the component level for change EDA.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	38.46%	51.85%	41.70%	26.60%
Median	25.69%	45.87%	32.89%	26.60%
Standard Deviation	30.93%	28.04%	33.38%	NA
Maximum	96.30%	97.47%	95.65%	26.60%
Minimum	11.94%	18.25%	8.96%	26.60%

Change Impact at the Routine Level with Comparative Sizing:

Table R.30 Change impact at the routine level with comparative sizing for change HBSQ.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	17.48%	16.75%	26.42%	11.26%
Median	11.02%	13.56%	26.57%	11.26%
Standard Deviation	21.14%	13.91%	13.84%	NA
Maximum	54.29%	45.02%	49.70%	11.26%
Minimum	2.30%	1.95%	11.95%	11.26%

Table R.31 Change impact at the routine level with comparative sizing for change DFORM.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	0.00%	0.00%	0.00%	0.00%
Median	0.00%	0.00%	0.00%	0.00%
Standard Deviation	0.00%	0.00%	0.00%	NA
Maximum	0.00%	0.00%	0.00%	0.00%
Minimum	0.00%	0.00%	0.00%	0.00%

Table R.32 Change impact at the routine level with comparative sizing for change RFORM.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	0.00%	0.00%	0.00%	0.00%
Median	0.00%	0.00%	0.00%	0.00%
Standard Deviation	0.00%	0.00%	0.00%	NA
Maximum	0.00%	0.00%	0.00%	0.00%
Minimum	0.00%	0.00%	0.00%	0.00%

Table R.33 Change impact at the routine level with comparative sizing for change CPSRC.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	21.65%	29.45%	15.75%	8.24%
Median	10.21%	29.57%	9.80%	8.24%
Standard Deviation	16.90%	16.93%	11.93%	NA
Maximum	42.14%	57.92%	40.15%	8.24%
Minimum	8.06%	8.26%	8.75%	8.24%

Table R.34 Change impact at the routine level with comparative sizing for change FDBL.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	36.30%	34.27%	19.81%	7.69%
Median	37.99%	30.22%	20.90%	7.69%
Standard Deviation	5.09%	11.57%	11.80%	NA
Maximum	42.14%	57.92%	40.15%	7.69%
Minimum	28.49%	18.81%	4.78%	7.69%

Table R.35 Change impact at the routine level with comparative sizing for change IMSWT.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	38.21%	40.78%	36.38%	0.00%
Median	40.86%	32.07%	40.25%	0.00%
Standard Deviation	13.41%	27.32%	30.59%	NA
Maximum	54.29%	99.45%	85.23%	0.00%
Minimum	23.72%	12.34%	0.55%	0.00%

Table R.36 Change impact at the routine level with comparative sizing for change HLWS.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	25.57%	25.08%	19.39%	0.00%
Median	23.72%	21.40%	19.17%	0.00%
Standard Deviation	17.79%	16.80%	18.17%	NA
Maximum	54.29%	59.25%	49.70%	0.00%
Minimum	9.51%	2.05%	0.00%	0.00%

Table R.37 Change impact at the routine level with comparative sizing for change CHD.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	41.34%	44.15%	36.74%	25.27%
Median	42.14%	43.26%	34.61%	25.27%
Standard Deviation	4.70%	7.12%	7.49%	NA
Maximum	47.75%	57.92%	48.48%	25.27%
Minimum	34.75%	29.08%	28.66%	25.27%

Table R.38 Change impact at the routine level with comparative sizing for change PSRC.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	24.52%	35.43%	32.31%	4.40%
Median	14.75%	43.24%	26.51%	4.40%
Standard Deviation	16.23%	18.15%	22.12%	NA
Maximum	42.29%	57.92%	76.36%	4.40%
Minimum	10.51%	4.36%	4.37%	4.40%

Table R.39 Change impact at the routine level with comparative sizing for change PSNSR.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	18.08%	29.21%	17.20%	3.02%
Median	4.59%	32.49%	13.41%	3.02%
Standard Deviation	20.14%	19.26%	13.82%	NA
Maximum	42.14%	57.92%	40.15%	3.02%
Minimum	2.69%	2.48%	2.92%	3.02%

Table R.40 Change impact at the routine level with comparative sizing for change DSNSR.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	17.51%	26.05%	16.98%	2.75%
Median	2.70%	26.50%	13.41%	2.75%
Standard Deviation	20.64%	21.43%	13.98%	NA
Maximum	42.14%	57.92%	40.15%	2.75%
Minimum	2.30%	1.62%	2.62%	2.75%

Table R.41 Change impact at the routine level with comparative sizing for change TIMER.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	21.38%	29.38%	20.46%	2.75%
Median	6.61%	34.94%	11.62%	2.75%
Standard Deviation	24.70%	25.58%	20.60%	NA
Maximum	52.69%	60.92%	53.94%	2.75%
Minimum	1.64%	1.57%	2.62%	2.75%

Table R.42 Change impact at the routine level with comparative sizing for change ADO.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	24.79%	15.93%	25.95%	9.07%
Median	11.83%	10.16%	26.57%	9.07%
Standard Deviation	20.40%	14.92%	14.68%	NA
Maximum	54.29%	45.02%	49.70%	9.07%
Minimum	8.87%	2.92%	9.62%	9.07%

Table R.43 Change impact at the routine level with comparative sizing for change EDA.

Statistic	Control Group	Rationale Group	Rationale+Method Group	Benchmark Design
Mean	36.30%	35.49%	19.57%	5.22%
Median	37.99%	34.39%	20.90%	5.22%
Standard Deviation	5.09%	11.91%	12.07%	NA
Maximum	42.14%	57.92%	40.15%	5.22%
Minimum	28.49%	18.81%	5.67%	5.22%

Appendix S Analysis of Variance, the F Statistic, and the Correlation Analysis

Experimental error can cause differences between the treatment groups that are independent of the treatment effects. Experimental error frequently originates from inadequate control of nuisance variables such as the differences between subjects or from measurement error. It is extremely difficult, if not impossible, to eliminate experimental error completely.

The research studies focused on change impact. In review,

> Change impact is a measure of the part of a software system which must be modified to satisfy new requirements for the software. Reducing change impact is important because it helps to reduce the amount of effort needed to evolve the software system.

