EVOLVING LEGACY SYSTEM FEATURES INTO FINE-GRAINED COMPONENTS

by

Alok Mehta

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Computer Science

November 2002

APPROVED: [illegible]

Dr. George T. Heineman, Computer Science Department, WPI, Major Advisor

[illegible]

Dr. Elke A. Rundensteiner, Computer Science Department, WPI

[illegible]

Mr. Mike Ciaraldi, Computer Science Department, WPI

[illegible]

Dr. Alexander L. Wolf, Computer Science, University of Colorado at Boulder

Abstract

Because many software systems used for business today are considered legacy systems, the need for software evolution techniques has never been greater. We propose a novel evolution methodology for legacy systems that integrates the concepts of features, regression testing, and Component-Based Software Engineering (CBSE). Regression test suites are untapped resources that contain important information about the features of a software system. By exercising each feature with its associated test cases using code profilers and similar tools, code can be located and refactored to create components. The unique combination of Feature Engineering and CBSE makes it possible for a legacy system to be modernized quickly and affordably. We develop a new framework to evolve legacy software that maps the features to fine-grained software components refactored from their feature implementation. In this dissertation, we make the following contributions: First, a new methodology to evolve legacy code is developed that improves the maintainability of evolved legacy systems. Second, the technique describes a clear understanding between features and functionality, and relationships among features using our feature model. Third, the methodology provides guidelines to construct feature-based reusable components using our fine-grained component model. Fourth, we bridge the complexity gap by identifying feature-based test cases and developing feature-based reusable components. We show how to reuse existing tools to aid the evolution of legacy systems rather than re-writing special purpose tools for program slicing and requirement management. We have validated our approach on the evolution of a real-world legacy system. By applying this methodology, American Financial Systems, Inc. (AFS), has successfully restructured its enterprise legacy system and reduced the costs of future maintenance.

Acknowledgments

Writing a dissertation is a long tedious process. It requires tremendous amount of hard work, self-discipline, sacrifice and support from others. I have been very fortunate to have several people in my life that supported me in many ways to help complete this dissertation. Without following people’s help, I would not have been able to finish this work:

I am grateful to my advisor Dr. George T. Heineman. It has been serendipitous and fateful that I have come to know him. His guidance, attention to detail, hard work, friendly demeanor and ability to motivate has helped me professionally, academically and personally all these years. I would not have been able to complete this dissertation without his help, support and valuable input.

I am thankful to AFS for supporting my academic career. AFS has provided partial funding and a fertile ground to apply research ideas. There are several co-workers at AFS who have provided valuable input and support in my research endeavors. In particular, I am grateful to Dan Johnson, Lisa Amaya Price, Mary Mullay and German Rincon.

I am thankful to my father Dr. M. C. Mehta for being a role model. I wish I were a medical Dr. like him. I am proud to be his son and I appreciate his support, love and interest in my academic career. I am also grateful to my parents (both, Indian and American) and friends for believing in me and constantly providing support. Special thanks goes to Alex Rose and Suzie Gath for their help and support.

Lastly but certainly not the least, I am so very grateful to my wife Heather. No words can express my gratitude towards her. She has been a pillar of support for all these years. Her belief, support, help, patience, sacrifice and love have allowed me to pursue the PhD program. Heather – “you are the wind beneath my wings”. I dedicate this dissertation to you and our kids Rani and Amber!

To Heather, Rani and Amber

Table of Content

List of Figures

List of Tables

1 Introduction

Increasingly, organizations view their software assets as investments that grow in value rather than liabilities whose value depreciates over time [97]. Organizations are under tremendous pressure to evolve their existing systems to better respond to marketplace need and stay competitive. This constant pressure to evolve is driven by escalating expectations of the customer for new enterprise standards, new products and system features, and improved performance. Evolution is also necessary to cope with endless new software releases.

We borrow the definition of legacy system from [114]:

> ***Any software system that is currently in operation is considered legacy system.***

Legacy systems provide the support for businesses around the world. They manage vast volumes of data while supporting millions of transactions each day. The National Science Foundation [54] estimates that legacy systems capture and manage 75% of the world's data and that by virtue of their size and importance to business, they consume at least 80% of available information technology resources. To effectively evolve legacy systems in such a rapidly changing environment organizations must answer two questions [31]: What are the critical success factors of system evolution? How can a system be evolved without adversely affecting operations and revenue?

When legacy systems are small and involve only a fraction of an organization's activities, it is possible to consider redesigning and replacing a system or subsystem that no longer satisfies that organization's needs. However, legacy systems that have grown to be the main source of revenue are often substantial investments whose replacement is more difficult, if not impossible. These legacy systems provide a competitive advantage to many organizations but are expensive to maintain. Thus, these organizations face a dilemma - they cannot afford lose their competitive advantage nor can they ignore the high maintenance cost. At the same time, organizations are under pressure to reduce costs. This dissertation is motivated by these pressing business concerns.

1.1 Managing Legacy Systems

There are many strategies for managing legacy systems [98][64][83]:

1.1.1 Status quo

Do nothing. This is the easiest option and, in reality, most often chosen by an organization. However, this option is not attractive to many organizations because it will not improve their competitive edge in the future and leaves legacy systems maintenance costs high.

1.1.2 Rewrite legacy system

Sometimes an organization will embrace new development and deployment technology to rewrite the legacy system. Apart from using the legacy system to be retired as a "design guide", this option does not leverage off the organization's substantial investment in the prior system. Redevelopment of large "mission critical" legacy systems takes a long time, costs a lot of money, and carries a high degree of risk of failure. In most cases, it is very difficult to build a strong business case for the redevelopment of a legacy system.

1.1.3 Replace legacy system

Replacing a legacy system with another existing solution can be a practical option when the proposed solution provides a good functional fit to the business requirements of an organization. Rarely is this the case, however. Most often, the existing solution requires considerable enhancement and customization in order for it to meet business needs. This customization is generally difficult and expensive. Alternatively, at the loss of competitive advantage, the organization can change its business practices to fit the proposed solution, which may be a risky proposition. Replacement does not leverage off the current investment in the legacy system(s) to be retired. Finally, adopting a proposed solution can be a lengthy and expensive exercise.

1.1.4 Incrementally evolve legacy system

Incremental evolution of legacy systems focuses on problems that are most visible to end-users. Rather than replacing or rewriting the entire legacy system, incremental evolution directly "fixes" the end-users' problems "one at a time". This option leverages off current investment because it provides a smooth transition path to new technology and infrastructure in a timely and cost-effective manner. Importantly, incremental evolution supports the needs of organizations to continually provide stability and accuracy. Incremental evolution is the only choice left to many organizations that wish to continue to receive revenue from software systems and stay competitive. However, many incremental evolution initiatives do not sufficiently incorporate the end-user's point of reference (or features) [111]; such lack of consideration can leave end-users unsatisfied and frustrated because they may not see the benefit of these initiatives.

1.1.5 Summary

Analyzing all options at hand to manage legacy systems it is clear that incremental evolution is the best because it considers perspectives of the end-user's and all stakeholders point of view. Thus, we strongly believe that incrementally evolving legacy system is the most efficient option for managing legacy system.

1.2 Problem description

Researchers [116][82][47][3][4] have identified the two domains around which the entire field of software engineering revolves: the *problem domain* and the *solution domain*. End-users interact with the system by inputting their information in the form of input files that the system uses or through a direct user interface. Because these users are directly concerned with system features, their perspective is always in the problem domain. Developers (and the software process team) are primarily concerned with creating and maintaining software development life cycle artifacts such as components; their perspective is therefore firmly rooted in the solution domain.

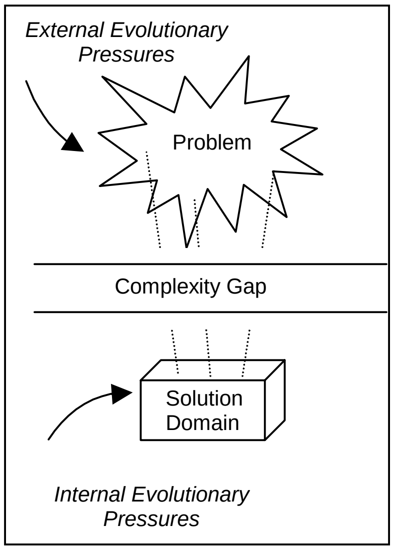

External Evolutionary Pressures
Problem
Complexity Gap
Solution Domain
Internal Evolutionary Pressures

Figure 1.1: An Incremental Evolution Methodology is Needed.

A major source of difficulty in developing, delivering, and evolving successful software is the *complexity gap* that exists between the problem and the solution domains (as termed by Raccoon [82]) as shown in Figure 1.1. To view evolution from a single domain upsets the delicate balance between the two domains. Evolution focused solely on the problem domain may lead to changes that degrade the structure of the original code; similarly, evolution based solely on technical merits could create changes unacceptable to end-users. External evolutionary pressures drive the implementation of new enhancements and functionality by causing developers to focus on implementing the business logic that is *directly* visible to end-users, such as a menu item that spell checks a document in a word processing application. While responding to external pressures, developers often bypass standard processes to meet project deadlines; this results in inferior coding, such as adding a global variable when one is not required. Internal evolutionary pressures force the developers to either restructure or refactor their code so the future enhancement or maintenance becomes manageable and cost-effective. During such evolution, the code is refactored, and protocols and standards are reestablished. Furthermore, the end-users should always benefit from the evolution initiatives.

The repeated modification of a legacy system has a cumulative effect that increases system complexity because of lack of documentation and implicit communication between the system's components. Eventually, existing information systems become too fragile to modify and too important to discard; organizations must consider modernizing these legacy systems so that they remain viable. Incremental evolution offers an approach to transforming a legacy system into one that can evolve in a disciplined manner. To be successful, evolution requires insights from software, managerial, and economic perspectives [114]. Thus, businesses must sponsor and endorse evolution initiatives. Such endorsement becomes easier if the end-user's perspective is kept as primary focus of the evolution initiative. Yet, another way to secure organization's endorsement is to show that the evolution initiative can result in reusable software assets. In cases where businesses have multiple product lines, it is desirable to leverage evolution initiatives from one legacy system to another [87]. One such way of leveraging is sharing reusable software assets such as components.