Therefore, the reader should note that differences in the observations within the same treatment group (e.g. change impact values for the designs created by the subjects) are due to experimental error. On the other hand, differences in the group means (e.g. mean change impact for each treatment group) are the result of treatment effects and/or experimental error.

As shown in Figure S.1, the null hypothesis, H_0, depicts the expectation that observed differences between the treatment group means are due primarily to experimental error (the treatment effect is null or negligible). The alternative hypothesis, H_1, conveys the experimental goal that observed differences are due to both treatment effects and experimental error. H_0 predicts that the treatment group means are near equal. H_1 predicts that differences between the treatment group means is statistically significant. In any given experiment, it is possible to obtain a value that is *greater* than 1.0 when H_0 is *true* or one that is *equal* to or *less* than 1.0 when H_1 is *true*. Therefore, analysis of variance through application of the F statistic is used to determine if H_0 is true or false within a chosen degree of confidence.

$$\frac{\text{differences among means of treatment groups}}{\text{differences among subjects treated alike}} = \frac{\text{experimental error}}{\text{experimental error}} \sim 1.0 \text{ for } H_0$$

$$\frac{\text{differences among means of treatment groups}}{\text{differences among subjects treated alike}} = \frac{\text{(treatment effects)} + \text{(experimental error)}}{\text{experimental error}} > 1.0 \text{ for } H_1$$

Figure S.1 Comparison between the null and alternative hypotheses [83].

Variance is a measure of the deviation of a group of values from the group mean. Figure S.2 shows the general mathematical expression for calculating variance.

$$\frac{\sum_{i=1}^{n}(y_i-\bar{y})^2}{df}$$

where n is the number of values in the group, each y_i is a value in the group, $\bar{y}$ is the group mean, and df is the degrees of freedom.

Figure S.2 General mathematical expression for calculating variance [83,97].

The term in the numerator is the *sum of squares*, or the sum of the squared deviations between each value in the group and the group mean. The *degrees of freedom* term in the denominator corresponds to the number of values with independent information that enter into the calculation of the sum of squares.

Three types of deviation which are important to testing H_0 are:

1. Within each treatment group G_j, the deviation of each score y_{ij} from the group mean $\bar{y}_{Gj}$ (within-groups deviation).
2. The deviation of each group mean $\bar{y}_{Gj}$ from the grand mean $\bar{y}_T$ (between-groups deviation).
3. The deviation of each score y_{ij} from the grand mean $\bar{y}_T$ (total deviation).

Figure S.3 clarifies the meaning for each deviation. The mathematical expression for the grand mean appears in Figure S.4.

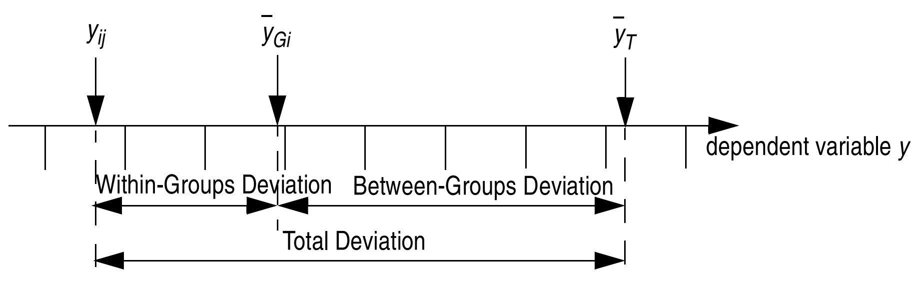


Figure S.3 Component deviations for testing H_0 [83].

$$\text{grand mean} = \bar{Y}_T = \frac{\sum_{j}^{g}\sum_{i}^{n} y_{ij}}{gn}$$

where n is the number in each treatment group, g is the number of treatment groups, and y_{ij} is the i-th value in the j-th treatment group.

Figure S.4 Mathematical expression for the grand mean [99].

Figure S.5 through Figure S.7 show the sums of squares related to the three types of deviation discussed previously. The sums of squares are related mathematically as shown in Figure S.8.

$$\text{Treatment Sum of Squares } SST = n\sum_{j}^{g} \langle \bar{y}_j - \bar{y}_T \rangle^2$$

where g is the number of treatment groups, n is the number of observations in each treatment group, $\bar{y}_j$ is the mean of the *j-th* treatment group, and $\bar{y}_T$ is the grand mean.

Figure S.5 Mathematical expression for the Treatment Sum of Squares which measures the variability between the treatment group means. This variability is sometimes called the explained variability [100].

$$\text{Error Sum of Squares } SSE = \sum_{j}^{g} \sum_{i}^{n} \langle y_{ij} - \bar{y}_j \rangle^2$$

where n is the number of observations in each treatment group, g is the number of treatment groups, y_{ij} is the *i-th* value in the *j-th* treatment group, and $\bar{y}_j$ is the mean of the *j-th* treatment group.

Figure S.6 Mathematical expression for the Error Sum of Squares which measures the individual variation within a treatment group due to chance or unexplained error [100].

$$\text{Total Sum of Squares or Total } SS = \sum_{j}^{g} \sum_{i}^{n} \langle y_{ij} - \bar{y}_T \rangle^2$$

where n is the number of observations in each treatment group, g is the number of treatment groups, y_{ij} is the *i-th* value in the *j-th* treatment group, and $\bar{y}_T$ is the grand mean.

Figure S.7 Mathematical expression for the Total Sum of Squares which measures the variability that results when all values are treated as a combined sample coming from a common population [100].

$$\text{Total } SS = SST + SSE$$

Figure S.8 Property of Sum of Squares [100].

The total variation, Total *SS*, has the two components *SST* (explained variation) and *SSE* (unexplained variation). A comparison of the magnitudes of these two components is useful in determining whether or not the variation is great enough to significantly refute H_0. For instance, the experimenter will most likely accept H_0 if the explained and unexplained variations are about the same magnitude. The experimenter cannot directly compare *SST* and *SSE* because they are the sums of different numbers of squared deviations. The experimenter must first convert them to mean sums of squares via division by the appropriate degrees of freedom. The resulting statistics are the Treatment Mean Square and Error Mean Square shown in Figure S.9 and Figure S.10, respectively.

$$\text{Treatment Mean Square} = MST = \frac{SST}{g-1}$$

where SST is the Treatment Sum of Squares and g is the number of treatment groups.