Simply stated, the problem is that businesses are looking for an incremental evolution methodology that can reduce future maintenance cost, bridge the complexity gap and leverage evolution results across product lines, without disrupting their operations.

1.3 Motivation

We are motivated by the following three objectives:

- Evolve system features into components to reduce future maintenance costs.
- Reuse evolved components in multiple product lines.
- Reduce the complexity gap between user expectations and software functionality.

1.3.1 Reduce future maintenance costs

One objective of this research is to reduce the maintenance cost of features that are hard to maintain. We identify these system features with the help of end-users, locate their implementation within the source code and then evolve them into reusable units. In one of the first dissertations on Feature Engineering, Turner [25] mentioned the possibility of using Feature Engineering for software evolution, but he was focused on using features for configuration management. To the best of our knowledge, there has been no attempt to use Feature

Engineering as the basis for identifying parts of legacy software for evolution purposes. We have developed techniques for identifying evolvable features with high maintenance costs to be refactored into reusable software components. In the accompanying case study discussed in detail in Chapter 6 for this dissertation, we show that our methodology reduces maintenance costs.

1.3.2 Reuse components in multiple product lines

The Internet makes it possible for an organization to attract potential customers from the global marketplace by breaking down communication barriers. The Internet also increases the need to evolve and refactor legacy systems to new hardware and software development platforms. Evolving a legacy software system to become web-enabled is a challenging task for numerous reasons, including poor documentation and high maintenance costs. One challenge for evolving a legacy system into a web-enabled system is the need to provide continuous availability of the system (and thus its revenue-generating income) during the transition. Often organizations must support the two product lines (desktop and the Internet) longer than expected, so there is a need for an evolution methodology that reduces the maintenance cost during the migration period.

1.3.3 Reduce the complexity gap

End-users interact with the system and are directly concerned with its functionality; their perspective is always in the problem domain. Developers (and the rest of the software process team) are concerned with the creation and maintenance of software development life cycle artifacts such as components and executables; their perspective is rooted in the solution domain. One of the effects of this gap is that changes are often required to features after software is released thereby increasing maintenance costs. We are motivated to bridge the complexity gap by mapping problem-domain features and the solution-domain functions in the source code.

1.4 Our Approach

Various domain analysis and requirements engineering techniques push the end-user's perspective into the solution domain by either working toward design [77][22][110] or through scenario and use cases [47][85][36]. These solutions help the developers understand how a system is to be used, but they do not address the solution domain concerns of software evolution, configuration management, testing, and documentation. In addition, these techniques certainly do not address the important issue of reducing the complexity gap [82].

Similarly, many software evolution techniques exist [97][111][64][105][80], but none considered Feature Engineering as a software evolution driver. This is a serious oversight because Feature Engineering is a promising discipline that can help to reduce the complexity gap between user expectations and software functionality. The techniques of software evolution and reengineering either focus on entire system rewrites [117] or using reverse reengineering for comprehension purposes [113] rather than incrementally evolving the legacy system. There are other techniques [111][114][64][105][80], but they all explore the solution domain only. There has been no attempt made to use Feature Engineering as the basis for identifying parts of legacy software for evolution purposes. Current software evolution and reengineering techniques continue to work in the solution domain. The important problem of linking the problem domain and the solution domain for the purposes of evolution remains unsolved.

Component-Based Software Engineering (CBSE) offers promising techniques to solve the problem of component construction [2], but CBSE has not yet been connected to the features that are present in a system; creating this connection explicitly is one of the contributions of this dissertation. This connection, in essence, is a mapping problem. The functionality provided by CBSE solutions must be mapped to the features available to the end-user.

We have developed a novel evolution methodology that integrates the concepts of features, regression tests, and CBSE. Regression test suites are untapped resources that contain important information about the features for a software system. CBSE is one of the best techniques for engineering and reengineering modular systems. Combining these two disciplines makes it possible for a legacy system to be modernized quickly and affordably. By combining Feature Engineering and CBSE to the problem of software evolution, this dissertation will answer the following questions:

1. How can features be used to create components in a legacy system?

2. How can the complexity gap be reduced using features and components?
3. What is a feature and how is a feature related to functions within the source code?

Our methodology answers these questions based on two important goals: **(G1)** Identify system features that have already exhibited disproportionate maintenance costs and are likely to change. **(G2)** Extract fine-grained components from these features within the legacy system to share between the original desktop platform and a planned web application.

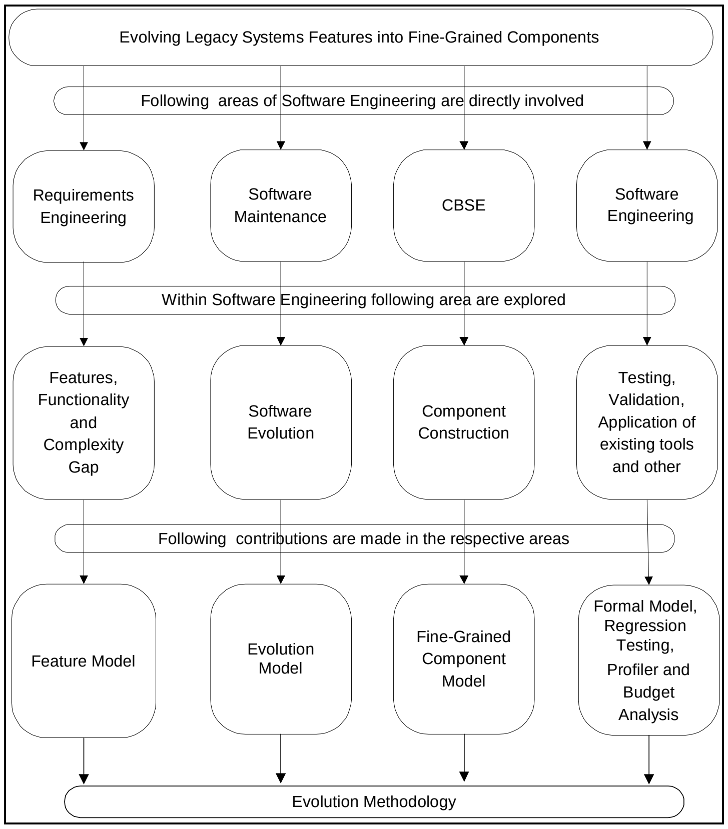

Evolving Legacy Systems Features into Fine-Grained Components
Following areas of Software Engineering are directly involved
Requirements Engineering
Software Maintenance
CBSE
Software Engineering
Within Software Engineering following area are explored
Features, Functionality and Complexity Gap
Software Evolution
Component Construction
Testing, Validation, Application of existing tools and other
Following contributions are made in the respective areas
Feature Model
Evolution Model
Fine-Grained Component Model
Formal Model, Regression Testing, Profiler and Budget Analysis
Evolution Methodology

Figure 1.2: The Big Picture.

1.5 Scope

The overall scope of this dissertation is summarized in Figure 1.2. Our work revolves around four areas of software engineering, namely requirements engineering, software maintenance, CBSE, and general software engineering practice. Specifically, we use ideas from Feature Engineering, Testing, and CBSE to develop an evolution methodology. Our evolution methodology consists of a feature model and component model and is supported by various software engineering practices such code coverage tools. Furthermore, we provide a solid foundation for these models using relational calculus and first order logic. The methodology proposed in this dissertation does not reduce the complexity of a legacy system, but it will help to clarify that complexity by explicitly defining component interfaces.

To further increase productivity and demonstrate the immediate usability of the techniques outlined above, we use several tools that are already available in the marketplace, in particular, the NuMega ® True Time Code Profiler [56].

While several disciplines of software engineering are related to this dissertation, we are primarily concerned with developing a software evolution methodology using Feature Engineering and CBSE. This dissertation will not address all the issues associated with requirements engineering, CBSE, Testing or Configuration Management. The two main areas of software engineering that are directly addressed through this methodology are:

1.5.1 Locating system features

Our methodology enables developers to trace functions within the source code that implement particular feature(s) by running regression test cases. We incorporate ideas from Feature Engineering, regression test cases and dynamic slicing. This feature-function mapping can be used for program understanding by identifying and associating structures that were previously ambiguous. However, program understanding is outside the scope of our work. Likewise, our dissertation is not about issues in testing and dynamic slicing, we simply use regression test cases to locate feature-based program slices.

1.5.2 Evolving features into components:

Once feature implementations are located, we evolve them into components using refactoring and CBSE techniques. We use Fowler's [86] definition of refactoring code: "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." Our methodology is novel because we leverage off existing artifacts such as test cases and code profiling utilities. While we do not address the issue of architectural evolution, our methodology does produce better-structured component-based code that can be evolved/maintained easily. Likewise, refactoring alone cannot be used to evolve a system. While we make use of some common refactoring techniques, this dissertation is not simply about refactoring code. The methodology further promotes scheduled evolution in a systematic way by clarifying the structure of program evolution, and its results are measurable and can be validated.

1.6 Assumptions

Our methodology has three basic assumptions. First, we assume that the source code for the legacy system to be evolved is available and that it was developed using a modern programming language such as Visual Basic, C++, Java, or Fortran. The reason for this assumption is that we use code-profiling tools for tracing the source that implements a particular feature. If these code-profiling tools are unavailable, they could be developed, but this is outside the scope of this dissertation. Second, we assume that the legacy system has regression test suites. These regression test suites are untapped resources from the evolution point of view because they can be used to identify the features most important to the end-users. Third, we assume that some domain knowledge and expertise is available, although this is not a binding constraint. The purpose of this dissertation is not to explore domain analysis; however, as a part of related work, several domain analysis techniques are discussed.

This methodology is not intended for all legacy systems, neither will all software evolution initiatives benefit from it. However, legacy systems that have kept up with their compiler upgrades and maintained over a decade or more will greatly benefit. We will discuss the characteristics of legacy systems and software processes that can make use of this methodology and benefit in Chapter 2.

1.7 Scope and Contributions

The overall scope of this dissertation is sketched in Figure 1.2 and the methodology is summarized in Figure 2.1. The major contribution of this dissertation is the evolution methodology that integrates Feature Engineering, software evolution, and CBSE. To validate our methodology, we examine the evolution of a real-world legacy system, American Financial Systems (AFS) Master System. Information about the legacy system’s features is obtained through interviews with testers, project managers and the end-users of the system. Researchers use refactoring to isolate the code associated with extracted features to create components. These components are then inserted back into the legacy system to continue functioning to verify the results of this technique. Our results show an innovative use of existing regression test suites and give extra incentives for designing and maintaining such test suites. In addition to verifying the integrity of the system, regression test suites can be used to guide refactoring efforts during software evolution to create reusable software assets (components) within the enterprise.

In this dissertation, we make the following contributions: First, a new methodology to evolve legacy code is developed that improves the maintainability by reducing maintenance costs of evolved legacy systems. Second, the technique describes a clear understanding between features and functionality, and relationships among features using our feature model. Understanding the interactions and relationships among features can bridge the complexity gap and aid in evolving feature(s) with high maintenance costs. Third, the methodology provides guidelines to construct feature-based reusable components using our fine-grained component model. These fine-grained components can then be reused across multiple product lines. Fourth, we bridge the complexity gap by identifying feature-based test cases and developing feature-based reusable components. Bridging this gap is important as it aligns user expectations and software functionality.

1.8 Road Map

The rest of the dissertation is organized as follows:

Chapter 1 outlines the problem and the scope of this work. Chapter 2 provides an overview of our methodology and discusses how the Internet plays an important role for reusing components. We also discuss the role of refactoring and regression test cases with respect to evolution. Chapter 3 presents a detailed analysis of related work. We compare our work to other evolution methodologies and techniques to locate program features. Chapter 4 provides the details on our feature model, fine-grained component model, budget analysis model, and the formal model. We also discuss Feature Engineering and its role in the evolution of legacy systems. Feature Engineering addresses the problem of the complexity gap in an explicit way [26]. Our feature model captures the relationships between features and functionality. The chapter also addresses traceability and how it can be used for program understanding within the context of Feature Engineering. Our fine-grained component model shows how features can be extracted to create fine-grained components. The budget analysis model provides elements that are required to show cost benefit of our evolution methodology. Chapter 5 provides an intuitive example to show the power of our methodology using a feature-based evolution manager utility that assists in identifying a fine-grained component's properties and methods. Chapter 6 contains a case study of product evolution in a software firm using our evolution methodology. Chapter 7 provides conclusions, lessons learned, and future work.

2 Methodology Overview

In this chapter, we present an overview of our methodology and discuss the factors that influenced its criteria, our methodology will be described in detail in Chapter 4. Here we briefly outline basic concepts and models that our methodology uses.

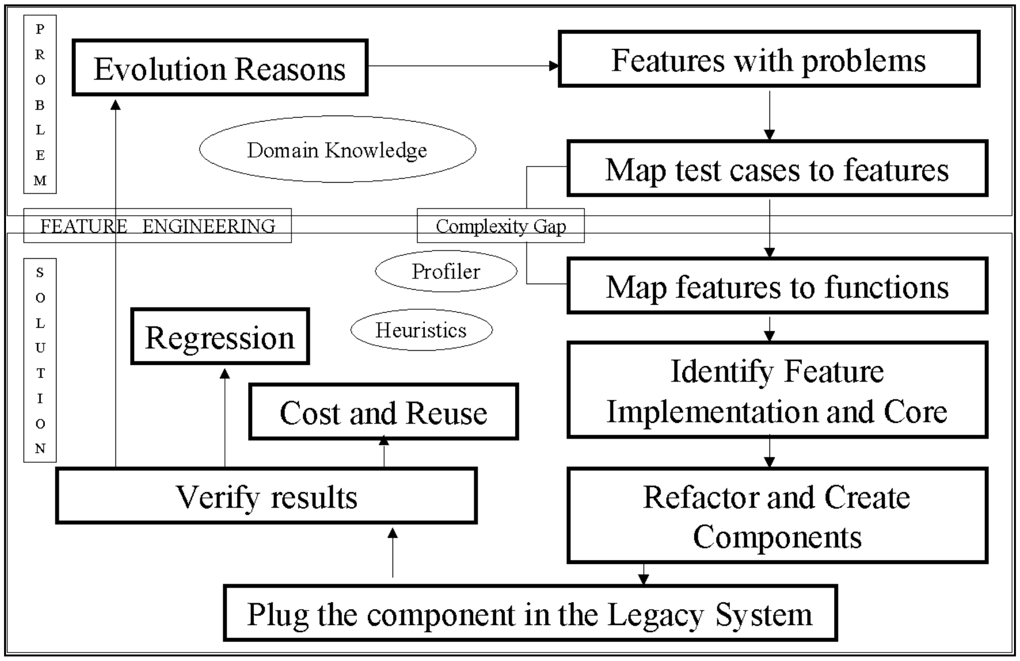

PROBLEM
Evolution Reasons
Features with problems
Domain Knowledge
Map test cases to features
FEATURE ENGINEERING
Complexity Gap
SOLUTION
Profiler
Map features to functions
Regression
Heuristics
Identify Feature Implementation and Core
Cost and Reuse
Verify results
Refactor and Create Components
Plug the component in the Legacy System

Figure 2.1: Evolution Methodology.

2.1 Methodology steps

Any evolution task must first examine the reasons that trigger evolution of a legacy system; these reasons have been well documented in [97][114] and from personal experience as a software engineer for over 10 years, the author agrees that these reasons are sound. We start with recommended changes to end-users features. The reasons for evolving the system are then mapped to their associated features within the system. The system features are then identified and the code that implements each feature is identified. The code is then extracted to create a fine-grained component. The fine-grained component is inserted back into the legacy system to validate results in three ways. First, we match the output of the regression tests after the insertion with original output. Second, we measure the cost of maintaining the feature after evolution and compare that to the prior costs (hopefully showing a cost-benefit). Third, the newly created components are reused in other product lines. The outline of our methodology is shown in Figure 2.1. Specifically, there are ten steps to our methodology.

2.1.1 Evolution reasons

The end-users work with testers and project managers to report the problems they are facing with a particular feature or a group of features. If features are common among product lines then they are likely be candidates for reuse.

2.1.2 Identify features with problems

The testers, project managers, and developers work together to identify the underlying feature that is the source of the maintenance problem(s).

2.1.3 Map test cases to features

Testers and end-users work together to select test cases from the regression test suite for the feature(s) with problem(s). This step also compares the execution trace of the selected test case with the entire regression test suite to ensure that we do not miss any critical test cases that may be needed.

2.1.4 Map features to functions

Selected test cases are executed using code profilers to locate source code that implements features. Features are well known for being “cross-cutting” through software [87]. Cross-cutting means that a function can implement many features and these features share the same code/data. A feature may be implemented in many functions and share code/data with other features. The close relationship between features and functions means that features will interact with each other. This interaction is also defined as the feature-interaction problem in the literature [87][116]. Our ***feature model*** helps to identify where features are located within the legacy system, how features are related to other features, and how they interact with each other.

2.1.5 Identify feature implementation and CORE

We analyze the data from the code profiler and the execution traces of the regression test cases. Using heuristics, we then decide if creating components will in fact benefit other product lines as well as the existing legacy system. While a detailed description of the feature model and heuristics is provided in Section 4.1.5, we briefly present in Table 2.1 the properties a feature should posses to be a candidate for evolution.

Properties	Description
Visible to end-user	End-user must be able to execute the test case himself and see the problems
Testable	Testers must have test cases in the regression test suite to test the feature
Exhibit recurring problems	Fixing feature inadvertently affects other features
High maintenance cost	Due to recurring problems and unwanted side affects the feature's maintenance cost is higher
Feature exists in other product line	Reuse can be leveraged

Table 2.1: Properties of Evolvable Features.

2.1.6 Refactor and create components

Once the location of a feature within the code is identified and we decide to extract the feature implementation into a component, we must refactor the code to create component. Our ***fine-grained component model*** provides guidelines to extract feature specific code/data. This code/data is then encapsulated in a fine-grained component. A detailed description of the feature model is provided in Section 4.1.5.

2.1.7 Plug the component in the legacy system

The developers integrate the components created in Step 2.1.6 in the target legacy system. Many common integration techniques can be used for this purpose.

2.1.8 Verify results

The testers ensure that the stability of the newly integrated system is maintained and that no side effects are introduced. The performance of the system is also compared. This step usually results in running a full regression test.

2.1.9 Reuse

This is similar to Step 2.1.7 except that integration is now performed in other product lines of the organization that share the feature that we have evolved.

2.1.10 Measure results

The project manager works with testers and the developers to use *Budget Analysis* to report results to end-users and management. We show how maintenance costs of the feature is reduced and how these feature-based components are reused across product lines. The budget analysis presents the cost and the benefit of applying the methodology.

2.2 Factors affecting methodology

Although our methodology is programming language-independent and does not depend on specific code profiler tools, several factors affect our methodology. These factors are as follows: uniqueness of legacy system, role of the Internet in evolution, regression testing process, code coverage tools and refactoring techniques.

2.2.1 Each legacy system is unique

In theory, our methodology is generic and can be applied to any legacy that meets the list of assumptions discussed in Section 1.6; however we have only applied this methodology to one large industrial sized application. As we describe this legacy system throughout this dissertation, we hope to convince the reader that our approach remains applicable to numerous other systems.

We now provide a brief description of the legacy system we used as our case study. The legacy system used is called The Master System (AMS) a product of American Financial Systems, Inc. (AFS). AFS, a small (60 employees) software firm, develops software for the Corporate Owned Life Insurance (COLI) market. AFS originally developed AMS in Microsoft DOS BASIC to integrate life insurance and executive benefits using mathematical and financial modeling. AFS evolved AMS from its original DOS version to the more modern Microsoft Windows® operating system. Currently, AMS uses Microsoft Visual Basic 6.0 ®. Appendix E contains an overview of the AMS architecture. There are about 500,000 lines of code in AMS. AMS is divided into the three engines described in Appendix E. The evolution methodology was applied to the Input Engine with a team of one project manager, one tester, and one developer.