Figure S.9 Mathematical expression for the Treatments Mean Square [100].

$$\text{Error Mean Square} = MSE = \frac{SSE}{g\langle n-1\rangle}$$

where SSE is the Error Sum of Squares, g is the number of treatment groups, and n is the number of values in each treatment group.

Figure S.10 Mathematical expression for the Error Mean Square [100].

The test statistic for the analysis of variance, the F statistic, is the ratio of the variance explained by treatments (MST) and the unexplained variance (MSE) as shown in Figure S.11.

$$F = \frac{\text{Variance explained by treatments}}{\text{Unexplained variance}} = \frac{MST}{MSE}$$

Figure S.11 Test statistic for analysis of variance [84,101].

The experimenter can reject H_0 if the calculated value of the F statistic is greater than or equal to $F_{\alpha,}$ the value from the F distribution associated with the appropriate degrees of freedom and desired significance level α. An α of 0.05 means that there is a 5% probability that values taken by the random variable F will be greater than F_α. The significance level α also indicates the probability of a type I error, rejecting H_0 when it is true. The degrees of freedom are those used to calculate MST and MSE. Many statistics texts contain a table of F values for varying pairs of degrees of freedom and significance levels [82,96]. Table S.1 through Table S.3 explain the data and calculations used to test the H_0 hypothesis.

Table S.1 Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	specified as part of the results of the experiment
n	number of subjects in a treatment group	specified as part of the results of the experiment

Table S.2 Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1$	$MST = \frac{SST}{df_{MST}}$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle$	$MSE = \frac{SSE}{df_{MSE}}$

Table S.3 Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE}$.
F_{α}	Select the value from a statistical table of F values, where α is the desired significance level. $F_{\alpha}(df_{MST}, df_{MSE})$
Accept/Reject H_0	Accept H_0 if $F_{Calculated} < F_{\alpha}$. Reject H_0 if $F_{Calculated} \geq F_{\alpha}$.

The reader should note that the analysis of variance expressions presented above depend on the number of observations (e.g. change impact values) for each treatment group being equal. In some cases, the number of observations for each group may not be equal. This may happen if subjects withdraw from an experiment, error in the data collection process occurs, or observations are unusable for some other reason. In the case of the research studies, some subjects did not complete the required designs or produced designs which were not measurable as discussed in Section 7.9. The result was an unequal number of observations in each treatment group.

The following techniques are available for handling unequal sample or treatment groups.

1. Removal of observations from the largest treatment groups until each group has the same number of observations via:
   a. Random removal.
   b. Selective removal based on a policy that does not attempt to bias the results of any particular treatment group.
2. Application of an approach to adjust for unequally sized treatment groups such as:
   a. Analysis of unweighted means.
   b. Analysis of weighted means.

Though random removal is obviously the least biased removal approach, removal of specific observations may be justifiable if the experimenter can determine a reason for the unequal sizes that does not unfairly bias the results of any particular treatment group. For instance, completion of the required task under the conditions of one of the treatments may require a higher level of some skill or of some innate capability by the subjects than the level required by the other treatments. If the level of this skill or capability was measured before the experiment, then removal of either low or high observations based on subject skill or capability possibly will not bias the results of the treatment group that requires more skilled or capable subjects. The choice of removing either low or high observations depends on the context of the experiment and the type of observations. In general, the experimenter would not remove both the lowest and the highest observations.

The two mathematical approaches for handling unequal treatment group sizes differ in the way that they handle the treatment group means. The method of unweighted means treats each mean *equally* by substituting a mean treatment group size (or sample size) for the actual sizes associated with the different treatment groups. The other approach, the method of weighted means, weights each mean according to the actual sample sizes. The next paragraphs describe the changes to the standard analysis of variance calculations to implement each approach.

The method of unweighted means requires two changes to the standard analysis of variance as shown in Figure S.12.

1. A new calculation for the grand mean $\bar{y}_T*$ that is an average of the group means $\bar{y}_i$. The number of groups is *g*.

$$\bar{y}_T^* = \frac{\sum_{i=1}^{g} \bar{y}_i}{g}$$

2. A substitution of the sample group size *n* with a special average n_h called the *harmonic mean* that is obtained by dividing the number of groups *g* by the sum of the reciprocals of the group sample sizes.

$$n_h = \frac{g}{1/n_1 + 1/n_2 + \ldots + 1/n_g} = \frac{g}{\sum_{i=1}^{g} \langle 1/n_i \rangle}$$

Figure S.12 Mathematical expressions for calculating the grand mean and harmonic mean used by the method of unweighted means [85].

The method of weighted means substitutes a slightly different mathematical expression for the treatment sum of squares. In this expression, the deviation of each group mean $\bar{y}_i$ is weighted by the related sample size n_i as shown in Figure S.13.

$$\text{Treatment Sum of Squares} = SST = \sum_{i}^{g} \left[(n_i)(\bar{y}_i - \bar{y}_T)^2 \right]$$

Figure S.13 Mathematical expression for calculating the treatment sum of squares used by the method of weighted means [85].

The final calculation to be discussed is the product-moment correlation or *analysis of correlation*. This calculation expresses the degree to which two variables are linearly related. The sign of the resulting value indicates the direction of the relationship: a positive value indicates that the two variables vary in the same direction, and a negative value indicates an indirect correlation. The magnitude of the absolute value of the correlation ranges from 0 (indicating no correlation) to 1 (indicating perfect correlation). Figure S.14 presents the mathematical expression for the product-moment correlation. The analysis of correlation is useful in the research studies for determining if there is a correlation between structural complexity and change impact (the dependent variable being tested).

$$\text{product-moment correlation} = r_{xy} = \frac{SP_{XY}}{\sqrt{\langle SS_X\rangle\langle SS_Y\rangle}} = \frac{\sum_{i=1}^{n}\langle x-\bar{x}\rangle\langle y-\bar{y}\rangle}{\sqrt{\langle \sum_{i=1}^{n}\langle x-\bar{x}\rangle^2\rangle\langle \sum_{j=1}^{n}\langle y-\bar{y}\rangle^2\rangle}}$$

where X and Y are the variables being correlated, $\bar{x}$ is the mean of the group of X values, $\bar{y}$ is the mean of the group of Y values, and n is the number of values in each group of X or Y values.