There are several benefits in using the above-mentioned legacy system for this case study. First, there is historical data available on the system's maintenance costs, in terms of upgrades and evolution. This data will be used to validate results from the applied methodology. Second, although the tools and methodology used are language-independent, our use of a VB case study means that other legacy systems in VB can immediately benefit from our results. According to Microsoft, there are about 4 million Visual Basic developers worldwide as of December 2001 [69]. Third, AFS has a mature software process where the project manager, tester, and developer work together to manage a software release. Table 2.2 summarizes the characteristics of the legacy code to which the methodology was applied.

Characteristics	Description
1. Age of target legacy system	14 years
2. Lines of code	500,700
3. Lines of code implementing feature being evolved	12,000
4. Size of the project team	One developer, one tester, and one project manager
5. Size of production team	30
6. Programming language	Microsoft Visual Basic™ 6.0
7. Compile time	1 hour
8. Runtime environment	Windows desktop OS (9.x to 2000)
9. Version control software	Microsoft Source Safe™ 6.0
10. Product lines	AMS, AMS-WEB, DTS, DTS-WEB, SDEV
11. Software Process	Mature, with ability to provide past maintenance cost and track current costs per release per individual
12. Past evolution record of target legacy system	System was kept up with compiler and OS upgrades. It is a desktop system used by over 5000 end-users and many features are added per year
13. Industry	Financial Services
14. Regression Test Time	3 days

Table 2.2: Legacy System Characteristics where the Methodology is Applied.

2.2.2 Role of the Internet

There is an increasing need to evolve and refactor legacy systems to new hardware and software development paradigms [115][104][37] such as the Internet because the Internet makes it possible for an organization to attract potential customers from the global marketplace by breaking down communication barriers. Agarwal and Mishra identified that web-enabling a legacy system requires a combination of approaches, such as redevelopment, wrapping, evolution, reuse, component-off-the-shelf (COTS) integration, and configuration management [104]. One challenge for evolving a legacy system to become web-enabled is the need to provide continuous availability of the system (and thus its revenue-generating income) during the transition. For example, it is quite natural to continue to support a desktop version of application while its web counterpart is first launched. Thus, it is inevitable that organizations will maintain two product lines for an indefinite time-period because the time-period to migrate may be longer than expected. It is difficult for these organizations to justify the cost for maintaining more than one platform to the end-users. Thus, these organizations need an evolution methodology to help them reduce the maintenance cost during the migration period. One way to reduce this cost is to share components between the two versions of the application during and post migration (as shown in

Figure 2.2). Reusing components between the two platforms raises several interesting issues such as configuration management and deployment as shown in

Figure 2.2. A detailed analysis of which components can be reused along with platform issues is provided in Chapter 6.

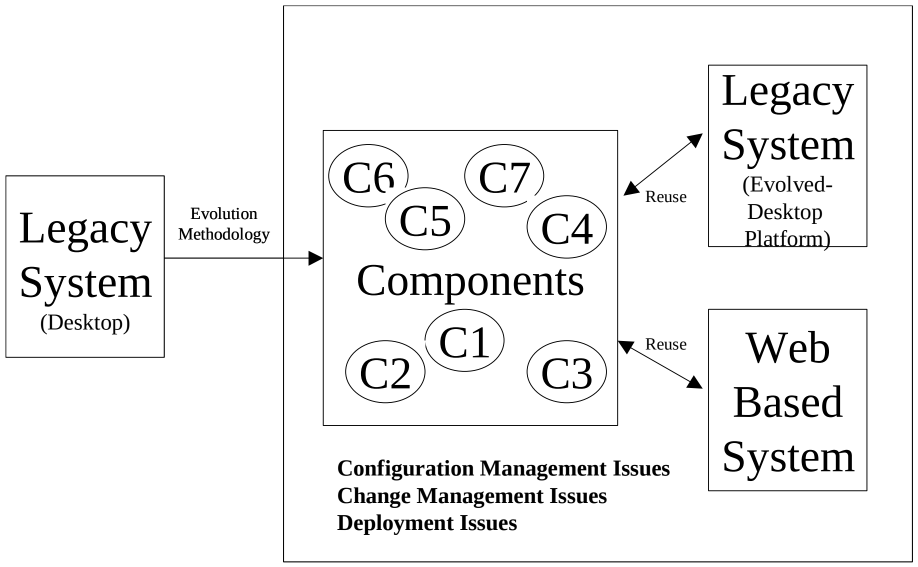


Figure 2.2: Component Sharing by Two Product Lines.

2.2.3 Regression Testing Process

Our methodology depends on regression test cases and a robust regression testing process. Regression testing has been extensively studied by researchers [46][127][48][41][40] from a theoretical point of view. Their theories show how to minimize and prioritize test cases to reduce the time of execution of the regression test suite. Researchers have also analyzed the source code and identified the test cases that should be part of the regression test suite in order to maximize the code coverage. Although test case minimization, prioritization and automatic-creation is important, we found that organizations primarily use regression test cases to measure the stability of a legacy system from one version to another. To the best of our knowledge there has been no investigation on applying regression testing in industrial environments specifically for evolutionary reasons. Some of the important issues mentioned in [11]:

- Regression testing is used extensively. In fact, other than functional testing (or black-box testing) and software inspection, regression testing is the most commonly used software testing technique.
- The frequent and extensive use of regression testing has led companies to develop in-house regression testing tools to automate the process.
- In some companies, all existing test cases are rerun in regression testing. In other words, minimizing test cases for rerun has not been a critical issue for these companies.
- New test cases are added to the regression test suite to reflect defects previously identified by end-users.
- End-users often apply their own regression test cases when they receive a new version of a system to ensure proper functioning of their favored features. This is a particularly important observation for our research because we analyze the regression test cases to help identify the implementation of system features in the code.

Different companies use different processes to develop and maintain software, such as the waterfall model and the spiral model. However, many companies [14] along with AFS follow certain steps in regression testing as shown in Table 2.3:

Normal	Description
1. Modification request	An issue is written when a defect is found in the system
2. Source code changes	The defect fix may require source code change
3. Test case selection	The fix is confirmed by use of test case(s)
4. Execution of test cases	Test cases are included in the regression run and are executed
5. Examine test results	Results from regression are analyzed
6. Failure identification	Test cases which fail are identified
7. Fault migration	A further fix is required by the developer

Table 2.3: Regression Testing Process.

- Modification request: When a defect is found by an end-user and verified by the organization, a modification request is created.
- Source code changes: The required software artifacts (requirement documentation, design specifications and source code) are changed to reflect the modification request.
- Test case selection: The modifications made to the software must be tested using test cases. Testers and Engineers work together to develop test cases to exercise the modified functionality. The selection of test cases is often a manual, analytical, iterative, and a time-consuming process. The goal in this step is to obtain the right test cases rather than minimizing the number of test cases.
- Execution of test cases: Test cases are scheduled to run. Since the number of test-case are often large, this step is usually automated in the form of batch mode operation that involves little or no user interaction.

- Examine the test results: The results from one version of the software are compared to a previous version to ensure that the changes included in a given version do not disturb the stability of the software in previous versions. In the case of new functionality, the results must be manually verified.
- Failure identification and fault mitigation: If the source code is suspected to be faulty, the developer examines and fixes the source code that caused the test case(s) to fail. In case of failure of the new functionality, this step often demands that the requirement and design specifications be reviewed and possibly modified.

Table 2.4 summarizes the requirements of a regression testing process. Figure 2.3 shows how the system is run in a batch mode. Essentially, the steps are as following:

- The list of test cases to be executed are stored in a batch script file
- The system is invoked via its batch interface
- The test cases are executed and system reads test cases information from a predefined data source
- The system processes some initialization functions such as connection to the database, paint screens and set global variables

- The test cases execute the features that they represent and store the output to an external ASCII text file. This output can be compared to the prior version's execution to measure stability
- The system shuts down and executes cleanup tasks such as resetting the global variables and closing the database connections

Properties	Description
Availability of regression test suite	Regression test suite is used to measure stability of the system from one version to another. We assume that regression test suites are available.
Ability to identify feature-specific test cases	Usually testers or project managers have this information.
Command-line execution of legacy system	Executing the legacy system with command-line options allows the system to execute the test cases that belong to the regression test suite the system in a batch mode thereby saving time.
Ability to output results in an ASCII text file	Using command-line option to execute the output of the system should be stored in an ASCII text file.
Ability to compare text files from one version to another	The output from one version can be compared to another to identify any unexpected changes.

Table 2.4: Properties of Regression Testing Procedures.

Running the system in a batch mode is a sign of a mature system because the system must be programmed to accept test case data in a command line interface. However, this interface is an efficient way to test the system and we recommend that a legacy system have such functionality. Many times the test cases may not be readily available in database or a file but are either manually inputted or an automated system may not be in place. Our methodology can still be applied in the absence of the batch interface or an automated testing system in place by manually entering the input data and executing the system one test case at a time.

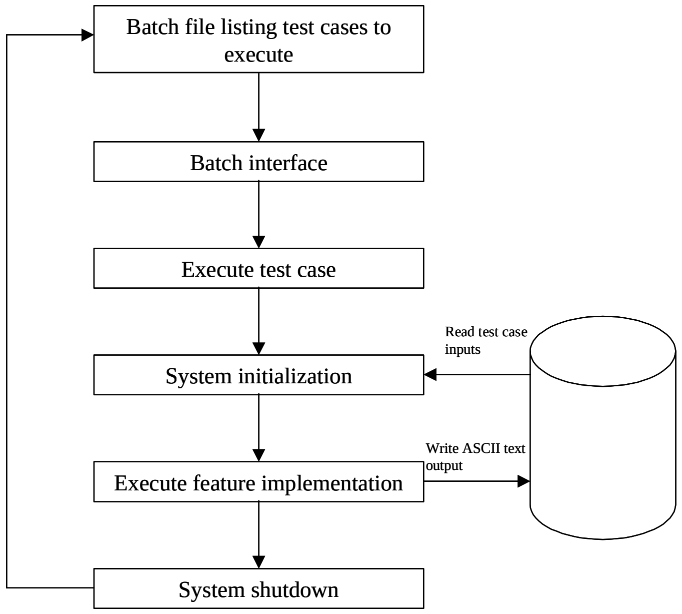


Figure 2.3: Running the System in Batch Mode.