Figure S.14 Mathematical expression for the product-moment correlation or correlation analysis [86,98].

For those cases in which the subjects within treatment groups may vary significantly with respect to an independent variable that can affect the observations, *analysis of covariance* can help to statistically reduce the experimental error. The reader should see [86] for a discussion of analysis of covariance.

Appendix T Experiment 1: Analysis of Variance (ANOVA) for Change Impact

Analysis of variance of change impact at the *routine level* using *equalized* sample sizes:

The parameters and calculations for the analysis of variance of the mean change impact at the routine level using equalized samples appear in Table T.1, Table T.2, and Table T.3. As can be seen in Table T.3, the experimenter should accept H_0 and reject H_1 when using equalized samples.

> **Conclusion:** Any difference between the mean change impact at the routine level of any treatment group is due to experimental error alone (and not due to differences in treatments).

Table T.1 Experiment 1 ANOVA: Routine Level, Equal Size Samples - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	6

Table T.2 Experiment 1 ANOVA: Routine Level, Equal Size Samples - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 24.0$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 15$	$MSE = \frac{SSE}{df_{MSE}} = 61.6$

Table T.3 Experiment 1 ANOVA: Routine Level, Equal Size Samples - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.39$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,15) = 3.68$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance of change impact at the *routine level* using *unequal* sample sizes and *unweighted means analysis*:

The parameters and calculations for the analysis of variance of the mean change impact at the routine level using unequal sample sizes and unweighted means analysis appear in Table T.4, Table T.5, and Table T.6. As can be seen in Table T.6, the experimenter should accept H_0 and reject H_1 when using unequal sample sizes and unweighted means analysis.

> **Conclusion:** Any difference between the mean change impact at the routine level of any treatment group is due to experimental error alone (and not due to differences in treatments).

Table T.4 Experiment 1 ANOVA: Routine Level, Unequal Size Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=6, n_2=12, n_3=8

Table T.5 Experiment 1 ANOVA: Routine Level, Unequal Size Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 13.79$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 23$	$MSE = \frac{SSE}{df_{MSE}} = 61.19$

Table T.6 Experiment 1 ANOVA: Routine Level, Unequal Size Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.23$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,23) = 3.39$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance of change impact at the *routine level* using *unequal* sample sizes and *weighted means analysis*:

The parameters and calculations for the analysis of variance of the mean change impact at the routine level using unequal sample sizes and weighted means analysis appear in Table T.7, Table T.8, and Table T.9. As can be seen in Table T.9, the experimenter should accept H_0 and reject H_1 when using unequal sample sizes and weighted means analysis.

> **Conclusion:** Any difference between the mean change impact at the routine level of any treatment group is due to experimental error alone (and not due to differences in treatments).

Table T.7 Experiment 1 ANOVA: Routine Level, Unequal Size Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=6, n_2=12, n_3=8

Table T.8 Experiment 1 ANOVA: Routine Level, Unequal Size Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 14.33$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 23$	$MSE = \frac{SSE}{df_{MSE}} = 61.19$

Table T.9 Experiment 1 ANOVA: Routine Level, Unequal Size Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.23$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,23) = 3.39$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance of change impact at the *component level* using *equalized* sample sizes:

The parameters and calculations for the analysis of variance of the mean change impact at the component level using equalized samples appear in Table T.10, Table T.11, and Table T.12. As can be seen in Table T.12, the experimenter should accept H_0 and reject H_1 when using equalized samples.

> **Conclusion:** Any difference between the mean change impact at the component level of any treatment group is due to experimental error alone (and not due to differences in treatments).

Table T.10 Experiment 1 ANOVA: Component Level, Equal Size Samples - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	15

Table T.11 Experiment 1 ANOVA: Component Level, Equal Size Samples - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 107.0$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 15$	$MSE = \frac{SSE}{df_{MSE}} = 226.0$

Table T.12 Experiment 1 ANOVA: Component Level, Equal Size Samples - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.47$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,15) = 3.68$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance of change impact at the *component level* using *unequal* sample sizes and *unweighted means analysis*:

The parameters and calculations for the analysis of variance of the mean change impact at the component level using unequal sample sizes and unweighted means analysis appear in Table T.13, Table T.14, and Table T.15. As can be seen in Table T.15, the experimenter should accept H_0 and reject H_1 when using unequal sample sizes and unweighted means analysis.

> **Conclusion:** Any difference between the mean change impact at the component level of any treatment group is due to experimental error alone (and not due to differences in treatments).

Table T.13 Experiment 1 ANOVA: Component Level, Unequal Size Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=6, n_2=12, n_3=8

Table T.14 Experiment 1 ANOVA: Component Level, Unequal Size Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 92.84$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 23$	$MSE = \frac{SSE}{df_{MSE}} = 464.90$

Table T.15 Experiment 1 ANOVA: Component Level, Unequal Size Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.20$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,23) = 3.39$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance of change impact at the *component level* using *unequal* sample sizes and *weighted means analysis*:

The parameters and calculations for the analysis of variance of the mean change impact at the component level using unequal sample sizes and weighted means analysis appear in Table T.16, Table T.17, and Table T.18. As can be seen in Table T.18, the experimenter should accept H_0 and reject H_1 when using unequal sample sizes and weighted means analysis.

> **Conclusion:** Any difference between the mean change impact at the component level of any treatment group is due to experimental error alone (and not due to differences in treatments).

Table T.16 Experiment 1 ANOVA: Component Level, Unequal Size Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=6, n_2=12, n_3=8

Table T.17 Experiment 1 ANOVA: Component Level, Unequal Size Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 96.06$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 23$	$MSE = \frac{SSE}{df_{MSE}} = 464.90$

Table T.18 Experiment 1 ANOVA: Component Level, Unequal Size Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.21$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,23) = 3.39$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance of change impact at the *routine level* with *comparative sizing* and using *equalized* sample sizes:

The parameters and calculations for the analysis of variance of the mean change impact at the routine level with comparative sizing using equalized samples appear in Table T.19, Table T.20, and Table T.21. As can be seen in Table T.21, the experimenter should accept H_0 and reject H_1 when using equalized samples.