As we can see, legacy systems undergo a fair amount of testing and they have a rich set of regression test suites and testing in place. While the nature and details of regression testing will vary from system to system, and organization to organization, our methodology is independent of the actual details. The sole dependence is on the availability of test suites. If meta-data about the test suites is available, our methodology will be more powerful and precise. Appendix C describes the AMS regression-testing tool and its batch capability in detail.

2.2.4 Source Code Profilers and Coverage Tools

Code coverage analysis is the process of finding areas of a program not exercised by a set of test cases, creating additional test cases to increase coverage, and determining a quantitative measure of code coverage. A code coverage analyzer automates this process. A profiler application provides information about which lines actually run, how many times a line is run, duration, and variable reference-use analysis. Coverage and profiling enable a developer to identify problem areas in an application, especially dead code and performance bottlenecks. Table 2.5 provides a list of features in currently available commercial profiler and coverage tools; most contemporary tools are able to provide the data needed for our methodology. The methodology can easily be tailored to use any code profiling tool that provides the required information in Table 2.5.

Properties	Description
Coverage percentage	Several commercial products provide this data when the tool is ran with input data. It represents the percentage of lines covered within a given function.
Line numbers covered	Line numbers are important because our methodology uses them in identifying which lines are covered.
Call sequence of function	A stack dump that the tool provides
Number of lines per function	Total number of lines per function
Reference-use of variables	This list allow us to see the location of local and global variables being changed

Table 2.5: Properties of Code Coverage and Profiler Tools.

2.2.5 Refactoring Techniques

Refactoring is a disciplined process of changing a software system in such a way as to improve the internal structure of the code while leaving the external behavior unmodified. Fowler states, refactoring is essentially "improving the design of the code after it has been written." [86]. Fowler recommends refactoring at three points in the coding process: when adding functionality; when fixing a bug; and when evolving. While refactoring techniques vary from project to project and depend on programming languages, we strongly agree with Fowler's last recommendation. We found that refactoring code into components in our methodology allows the developers to provide meaning to the data structures and functions that have lost their meaning over time. We provide a list of common refactoring techniques in Appendix F that we used in our case study when applying the methodology. Whichever refactoring techniques are applicable, the end result using our methodology is the same, namely, the creation of a fine-grained component.

2.3 Summary

Our methodology can be summarized in one sentence; ***by exercising each feature with its associated test cases using code profilers and similar tools, feature implementations can be located and refactored to create reusable fine-grained components.*** In this chapter, we outlined the ten-step methodology that will be discussed in detail in Chapter 4. We discussed factors that affect our methodology and hopefully convinced readers of the general applicability of the methodology to numerous legacy systems. A summary of each step and their input and output is outlined in Table 2.6. In Chapter 3, we discuss other research initiatives that are closely related to our work to show the novelty of our approach.

Sequential methodology steps	**Input**		**Output**	
	Requirement	**Responsibility**	**Deliverables**	**Responsibility**
1. Evolution reasons and problem report	Detailed textual description	End-user, tester and project manager	List of product lines where the feature is used	Tester, project manager and developer
2. Identify feature(s) with problems	Textual description and list of product lines.	Tester and project manager	Map problem to feature(s).	Tester and project manager
3. Map feature(s) and test case(s)	Map problems to feature(s)	Tester and end-user	Select test cases from regression suite for the feature. Analyze and verify test cases. Regression test suite cluster compared with selected test cases cluster. Add missing test cases to selected list	Tester
4. Map features to functions	Selected test cases, code coverage tool and legacy system	Developer	Feature-function cluster, feature-variable cluster and CORE functions report	Developer
5. Identify feature implementation and CORE using Feature Model	Feature-function, feature-variable cluster and CORE functions report	Developers, tester, end-users and project manager	Impact analysis, evolvable feature, and refactoring decision	Developer
6. Refactor and create components using the fine-grained component model	Feature implementation , legacy system and CORE	Developer	Fine-grained components and CORE library	Developer
7. Plug the component in the legacy system	Fine-grained components and CORE library	Developer	Integrated legacy system	Developer
8. Verify results by running regression	Integrated legacy system	Developer	Regression testing results. Report problem fix.	Tester
9. Reuse fine-grained components in other product lines	Fine-grained components, usage guide and other product lines	Developers	Run regression after integration on all integrated product lines	Tester
10. Measure results in terms of cost and report fix to end-users	Past and current feature maintenance cost, cost to apply the methodology	Project manager	Budget analysis report showing the result of applying the methodology	Project manager

Table 2.6: Methodology Showing Input and Output of Each Step.

3 Related Work

3.1 Introduction

To date, we have found no work that incorporates features, CBSE, and regression test cases to evolve a legacy system. The novel combination of Feature Engineering and CBSE presented by this dissertation greatly benefits software evolution by bridging the complexity gap between the problem domain and the solution domain.

There are many areas of software engineering that are related to our research: ***Software Evolution, Architectural Reconstruction, Feature Engineering, Product Lines, Requirements Analysis, CBSE, Program Understanding, Regression Testing, Separation of Concerns*** and ***AOP***. We feel that work in the area of ***locating system features*** has most directly affected our research. In Section 3.2, we analyze and document the shortcomings of existing techniques in the literature for locating program features. After carefully reviewing all the existing techniques for locating program features, we found that there was a need for ***feature model*** and ***fine-grained component model*** for software evolution. These models will be discussed in Chapter 4. Sections 3.3 through 3.8 present a broader perspective on the work from areas related to our research activities.

3.2 Locating Program Features

There are known techniques [18][6][92][99] to locate program features using execution slices, however they are predominantly used for system debugging rather than evolution. As far as identifying program features, there are four researchers whose work is directly related to ours. Wilde and Scully (WS) [99] pioneered the use of execution trace to locate the implementation of features; Wong *et. al.* (W) [125], Reps *et. al.* (R) [121] and Deprez and Lakhotia (DL) [71] developed techniques that operate on execution traces to collect information about features. In this section, we compare and contrast their work relative to ours. We will compare the motivation, models, techniques and applicability of each of their work. Table 3.1 summarizes our findings.

3.2.1 Motivation

(WS) developed their technique to locate program features for the maintainers of the private branch telephone exchange (PBX) switch. This switch had several hundred features that users could use such as ‘speed calling’ or ‘call waiting’. Maintainers of the PBX often needed to locate the code that implemented one of these features. Their motivation is best described in one phrase, *software reconnaissance*, which implies “*preliminary survey of enemy terrain*” where the software program is considered as an enemy whose secrets must be extracted. The general idea of *software reconnaissance* is to aid developers in program understanding and to debug and enhance program features. *Software* *reconnaissance* can also be used to recover some requirements traceability information from old code.

(W) developed a technique to identify the feature implementation for program understanding, debugging, and testing. Their study reported how to apply an execution slice-based technique to a reliability and performance evaluator to identify the code that is unique to a feature, or is common to a group of features. Supported by χSuds tools [1], the program features in the source code are tracked down to files, functions and lines of code. Their study suggests that the technique provides software developers and maintainers with a good starting point for quick program understanding during debugging and testing.

(R) described techniques to help with testing and debugging, using information obtained from path profiling. They instrumented a program to collect the path information for an execution run. With such an instrumented program, each execution of the program generates a path spectrum. Their technique compares path spectra from different runs of the program to identify paths in the program along which control diverges in the two runs. By choosing input datasets to hold all factors constant but one, the divergence can be attributed to this factor. The point of divergence itself may not be the cause of the underlying problem, but it provides a starting place for a developer to begin his/her exploration. (R) is also motivated by program understanding and debugging, specifically in analyzing year 2000 (Y2K) problems.

Lastly, (DL) were successful in mapping program features to code using input sets. They extended (WS)'s idea of *software reconnaissance* after (WS) reported that, "*The most time consuming step, and the one most difficult to automate, is the preparation and running of test cases*". (DL) present a formalism to automate mapping from program features to code by partitioning input-sets into *invoking and non-invoking sets*. Although their motivation is largely theoretical, (DL) believe that applying their theory would be used for program understanding.

Our methodology complements and extends the work of the four researchers mentioned above. We are motivated by the three factors described in Section 1.3. We know the researchers have had different motivations for locating features implemented in the source code. Due to these different motivations and the varying techniques to locate the features, they defined the research elements differently.

3.2.2 Models

The research elements that concern us most are features, functionality, test case and components.

(WS) interchangeably used feature, functionalities and functional requirements for the end-user features. They define a feature to be anything that is testable. (W) defined a feature as an abstract description of a functionality given in a specification. (R) has no formal definition of a feature. (DL) definitions are formally based on the grammar of a program; they parse the input sets of a program and define features based on language syntax. While both (W) and (WS) consider the end-user's perspective, neither provides any insight on evolution or considers the solution domain's point of view (POV). (DL) is not practical because it requires formal automata to represent features, which most maintainers and programmers performing evolution do not have available; (DL) also avoids discussing the solution domain's POV. Our approach (see Chapter 4) defines features in a comprehensive manner by considering both the problem and the solution domain.

Our definition for feature is rooted in the problem domain by focusing on the requirements but shows how a feature can be used in software evolution. For example, a system might support a feature that performs complex calculations in batch mode without user interaction. To an end-user, POV this feature is a time saver because input can be stored in a file or a database to be used at a later time. At the same time, testers might employ this feature to enable regression testing between two versions of the system during maintenance; developers might design a specific set of modules to process user input without user interaction to analyze code coverage.

(WS) reported that considering a subroutine (i.e., function) is not enough to address the feature interaction problem. They found that a finer level of granularity is needed. They solved this issue by using arc of program flow graphs as components. According to their work, it is easy to produce the necessary execution data using any test coverage tool that produces branch of decision coverage. It is important to note that global data is not mentioned in (WS), but legacy systems have many global variables and usually the global data is shared between two or more features. Our feature model addresses the issue of shared global data.

(W) was primarily interested in calculating how close one feature is to the program component rather than another feature. (W) presented three metrics to calculate relationships between feature and component namely; *disparity* between a feature and a component, the *concentration* of a feature in a program component and finally *dedication* of a program component to a feature. (W) defined a component to be the source file where the feature was implemented. Because these metrics are only an approximation, as they do not address the feature interaction problem directly, they simply use set theory and statistics principles to calculate the intersecting set between the interacting features.

(R) follows the approach of (WS) using program graphs to solve the feature interaction problem. Using path coverage from different input data sets they were able to identify code associated with a given feature. They numbered the paths with a unique source node and sink node. Every cycle contained one back edge, which can be identified using depth-first search. Since their motivation was only to solve the year 2000 problem not much attention was paid to the shared global data or shared code. Additionally, their approach has not been applied in identifying features other than the year 2000 problem. Their approach leaves the burden of relating features and program to the programmer.

(DL) identified the feature interaction problem and labeled it as an *imprecision due to feature coupling*. Their work identified the code that is executed by invoking input sets vs. excluding sets; by calculating the difference between the two sets they determined the execution traces. They attempted to solve the feature interaction problem by selecting large input sets. They suggested that the best way to minimize imprecision is to execute a program with large input sets that invoke different combinations of features. Using such an approach, one would hope that the complete implementation of each feature has been covered by some of the input sets. While theoretically it is possible to minimize the imprecision, this solution will be problematic in cases when two or more test cases invoke the same set of features.

Our solution (see Section 4.1.4) to feature interaction is rigorous and intuitive. We believe that it overcomes the shortcomings of the four researchers. Unlike their approaches, our approach addresses the issue of global data discussed earlier. In addition, our work is independent of the type of input sets selected. We also explore the feature-feature relationships in far more detail than the other researchers. Our feature model identifies functions that implement more than one feature but instrumentation does not show any differences between the execution paths.

3.2.3 Technique

Generally, all four researchers developed techniques to identify program features using the following generic steps:

1. Identify or build input sets that *invoke* the feature.
2. Identify or build input sets that do not *invoke* the feature.
3. Execute the instrumented program with each input sets to create its execution trace.
4. Classify each execution trace in the *invoking* category if its input sets invoke the feature or in the *non-invoking* category if it does not.
5. Apply a method that operates on execution traces to map the feature to the code.

However, when we look beyond these generic steps to the specifics of the researchers' techniques, we see differences. Although there are some similarities between (WS) and our approach, such as the concept of CCOMPS and IICOMPS, these similarities remain at different level of granularities. Our model addresses these sets at a much finer level of detail by considering local and global variables (and functions) that may be shared between two or more features. (R) does not assign code to a feature but rather identifies the points of divergence between several execution traces and the programmer is responsible for determining the relationship between feature implementation and a feature from this divergent point. (W) identifies program features in C and C++ programs by running a small, carefully selected set of tests. While (W) provides metrics to measure a feature's relationship to the code, there is no explicit discussion of relationships among features. Furthermore, (W) metrics show severe shortcomings when two or more features may have the same slice for different test cases.

3.2.4 Applicability

(WS)'s work has been realized in a tool called TraceGraph. Two case studies on large-scale systems, written in 'C', have been reported in [78]. TraceGraph provides a visual display of the program's trace that allows changes in execution to be distinguished. It is important to note the limitations that TraceGraph shares with Software Reconnaissance. First, it may only be used to locate features that the program's user can control. Most programs contain a significant amount of common code that is *always* executed on every non-trivial test. While a maintainer may need to locate some specific part of this common code, such as the symbol table handler in a compiler, neither TraceGraph nor Software Reconnaissance can help. Second, as for any dynamic analysis technique, the results depend to some extent on the test cases used. If a feature is sometimes handled one way and sometimes another, neither technique will find the full extent of the feature unless the maintainer supplies input that cover both cases. On the other hand, both techniques will identify unwanted code components if the "with" tests accidentally include functionality that is absent in the "without" cases. Finally, both techniques only provide *starting points* for the exploration of code. The maintainer still needs to do the hard work of studying each identified component and understanding how it fits into the rest of the target system.

(W)'s work has been realized in a tool suite called χSuds [1]. The tools were developed in the *Telcordia Applied Research* laboratories to analyze the dynamic behavior of software and to allow the user to visualize program data through graphical user interface. The tasks of determining code coverage, finding a minimized test set, debugging, identifying what part of the software implements a specific feature, profiling program performance, and finding static program relationships are made available to the developer by the various tools. The tool suite focuses on the testing and maintenance of C and C++ systems. *x*Vue is the software maintenance tool in χSuds tool suite. To determine where a feature is implemented in a program, one would run a small, carefully selected, set of tests; some that involve the feature and others that do not. Such tests are classified into three categories: *invoking tests, excluding tests and don't_know tests*. *x*Vue analyzes traces of program execution to find program components that were executed in the invoking tests but not in the excluding tests. The χSuds tool suite is commercially available and has been tested on large C and C++ programs. However, it faces similar shortcomings as TraceGraph.

(R)'s work has been realized in a tool called DynaDiff [121]. DynaDiff is not a commercial tool and has been built at the University of Wisconsin. DynaDiff works on programs that run under Solaris on Sun SPARC stations. It is a software visualization tool like TraceGraph and *x*Vue. DynaDiff itself is language independent as long as the compiler (of the target program being analyzed) can create symbolic debug information. DynaDiff has been tested on small programs (as prototype case studies.)

(DL)'s work is theoretical in nature and there are no tools available that realize it as of yet. Their technique itself is an independent programming language but developers must have advanced knowledge of BNF grammar to develop a toolset to support their technique.

Although we are working on developing tools, we have applied our methodology in an industrial strength application, namely AMS (see section 2.2.1). We focused on evolving the *Input Processing* (12 KLOC) functionality of AMS. *Input Processing* validates and prepares data from user inputs so AMS can perform complex calculations to generate various reports. After applying our methodology, AFS is using evolved components in two of their product lines namely, AMS and the web version of AMS. Our methodology works with any programming language as long as there are code profilers (or similar) tools available for that language.

3.2.5 Reasons for a new feature model

The approaches of these four researchers have several shortcomings. In particular, there are six issues:

How to select input cases: All four rely on the developers to generate the input cases and then attempts to generate execution traces for *all* input sets. In industrial strength applications, the size of the input set makes such data collection impossible. Since regression test cases reflect the end-user feature, they are already focused so it is not necessary to collect execution traces on all inputs or to divide the input sets into *invoking* or *non-invoking* category. Thus, we suggest that regression test cases are the best choice for the input cases.

How to capture relationships between features: We present several feature relationships (among features) that can exist within a function that can implement more than one feature. These relationships are important because; they not only can be used for traceability purposes but can also lay groundwork for future evolution. Our heuristics allow developers to quickly determine whether a feature is evolvable or not.

How to classify functions called from a feature implementation: We classify whether these functions are Stateless (SS), Stateful (SSF), or Dependant (DF) in nature. This is more than just stack dump or calling sequence because by knowing the types of the functions, one could implement the component’s interfaces and its private/public methods.

How to classify *core* software*:* If all related test cases execute SS functions then it is likely that those functions belong to the core. WS makes no distinction between the states. We decided to treat the core as a shared reusable library only when the functions are SS while the four researchers focused on calculating the set differences and manipulating anomalies (because core gets executed with every execution).

How to manage shared variables: Since large industrial legacy applications are notorious for having global variables, it is important that the technique to identify feature addresses what to do when two or more features share a single global variable. Our model also integrates the local variables that are shared among features.

How to discover feature interactions when code coverage has no apparent differences: While all four researchers identify the feature implementation, it is not clear what happens when a single function implements more than one feature and execution traces do not reveal any differences (i.e., the coverage is exactly the same for all features in a function). Our approach can help developers understand how a feature is related to another feature in such a circumstance.

Although most of these works derive results from execution traces left by the execution of a program with input sets, none use the methodology we propose. Based upon our analysis of the locating program features techniques, there was a distinct need for a more detailed model and definition for features. While all related models are more-or-less equivalent in expressive power, our work was motivated by the need to evolve legacy systems which made it infeasible to simply use many of these existing models.

Approaches **Factors**	**Wilde and Scully**	**Wong et al.**	**Reps at al.**	**Deprez and Lakhotia**	**Our Methodology**
Motivation	Program Understanding	Program Understanding and Debugging	Year 2000 Solution	Program Understanding and Testing	Evolution, Bridging the Complexity Gap and Software Reuse
Key Elements and their Definitions	Features are defined to be entities that must be testable. Subroutines are defined to be lines of code as components as opposed to FI.	Feature is same as functionality represented in specification. Source files, Subroutines and lines of code as components as opposed to FI.	Feature and Component definitions are left to the developer. No definition presented.	Feature is defined to be the language used in feature syntax. The program components are defined to be statements, and procedure as opposed to FI. Execution Traces are defined.	Features, FI and components are defined explicitly with end-use perspectives in mind. Core, Execution Traces are also defined.
Technique	Input sets developed by maintainers are used to identify program features using instrumentation and tracing tools	Difference between execution traces (via input sets) that invoke a feature and do not is calculated.	Input sets are executed to generate program graph and path spectrum is analyzed. Input sets are varied incrementally.	Features are mapped to input cases. Input set is classified into invoking and excluding sets. Finally execution traces of all input sets are collected so set differences can be calculated.	Regression tests used by testers and end-users that represent features are run in conjunction with code profiler to identify feature implementation for evolution purpose.
Formalism	Set theory based	Statistical and Set Theory based.	Graph theory based	BNF grammar and Formal Automata based	Relational Model and First Order Logic based
Applicability	Used in two large-scale C programs for research purposes but only as prototype. Commercial application yet to be seen. Appears to be scalable.	Used in a medium sized C program. Appears to be scalable. Three metrics (approx. only) are developed to calculate disparity, dedication and concentration between features and program.	Used in solving Year 2000 problem. Cannot be used to identify specific end-users features as each graph requires several inputs and usually results in huge permutations.	Theoretic in nature. No real practical application seen. Since it depends on large input data it takes a long time to identify relevant code. Generating BNF grammar is also cumbersome.	Used in evolution of features of large VB program into reusable components. In addition, feature and component model provide sound theoretical background.
Tools	TRACEVIEW	χSuds/χVue	DYNADIFF	None	Any Code Profiler
Feature Interaction Problem	Identify and suggested solving by selecting proper test cases	Identify and suggested solving by selecting proper test cases	Generally ignored but attempted to solve it implicitly by analyzing program graph via brute force	Identified and suggested solving by selecting large number of test cases	Identify and provide solid model to evolve features that share code and data into components
Mapping Test Cases and Features	Left to developers. End-user perspective is ignored.	Left to developers. End-user perspective is ignored.	Brute force. End-user perspective is ignored.	Must use large test suite and end-user perspective is ignored.	Regression test cases are used as they represent the end-user features
Generates Reusable Components	No	No	No	No	Yes
Programming Language Dependent	Yes	Yes	No but compiler should be able to generate symbolic debug info.	No	No
Addresses Global Data Issue in identifying features	No	No	No	No	Yes
Identifies and addresses Core	Yes, Partially	No	No	No	Yes

Table 3.1: Comparison of Closely Related Work.