> **Conclusion:** Any difference between the mean change impact at the routine level with comparative sizing of any treatment group is due to experimental error alone (and not due to differences in treatments).

Table T.19 Experiment 1 ANOVA: Routine Level with Comparative Sizing, Equal Size Samples - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	15

Table T.20 Experiment 1 ANOVA: Routine Level with Comparative Sizing, Equal Size Samples - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 72.5$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 15$	$MSE = \frac{SSE}{df_{MSE}} = 89.0$

Table T.21 Experiment 1 ANOVA: Routine Level with Comparative Sizing, Equal Size Samples - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.81$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,15) = 3.68$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance of change impact at the *routine level* with *comparative sizing* and using *unequal* sample sizes and *unweighted means analysis*:

The parameters and calculations for the analysis of variance of the mean change impact at the routine level using unequal sample sizes and unweighted means analysis appear in Table T.22, Table T.23, and Table T.24. As can be seen in Table T.24, the experimenter should accept H_0 and reject H_1 when using unequal sample sizes and unweighted means analysis.

Conclusion: Any difference between the mean change impact at the routine level with comparative sizing of any treatment group is due to experimental error alone (and not due to differences in treatments).

Table T.22 Experiment 1 ANOVA: Routine Level with Comparative Sizing, Unequal Size Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=6, n_2=12, n_3=8

Table T.23 Experiment 1 ANOVA: Routine Level with Comparative Sizing, Unequal Size Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 71.25$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 23$	$MSE = \frac{SSE}{df_{MSE}} = 103.50$

Table T.24 Experiment 1 ANOVA: Routine Level with Comparative Sizing, Unequal Size Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.69$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,23) = 3.39$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance of change impact at the *routine level* with comparative sizing and using *unequal* sample sizes and *weighted means analysis*:

The parameters and calculations for the analysis of variance of the mean change impact at the routine level with comparative sizing using unequal sample sizes and weighted means analysis appear in Table T.25, Table T.26, and Table T.27. As can be seen in Table T.27, the experimenter should accept H_0 and reject H_1 when using unequal sample sizes and weighted means analysis.

> **Conclusion:** Any difference between the mean change impact at the routine level with comparative sizing of any treatment group is due to experimental error alone (and not due to differences in treatments).

Table T.25 Experiment 1 ANOVA: Routine Level with Comparative Sizing, Unequal Size Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=6, n_2=12, n_3=8

Table T.26 Experiment 1 ANOVA: Routine Level with Comparative Sizing, Unequal Size Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g-1 = 2$	$MST = \frac{SST}{df_{MST}} = 88.03$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 23$	$MSE = \frac{SSE}{df_{MSE}} = 103.50$

Table T.27 Experiment 1 ANOVA: Routine Level with Comparative Sizing, Unequal Size Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.85$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,23) = 3.39$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Appendix U Experiment 1: Analysis of Covariance (ANOCOV) for Change Impact

Analysis of covariance of the total time and change impact at the *routine level* using *equalized* sample sizes:

Table U.1 Experiment 1 ANOCOV with Time: Routine Level, Equal Size Samples - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	6

Table U.2 Experiment 1 ANOCOV with Time: Routine Level, Equal Size Samples - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST_{Adj} = \frac{SST_{Adj}}{df_{MST}} = 22.89$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle - 1 = 14$	$MSE_{Adj} = \frac{SSE_{Adj}}{df_{MSE}} = 55.02$

Table U.3 Experiment 1 ANOCOV with Time: Routine Level, Equal Size Samples - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST_{Adj}}{MSE_{Adj}} = 0.42$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,14) = 3.74$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of covariance of total time and change impact at the *component level* using *equalized* sample sizes:

Table U.4 Experiment 1 ANOCOV with Time: Component Level, Equal Size Samples - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	6

Table U.5 Experiment 1 ANOCOV with Time: Component Level, Equal Size Samples - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST_{Adj} = \frac{SST_{Adj}}{df_{MST}} = 197.85$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle - 1 = 14$	$MSE_{Adj} = \frac{SSE_{Adj}}{df_{MSE}} = 201.49$

Table U.6 Experiment 1 ANOCOV with Time: Component Level, Equal Size Samples - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST_{Adj}}{MSE_{Adj}} = 0.98$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,14) = 3.74$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of covariance of total time and change impact at the *routine level* with *comparative sizing* using *equalized* sample sizes:

Table U.7 Experiment 1 ANOCOV with Time: Routine Level with Comparative Sizing, Equal Size Samples - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	6

Table U.8 Experiment 1 ANOCOV with Time: Routine Level with Comparative Sizing, Equal Size Samples - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST_{Adj} = \frac{SST_{Adj}}{df_{MST}} = 74.84$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle - 1 = 14$	$MSE_{Adj} = \frac{SSE_{Adj}}{df_{MSE}} = 83.49$

Table U.9 Experiment 1 ANOCOV with Time: Routine Level with Comparative Sizing, Equal Size Samples - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST_{Adj}}{MSE_{Adj}} = 0.90$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,14) = 3.74$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of covariance of the largest program written and change impact at the *routine level* using *equalized* sample sizes:

Table U.10 Experiment 1 ANOCOV with Largest Program Written: Routine Level, Equal Size Samples - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	6

Table U.11 Experiment 1 ANOCOV with Largest Program Written: Routine Level, Equal Size Samples - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST_{Adj} = \frac{SST_{Adj}}{df_{MST}} = 27.17$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle - 1 = 14$	$MSE_{Adj} = \frac{SSE_{Adj}}{df_{MSE}} = 63.32$

Table U.12 Experiment 1 ANOCOV with Largest Program Written: Routine Level, Equal Size Samples - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST_{Adj}}{MSE_{Adj}} = 0.43$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2{,}14) = 3.74$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of covariance of the largest program written and change impact at the *component level* using *equalized* sample sizes:

Table U.13 Experiment 1 ANOCOV with Largest Program Written: Component Level, Equal Size Samples - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	6

Table U.14 Experiment 1 ANOCOV with Largest Program Written: Component Level, Equal Size Samples - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST_{Adj} = \frac{SST_{Adj}}{df_{MST}} = 80.58$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle - 1 = 14$	$MSE_{Adj} = \frac{SSE_{Adj}}{df_{MSE}} = 241.69$

Table U.15 Experiment 1 ANOCOV with Largest Program Written: Component Level, Equal Size Samples - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST_{Adj}}{MSE_{Adj}} = 0.33$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,14) = 3.74$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of covariance of the largest program written and change impact at the *routine level* with *comparative sizing* using *equalized* sample sizes:

Table U.16 Experiment 1 ANOCOV with Largest Program Written: Routine Level with Comparative Sizing, Equal Size Samples - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	6

Table U.17 Experiment 1 ANOCOV with Largest Program Written: Routine Level with Comparative Sizing, Equal Size Samples - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST_{Adj} = \frac{SST_{Adj}}{df_{MST}} = 71.79$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle - 1 = 14$	$MSE_{Adj} = \frac{SSE_{Adj}}{df_{MSE}} = 88.99$

Table U.18 Experiment 1 ANOCOV with Largest Program Written: Routine Level with Comparative Sizing, Equal Size Samples - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST_{Adj}}{MSE_{Adj}} = 0.81$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,14) = 3.74$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of covariance of the number of programming courses and change impact at the *routine level* using *equalized* sample sizes:

Table U.19 Experiment 1 ANOCOV with Number of Programming Courses: Routine Level, Equal Size Samples - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	6

Table U.20 Experiment 1 ANOCOV with Number of Programming Courses: Routine Level, Equal Size Samples - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST_{Adj} = \frac{SST_{Adj}}{df_{MST}} = 33.40$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle - 1 = 14$	$MSE_{Adj} = \frac{SSE_{Adj}}{df_{MSE}} = 59.56$

Table U.21 Experiment 1 ANOCOV with Number of Programming Courses: Routine Level, Equal Size Samples - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST_{Adj}}{MSE_{Adj}} = 0.56$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,14) = 3.74$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of covariance of the number of programming courses and change impact at the *component level* using *equalized* sample sizes:

Table U.22 Experiment 1 ANOCOV with Number of Programming Courses: Component Level, Equal Size Samples - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	6

Table U.23 Experiment 1 ANOCOV with Number of Programming Courses: Component Level, Equal Size Samples - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST_{Adj} = \frac{SST_{Adj}}{df_{MST}} = 100.74$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle - 1 = 14$	$MSE_{Adj} = \frac{SSE_{Adj}}{df_{MSE}} = 231.79$

Table U.24 Experiment 1 ANOCOV with Number of Programming Courses: Component Level, Equal Size Samples - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST_{Adj}}{MSE_{Adj}} = 0.43$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,14) = 3.74$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of covariance of the number of programming courses and change impact at the *routine level* with *comparative sizing* using *equalized* sample sizes:

Table U.25 Experiment 1 ANOCOV with Number of Programming Courses: Routine Level with Comparative Sizing, Equal Size Samples - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	6

Table U.26 Experiment 1 ANOCOV with Number of Programming Courses: Routine Level with Comparative Sizing, Equal Size Samples - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST_{Adj} = \frac{SST_{Adj}}{df_{MST}} = 86.38$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle - 1 = 14$	$MSE_{Adj} = \frac{SSE_{Adj}}{df_{MSE}} = 88.39$

Table U.27 Experiment 1 ANOCOV with Number of Programming Courses: Routine Level with Comparative Sizing, Equal Size Samples - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST_{Adj}}{MSE_{Adj}} = 0.98$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,14) = 3.74$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Appendix V Correlation Between Structural Complexity Measures and Change Impact for Each Treatment Group

Table V.1 Correlation between structural complexity measures and mean change impact across all changes and within the Control Group (6 designs).

Structural Complexity Measure	Correlation to Mean Change Impact at the Routine Level	Correlation to Mean Change Impact at the Component Level	Correlation to Mean Change Impact at the Routine Level With Comparative Sizing
Mean Number of Routine Attributes Per Routine	-0.49	-0.27	-0.58
Mean Routine Size	-0.01	0.22	-0.10
Mean Number of Calls to Other Routines Per Routine	0.95	0.97	0.95
Mean Routine V(G)	0.61	0.74	0.53
Mean Number of Component Level Attributes Per Component	0.27	0.09	0.32
Mean Number of Routines Per Component	0.32	0.24	0.34
Mean Component Size	-0.06	0.03	-0.10
Mean Fan-In Per Component	-0.88	-0.76	-0.86
Mean Fan-Out Per Component	-0.88	-0.88	-0.83
Mean Number of Calls to External Routines Per Component	0.70	0.57	0.76
Mean Component V(G)	0.45	0.45	0.42
Number of Routines in the System	-0.94	-0.83	-0.98
Number of Components in the System	-0.90	-0.80	-0.92
System Size	-0.18	-0.24	-0.05
System V(G)	-0.49	-0.33	-0.58
Comparative System Size	-0.71	-0.66	-0.79
Comparative System V(G)	-0.50	-0.49	-0.60

Table V.2 Correlation between structural complexity measures and mean change impact across all changes and within the Rationale Group (12 designs).

Structural Complexity Measure	Correlation to Mean Change Impact at the Routine Level	Correlation to Mean Change Impact at the Component Level	Correlation to Mean Change Impact at the Routine Level With Comparative Sizing
Mean Number of Routine Attributes Per Routine	-0.29	-0.43	-0.29
Mean Routine Size	0.36	-0.13	0.30
Mean Number of Calls to Other Routines Per Routine	0.47	-0.06	0.41
Mean Routine V(G)	0.52	-0.09	0.41
Mean Number of Component Level Attributes Per Component	-0.14	-0.08	-0.10
Mean Number of Routines Per Component	-0.57	0.09	-0.49
Mean Component Size	-0.20	0.02	-0.07
Mean Fan-In Per Component	-0.05	-0.39	-0.28
Mean Fan-Out Per Component	0.10	-0.28	-0.03
Mean Number of Calls to External Routines Per Component	0.06	0.00	0.17
Mean Component V(G)	-0.04	0.13	0.04
Number of Routines in the System	-0.54	-0.45	-0.67
Number of Components in the System	-0.13	-0.54	-0.37
System Size	-0.36	-0.43	-0.43
System V(G)	-0.23	-0.40	-0.36
Comparative System Size	-0.15	-0.47	-0.07
Comparative System V(G)	0.01	0.29	0.27

Table V.3 Correlation between structural complexity measures and mean change impact across all changes and within the Rationale+Method group (8 designs).