3.3 Software Evolution

There are many approaches to the general problem of software evolution.

3.3.1 Incremental Evolution

While there is agreement on the importance of evolving legacy systems, it is hard to find a consensus on what the best model for evolution should be. Many models have been proposed over time, and these models differ not only in their approaches, but also in the way they define the deliverables of their methodologies. Importantly, the discourse among software engineering experts on the proper approach to legacy system evolution seems to have reached a stalemate in an old debate between two conflicting ideologies. These ideologies argue over the main problem of whether it is in the best interests of an organization if a large mission-critical legacy system is re-written or replaced all at once in its entirety, or is incrementally-evolved.

Over time, academics and industry experts such as Ransom et al.[76], Brodie et al. [83], and Tilley et al. [114] have introduced a number of names to describe these two approaches. Two of the most popular ones in use today to describe a model's persuasion in this dichotomy are the terms "Chicken Little" and "Cold Turkey." In this section, we discuss various models and approaches associated with incrementally evolving ("Chicken Little") the legacy systems.

The earliest references to these two terms that can be found date from 1991, and the pioneering DARWIN project from the University of California, Berkeley. The results of the project introduced "Chicken Little" as an approach to iteratively (also called "incrementally") evolve legacy systems [51]. As the DARWIN model was perfected, the incremental nature of the approach was stressed as the model was drawn out in eleven easy-to-remember "steps," each of which started with the word "Incrementally." This approach answers a clear "no" to the all-at-once question. With Chicken Little, "data gateways" are developed and introduced between the legacy system and the target system to maintain data consistency throughout a project. The key difference between DARWIN and our methodologies is that DARWIN addresses incremental evolution at architectural level rather than end-user feature level. Another important difference between DARWIN and our methodology is that DARWIN makes use of object-oriented techniques rather than CBSE.

The SEI at CMU developed a technique for developing an incremental code-migration strategy for large legacy Common Business-Oriented Language (COBOL) systems [19]. Specifically, the technical report published by SEI describes a case study that involves the modernization of a large Supply System (SS). The system consists of approximately two million lines of COBOL code operating in a mainframe environment. The SEI developed the System Analysis and Migration (SAM) tool to generate a code migration strategy based upon legacy system analysis data. SAM considers a set of factors that includes minimizing scaffolding code (code that is discarded before the completion of the project), balancing iterations, and grouping related functionality. SAM generates a call graph that allows developers to identify program elements with dependencies. These dependencies can be prioritized for evolution purposes. While the program elements can be viewed as end-users features, the SAM tool is dependent on COBOL. Our approach is programming language independent; we use existing source-code profiler to identify feature implementation. While a call graph can be used to understand program dependencies, our feature model provides feature-function matrix that shows an intuitive view of program dependencies. Finally, there is no mention of the issue of global data in SAM.

The Incremental Software Evolution of Real-Time System (INSERT) project was started by DARPA, SEI and NASA in 1992 [16]. The goal of INSERT is to improve war fighting capabilities of F16 fighter jets by incrementally evolving software systems used in the F16's operating and other software system. While there are similarities between INSERT and our methodological goals, the two approaches are different because of following reasons. First, our approach does not account for real-time systems. Second, the primary objective of INSERT is to incrementally replace F16's software components with COTS. Our methodology suggests refactoring of problematic feature implementation. Third, our approach targets identification and refactoring of specific problem areas (end-users features) while INSERT provides guide to replacing entire sub-system.

Entity-Life Modeling (ELM) is a method of software engineering that has elements in common with both function-oriented and object-oriented methods [13]. As in object-oriented design methods, the first step is the identification of objects from the problem domain, the identification of object attributes and operations belonging to each object, and the design of class structures that encapsulate state information and export attributes to other objects as needed. ELM departs from object-oriented methods in its ability to manage the timing and ordering of events as in some function-oriented methods. Threads of execution are defined wherein *entities* exhibit sequential behavior by operating on objects, perhaps concurrently with other entities. The application of ELM to evolution involves the identification of entities and their threads of execution. Some dynamic slicing may be necessary to identify objects and their behaviors. The steps in the application of ELM to incremental evolution may be listed as follows:

1. Identification of entities
2. Identification of concurrent tasks
3. Creation of Buhr diagrams
4. Design of interface objects
5. Composition of state transition diagrams

The entities in ELM method can be analogous to features in our methodologies. While ELM is certainly an incremental evolution methodology, the main difference between our methodology and ELM is that our methodology has not been tested on object-oriented systems and ELM has not been tested on function-based system.

The observation that software systems undergo continuing changes was first put forward by Belady and Lehman [79]. They termed this dynamic behavior of software systems evolution and carried out empirical research on about 20 releases of the OS/360 operating

system. The investigation led to five "laws" of software evolution: Continuing Change; Increasing Complexity; The Fundamental Law of Program Evolution; Conservation of Organizational Stability; and Conservation of Familiarity. These laws have been systematically studied by several researchers such as. Lehman and his colleagues have begun new investigations into software evolution. The FEAST/1 project (1996-1998) aimed to construct black- and white-box models of software system evolution, with special attention to feedback phenomena. The results of studying several data series from their industrial collaborators support, or at least do not contradict, the laws of software evolution formulated in the 1970s. Moreover, three new laws have been identified: Continuing Growth, Declining Quality and Feedback System. The recently completed FEAST/2 project focused on control and exploitation of process behavior.

There is a direct correlation between Lehman et al. third law (The Fundamental Law of Program Evolution) and sixth law (Continuing Growth) with our notion of incremental evolution. Both the laws support the hypothesis that focusing on specific problem areas within the legacy system businesses can justify return on investment (ROI). While there is no direct relationship between the case studies, we agree with Lehman et al. Furthermore, researchers such as Basili et al. [122], Coleman et al. [28], Kremerer at al. [21] and Kafura [30] have all used call graph and similar techniques to identify program dependencies based on Lehman's third and sixth laws of evolution to propose incremental evolution methodologies.

3.3.2 Legacy System Evolution

Many software evolution techniques exist, [96][64][105] but they focus on solution domain and do not consider Feature Engineering as a software evolution driver. In one of the first dissertations on Feature Engineering, Turner [25] mentioned the possibility of using Feature Engineering for software evolution purposes in his work, but he concluded that evolution was outside the scope of his work. The techniques of software evolution and reengineering either focus on entire system rewrites or simply deal with reverse reengineering for comprehension purposes.

System evolution is a broad term that covers a continuum from adding a field in a database to completely re-implementing a system. These system evolution activities can be divided into three categories: rewrite, evolution, and replacement [98][96]. Repeated system maintenance supports business needs sufficiently for a time, but as the system becomes increasingly outdated, maintenance falls behind the business needs. An evolution effort is then required that represent a greater effort, both in time and functionality, than the maintenance activity. Finally, when the old system can no longer be evolved, it must be replaced or rewritten. Determining the category of evolutionary activity that is most appropriate at different points in the life cycle is a daunting challenge. Should a system continue to be maintained or should it be modernized? Should the system be replaced or rewritten? To make the correct decision, the legacy system must be fully assessed in order to analyze the implications of each action. Ransom et. al. describe an assessment technique for determining if a legacy system should be replaced, modernized, or maintained [76]. Current software evolution and reengineering techniques continue to work in the solution domain. The important problem of linking the problem domain and the solution domain for the purposes of evolution remains unsolved.

3.3.3 Architectural Reconstruction

Architectural reconstruction is the process where the "as-built" architecture of an implemented system is obtained from the existing legacy system. This is done through a detailed analysis of the system using tool support. The tools extract information about the system and aid in building and aggregating successive levels of abstraction. If the reconstruction is successful, the end result is an architectural representation of the system that aids in reasoning about the system. There have been several efforts in architecture analysis and reconstruction. The Software Engineering Institute (SEI) has developed *Dali* [106]. Other examples of architectural reconstruction efforts include Sneed's reengineering effort [49], the software renovation factories of Verhoef et al. [123], and the re-architecting tool suite by Krikhaar of Philips Research [108]. In almost all the software architecture reconstruction efforts, the process comprises the following five phases:

- View extraction phase obtains information from various sources.
- Database construction phase involves converting the extracted information into a relational database format.
- View fusion phase combines various views of the information stored in the database.
- The architecture reconstruction phase builds abstractions and representations and to generate an architectural representation.
- Finally, the Architecture Analysis phase analyzes the resulting architecture.

There appear to be several similarities and differences between Architectural Reconstruction and our work.

Our motivation is to incrementally evolve legacy system features with problems. Architectural reconstruction attempts to migrate the entire legacy system to a newer architecture. We rely on code profilers to get information regarding feature implementation. Likewise, code profilers can also be used to populate the database in the fusion phase. It appears that architectural reconstruction can be used to identify feature implementation, not just the architecture. However, there is no mention of evolving feature implementation into components for reuse.

By considering end-user's features we bridge the complexity gap between the problem and the solution domain. Architectural reconstruction works in the solution domain only by focusing on extracting an architecture from the legacy system.

The outcome of architectural reconstruction effort is different than ours. We are focused in creating reusable components as opposed to representing architectures.