Structural Complexity Measure	Correlation to Mean Change Impact at the Routine Level	Correlation to Mean Change Impact at the Component Level	Correlation to Mean Change Impact at the Routine Level With Comparative Sizing
Mean Number of Routine Attributes Per Routine	0.37	0.81	0.68
Mean Routine Size	0.70	0.88	0.88
Mean Number of Calls to Other Routines Per Routine	0.77	0.86	0.86
Mean Routine V(G)	0.71	0.77	0.89
Mean Number of Component Level Attributes Per Component	-0.25	-0.72	-0.46
Mean Number of Routines Per Component	-0.11	0.50	0.14
Mean Component Size	0.41	0.88	0.70
Mean Fan-In Per Component	-0.87	-0.95	-0.95
Mean Fan-Out Per Component	-0.42	-0.47	-0.55
Mean Number of Calls to External Routines Per Component	0.65	0.87	0.73
Mean Component V(G)	0.64	0.92	0.83
Number of Routines in the System	-0.86	-0.75	-0.95
Number of Components in the System	-0.83	-0.87	-0.94
System Size	-0.51	-0.16	-0.50
System V(G)	-0.46	-0.15	-0.31
Comparative System Size	-0.40	-0.62	-0.69
Comparative System V(G)	0.03	-0.08	-0.08

Appendix W Experiment 2: Analysis of Variance (ANOVA) for Change Impact with the New RVM Design

Analysis of variance for the impacted data across all types of expected change using *unequal* sample sizes and *unweighted means analysis*:

Table W.1 Experiment 2 RVM ANOVA: Impacted Data, Unequal Size Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=18, n_2=8, n_3=14

Table W.2 Experiment 2 RVM ANOVA: Impacted Data, Unequal Size Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 56.19$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 37$	$MSE = \frac{SSE}{df_{MSE}} = 247.39$

Table W.3 Experiment 2 RVM ANOVA: Impacted Data, Unequal Size Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.23$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,37) = 3.26$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the impacted data across all types of expected change using *unequal* sample sizes and *weighted means analysis*:

Table W.4 Experiment 2 RVM ANOVA: Impacted Data, Unequal Size Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=18, n_2=8, n_3=14

Table W.5 Experiment 2 RVM ANOVA: Impacted Data, Unequal Size Samples, Weighted Means Analysis Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 84.03$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 37$	$MSE = \frac{SSE}{df_{MSE}} = 247.39$

Table W.6 Experiment 2 RVM ANOVA: Impacted Data, Unequal Size Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.34$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,37) = 3.26$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance for the impacted methods across all types of expected change using *unequal* sample sizes and *unweighted means analysis*:

Table W.7 Experiment 2 RVM ANOVA: Impacted Methods, Unequal Size Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=18, n_2=8, n_3=14

Table W.8 Experiment 2 RVM ANOVA: Impacted Methods, Unequal Size Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 375{,}504.05$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 37$	$MSE = \frac{SSE}{df_{MSE}} = 306{,}843.38$

Table W.9 Experiment 2 RVM ANOVA: Impacted Methods, Unequal Size Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 1.22$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,37) = 3.26$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the impacted methods across all types of expected change using *unequal* sample sizes and *weighted means analysis*:

Table W.10 Experiment 2 RVM ANOVA: Impacted Methods, Unequal Size Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=18, n_2=8, n_3=14

Table W.11 Experiment 2 RVM ANOVA: Impacted Methods, Unequal Size Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 548{,}814.88$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 37$	$MSE = \frac{SSE}{df_{MSE}} = 306{,}843.38$

Table W.12 Experiment 2 RVM ANOVA: Impacted Methods, Unequal Size Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 1.79$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,37) = 3.26$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance for the impacted classes across all types of expected change using *unequal* sample sizes and *unweighted means analysis*:

Table W.13 Experiment 2 RVM ANOVA: Impacted Classes, Unequal Size Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=18, n_2=8, n_3=14

Table W.14 Experiment 2 RVM ANOVA: Impacted Classes, Unequal Size Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 1{,}025{,}251.55$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 37$	$MSE = \frac{SSE}{df_{MSE}} = 546{,}704.86$

Table W.15 Experiment 2 RVM ANOVA: Impacted Classes, Unequal Size Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 1.88$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,37) = 3.26$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance for the impacted classes across all types of expected change using *unequal* sample sizes and *weighted means analysis*:

Table W.16 Experiment 2 RVM ANOVA: Impacted Classes, Unequal Size Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=18, n_2=8, n_3=14

Table W.17 Experiment 2 RVM ANOVA: Impacted Classes, Unequal Size Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 1{,}004{,}607.63$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 37$	$MSE = \frac{SSE}{df_{MSE}} = 546{,}704.86$

Table W.18 Experiment 2 RVM ANOVA: Impacted Classes, Unequal Size Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 1.84$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,37) = 3.26$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance for the impacted files across all types of expected change using *unequal* sample sizes and *unweighted means analysis*:

Table W.19 Experiment 2 RVM ANOVA: Impacted Files, Unequal Size Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=18, n_2=8, n_3=14

Table W.20 Experiment 2 RVM ANOVA: Impacted Files, Unequal Size Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 31{,}712{,}115.48$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 37$	$MSE = \frac{SSE}{df_{MSE}} = 12{,}697{,}138.82$

Table W.21 Experiment 2 RVM ANOVA: Impacted Files, Unequal Size Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 2.50$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,37) = 3.26$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_{\alpha}$.

Analysis of variance for the impacted files across all types of expected change using *unequal* sample sizes and *weighted means analysis*:

Table W.22 Experiment 2 RVM ANOVA: Impacted Files, Unequal Size Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=18, n_2=8, n_3=14

Table W.23 Experiment 2 RVM ANOVA: Impacted Files, Unequal Size Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 47{,}181{,}762.80$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 37$	$MSE = \frac{SSE}{df_{MSE}} = 12{,}697{,}138.82$

Table W.24 Experiment 2 RVM ANOVA: Impacted Files, Unequal Size Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 3.72$
F_{α}	$F_{\alpha}(df_{MST}, df_{MSE}) = F_{0.05}(2,37) = 3.26$
Accept/Reject H_0	Reject H_0 since $F_{Calculated} > F_{\alpha}$.

Appendix X Experiment 2: Analysis of Variance (ANOVA) for Change Impact with the New Kernel-Venus Interface Design

Analysis of variance for the impacted data across all types of expected change using *unequal* sample sizes and *unweighted means analysis*:

Table X.1 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Data, Unequal Sized Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=16, n_2=8, n_3=14

Table X.2 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Data, Unequal Sized Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 19{,}128.74$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 35$	$MSE = \frac{SSE}{df_{MSE}} = 62{,}916.32$

Table X.3 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Data, Unequal Sized Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.30$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,35) = 3.28$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the impacted data across all types of expected change using *unequal* sample sizes and *weighted means analysis*:

Table X.4 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Data, Unequal Sized Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=16, n_2=8, n_3=14

Table X.5 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Data, Unequal Sized Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 19{,}422.81$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 35$	$MSE = \frac{SSE}{df_{MSE}} = 62{,}916.332$

Table X.6 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Data, Unequal Sized Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.31$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,35) = 3.28$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the impacted methods across all types of expected change using *unequal* sample sizes and *unweighted means analysis*:

Table X.7 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Methods, Unequal Sized Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=16, n_2=8, n_3=14

Table X.8 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Methods, Unequal Sized Samples, Unweighted Means Analysis -Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g-1 = 2$	$MST = \frac{SST}{df_{MST}} = 20{,}235.54$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 35$	$MSE = \frac{SSE}{df_{MSE}} = 532{,}250.59$

Table X.9 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Methods, Unequal Sized Samples, Unweighted Means Analysis -Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.04$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,35) = 3.28$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the impacted methods across all types of expected change using *unequal* sample sizes and *weighted means analysis*:

Table X.10 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Methods, Unequal Sized Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=16, n_2=8, n_3=14

Table X.11 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Methods, Unequal Sized Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 20{,}355.45$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 35$	$MSE = \frac{SSE}{df_{MSE}} = 532{,}250.59$

Table X.12 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Methods, Unequal Sized Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 0.04$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,35) = 3.28$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the impacted classes across all types of expected change using *unequal* sample sizes and *unweighted means analysis*:

Table X.13 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Classes, Unequal Sized Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=16, n_2=8, n_3=14

Table X.14 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Classes, Unequal Sized Samples, Unweighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 1{,}957{,}231.94$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 35$	$MSE = \frac{SSE}{df_{MSE}} = 923{,}371.39$

Table X.15 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Classes, Unequal Sized Samples, Unweighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 2.12$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,35) = 3.28$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the impacted classes across all types of expected change using *unequal* sample sizes and *weighted means analysis*:

Table X.16 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Classes, Unequal Sized Samples, Weighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=16, n_2=8, n_3=14

Table X.17 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Classes, Unequal Sized Samples, Weighted Means Analysis - Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 1{,}969{,}773.82$
Error or Unexplained	SSE	$df_{MSE} = g \sum_{i=1}^{g} \langle n_i - 1 \rangle = 35$	$MSE = \frac{SSE}{df_{MSE}} = 923{,}371.39$

Table X.18 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Classes, Unequal Sized Samples, Weighted Means Analysis - Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 2.13$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,35) = 3.28$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the impacted files across all types of expected change using *unequal* sample sizes and *unweighted means analysis*:

Table X.19 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Files, Unequal Sized Samples, Unweighted Means Analysis - Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=16, n_2=8, n_3=14

Table X.20 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Files, Unequal Sized Samples, Unweighted Means Analysis -Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g - 1 = 2$	$MST = \frac{SST}{df_{MST}} = 21{,}015{,}862.87$
Error or Unexplained	SSE	$df_{MSE} = g\sum_{i=1}^{g}\langle n_i - 1\rangle = 35$	$MSE = \frac{SSE}{df_{MSE}} = 8{,}344{,}854.38$

Table X.21 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Files, Unequal Sized Samples, Unweighted Means Analysis -Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 2.52$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,35) = 3.28$
Accept/Reject H_0	Accept H_0 since $F_{Calculated} < F_\alpha$.

Analysis of variance for the impacted files across all types of expected change using *unequal* sample sizes and *weighted means analysis*:

Table X.22 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Files, Unequal Sized Samples, Weighted Means Analysis -Parameters used to calculate the F statistic.

Parameter Name	Parameter Meaning	Parameter Value
g	number of treatment groups	3
n	number of subjects in a treatment group	n_1=16, n_2=8, n_3=14

Table X.23 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Files, Unequal Sized Samples, Weighted Means Analysis -Calculations for determining the F statistic.

Type of Variation	Related Sum of Squares	Related Degrees of Freedom	Related Mean Square
Explained by Treatments	SST	$df_{MST} = g-1 = 2$	$MST = \frac{SST}{df_{MST}} = 27,614,558.97$
Error or Unexplained	SSE	$df_{MSE} = g\sum_{i=1}^{g}\langle n_i - 1\rangle = 35$	$MSE = \frac{SSE}{df_{MSE}} = 8,344,854.38$

Table X.24 Experiment 2 Kernel-Venus Interface ANOVA: Impacted Files, Unequal Sized Samples, Weighted Means Analysis -Testing the H_0 hypothesis.

Name of Statistic or Decision	Determination of Statistic or Decision
$F_{Calculated}$	$F_{Calculated} = \frac{MST}{MSE} = 3.31$
F_α	$F_\alpha(df_{MST}, df_{MSE}) = F_{0.05}(2,35) = 3.28$
Accept/Reject H_0	Reject H_0 since $F_{Calculated} > F_\alpha$.