3.4 Feature Engineering

3.4.1 Features

There is little reference to the word "feature" or to the practice of "Feature Engineering" in existing software engineering and other technical literature. For the most part, the use of the term feature has been used in regard to the research issue being addressed, such as features of a particular methodology or technique. One particular research effort, Feature Oriented Domain Analysis (FODA) explicitly uses the term [77]. However, FODA referred to a specific feature, not the concept of feature. There are a few typical examples along the same lines as FODA in the published work of Kamigaki *et. al.* and Larrondo-Petrie, *et. al.* [129][90]. The SEI FODA feature model ties business models together by structuring and relating feature sets [87]. The FODA framework explores how this structured information can be leveraged across the software development effort.

Griss extended the FODA methodology to create an explicit feature model of functionality to facilitate reuse-driven software engineering [87]. We agree with Griss that a feature model integrates the viewpoint of both the user and the developer; in this dissertation, we show the practical application of this integrated perspective.

Cusumano and Selby describe the strong orientation of software development toward the use of feature teams and feature-driven architectures at Microsoft Corporation [84]. While this orientation has more to do with project management than with product life-cycle artifacts and activities, there is a significant interest in features among many software development teams. Feature enhancements provide both a competitive tool and a healthy revenue stream from product upgrades. For requirements, a use-case based method is used to determine the feature set that should be added to a new product. Using focus group and automated testing these features are given scores. Features that score highly in the usage scenarios are most likely to be incorporated into the next product version. Microsoft's approach to features concentrates on specific features to be added to existing products. Feature Engineering, in contrast, is a general set of approaches geared toward understanding the concept of features and making use of the feature relationships in a disciplined fashion across the solution domain.

3.4.2 Feature Interaction

The feature interaction literature is primarily focused on telecommunications networks [116]. Telecommunications networks are massive, complex, distributed systems that incorporates a variety of hardware and software elements. In this domain, features represent capabilities that are incrementally added to a telephony network. The presence of multiple independent component providers makes the feature interaction problem even more difficult. Telecommunications networks provide many examples of features, such as call waiting, call forwarding, and voice mail; the primary focus is on understanding how features interact, rather than how the features will be evolved. Our feature model and fine-grained component model addresses evolution of interacting features.

3.4.3 Requirements Analysis

Features are problem space entities, and requirement engineering is the discipline that is focused on providing a concise, consistent, unambiguous, and complete definition of the problem domain. Years ago, researchers identified features as a natural organization of the problem space [4][101]. According to Turner et al. [26], Feature Engineering reemphasizes the need for requirement analysis efforts to identify the desired Feature set. While there are a few close synonyms for feature, such as goal and root requirement, surprisingly few approaches in the research literature concentrate on this organization of a system’s functionality. Several approaches in requirements engineering approach the Feature identification required by Feature Engineering. Hsia and Gupta [101] have worked on automated techniques for grouping requirement specifications. Their purpose is to support incremental delivery of system functionality. The cohesive structures that Hsia and Gupta search to identify are abstract data types (ADTs). It is clear that ADTs are a solution domain concept with limited relevance in the problem domain. In addition, their work requires using a development methodology based on ADTs. The goal of delivering ADT-based prototypes transcends analysis and forces a particular design choice. While [101] appears to reduce the complexity gap via ADTs we differ by reducing the same gap via regression test cases. Likewise, Karlsson and Ryan [73] seek to prioritize requirements using a cost-value evaluation of pairs of requirements. Since the number of requirement pairs grows as the square of the number of requirements, their approach is suited to high-level requirements identified in the problem domain. Their techniques can be used to trace artifacts in the solution domain. While there are similarities in the requirements analysis work regarding mapping the problem domain to solution domain, we differ mainly by using regression test cases as the starting point because most legacy systems do not have original requirements definition.

3.4.4 Function Points

Function point analysis is potentially applicable to Feature Engineering. The basic notion of this discipline is that the functionality of a software project can be objectively estimated independent of the implementation. Function point analysis considers five system characteristics: application inputs, application outputs, user inquiries, data files, and interfaces to other applications. Each application has a function point rating, which presumably can be determined objectively once the system specification is created. Capers Jones asserts that function point metrics have substantially replaced the older lines-of-code metrics for purposes of economic and productivity analysis [20]. Since the introduction of this metric, numerous refinements have been introduced, and in 1986, the International Function Point Users Group was formed to enhance the technique. Despite advances in function point analysis, subjective judgments remain a difficulty because of lack of evolutionary initiatives.

Five early goals were identified for the function point metric:

1. Relate to external features of the software
2. Deal with features important to the user base
3. Be applicable early in the life cycle
4. Relate to economic productivity
5. Be independent of source code or language

These goals are well aligned with, but considerably narrower than, the feature-engineering ideas identified in this dissertation. Since function point metrics are based on visible aspects of a software system, they fit naturally within the feature view of a software system. Function point analysis might be useful for estimating the development effort required to implement a particular feature. It might also be used to evaluate the complexity of various implementation alternatives during the feature design phase. By applying the metric to the incremental development required for adding features to a system, the cost and impact of each feature can potentially be estimated.

3.5 Component Based Software Engineering (CBSE)

Although Component Based Software Engineering (CBSE) provides viable techniques to develop modularized software systems, the components are often designed and implemented from scratch rather than re-engineering them from within a legacy system. In practice, CBSE is used as a design and construction tool, not an evolution tool [95][126][94][38]. In this section, we summarize many of the sub-discipline of CBSE as they relate to our dissertation.

3.5.1 Evolution

Recent approaches to evolution within CBSE, such as ArchStudio [102], focus on evolving systems that are already designed and constructed from well-defined components and connectors. The emerging discipline of Software Architecture as defined by Garlan and Shaw is concerned with a level of design that addresses structural issues of a software system, such as global control structure, synchronization and protocols of communication between components [29]. Software Architecture is thus able to address many issues in the development of large-scale distributed applications by using off-the-shelf components. In particular, it is a useful vehicle for managing *coarse-grained software evolution*, as observed by Medvidovic and Taylor[94]. However, Software Architecture does not provide an efficient solution for legacy system evolution.

Evolving a legacy system by wrapping it into a component is a common practice [115]. However, such wrapping results in coarse-grained components and does not address the issue of complexity gap. Our methodology identifies features that are a candidate for evolution and incrementally evolves them at much finer granularity.

3.5.2 Wrapping

While wrapping is a perfectly viable solution to evolve a legacy system onto a newer platform, our motivation is rooted in addressing problems associated with end-user features.

3.5.3 COTS

COTS can certainly provide functionality pertaining to the feature we are interested in, however we see following major differences in using COTS compared to our fine-grained components:

Researchers have found that COTS selection is a lengthy and arduous process [27][112]. The first step of the process is to determine the best COTS components candidate. The next step in the process is to determine if these components can be integrated, either directly, or through wrappers or other "glue" code. Determining if components can be integrated is also a complex process, as vendor claims are not always believable. If these components cannot be easily integrated, it is necessary to consider alternate products that may not be best choice but are compatible with other technologies. To make matters worse, the environment is constantly changing with new components and emerging product versions; existing products going away or being refocused, and evolving vendor relationships. This is hardly the case with our fine-grained components as these fine-grained components are evolved from within the legacy code they integrate well and provide the specific functionality that is needed.

COTS components are black boxes whose source code is not available for modification. Since fine-grained components are developed using an existing legacy system, its source code is readily available.

3.5.4 Reuse

One of the main ideas behind CBSE is to promote software reuse either within the product line or across multiple product lines [95][23][75]. However, the claim for such reuse has been challenged because CBSE has not been able to deliver its promise. Furthermore, the dynamic nature of requirements and software process pose a big hurdle for CBSE as far as reusability of components is concerned [126][44][66]. Since our methodology gathers the requirements and the specifications from an existing legacy system, we can simply refactor feature implementations into fine-grained components. Avoiding the complex process of gathering requirements to create components from scratch allows us to take the best of CBSE, namely component model and component specification, without having to consider time-consuming CBSE activities such as buy vs. build analysis, selecting a component model, or a component technology.

3.5.5 Features

CBSE offers promising techniques to solve the problem of component construction [2], but CBSE has not yet been connected to the features that are present in a system; creating this connection explicitly is one of the contributions of this dissertation. The functionality provided by CBSE solutions must be mapped to the Feature available to the end-user.

To the best of our knowledge, features and components have not been studied together in light of legacy system's evolution. Two areas that appear to bring the aspect of features to components are feature-oriented programming (FOP) and Feature-oriented classification of components (FOCS).

FOP is used for developing new systems [24] and has not been used in evolving existing legacy systems. However, FOP can be used to create feature-oriented components, which can possibly be used with our methodology. Integration of components created using FOP is outside the scope of our work.

FOCS is a component classification scheme using graphs [67]. Components are described by sets of features, called descriptors. Each feature represents a property or attribute of the component. To support the understanding and construction of descriptors, features are organized in a classification scheme. Storage and retrieval of components is done by means of these features sets. A thesaurus assists in the understanding of features. Searching is done with the help of descriptors. Users construct a descriptor (using an editor) containing the features the searched component should provide. This descriptor is interpreted as a query to the database of classified components. While there appears to be no direct relation to our work, FOCS can be used to store fine-grained components created by our methodology.

3.5.6 Fine-Grained Components

Granularity is the word that describes how much functionality is found in a component, or a set of components that work together. In the literature, two types of components exist: fine grained and coarse grained [38]. In [38], James Carey and Brent Carlson describe two types of components based on their many years with the San-Francisco project. The authors differentiate fine-grained components from course-grained components. Carey and Carlson make a persuasive case for the use of fine-grained components; they argue that such components are required for business domains where well-defined dependencies can be carefully managed. We strongly agree with Carey and Carlson, as our fine-grained component model encapsulates the feature we are interested in evolving for reusability across product lines within the same organization.

There are also similarities between our fine-grained components and ability to modify its code. According to [45], there are three possibilities for modifying a component:

White box where access to source code allows a component to be significantly rewritten to operate with other components. ***Gray box*** where source code of a component is not modified but the component provides its own extension language or Application Programming Interface (API). ***Black box*** where only a binary executable form of the component is available and there is no extension language or API. Our fine-grained component model is intended to be used across the product lines with an organization. Since the component may contain feature-based trade secrets, organizations may decide to not market it but to use the component exclusively. Since the code is available, technically all three approaches to the modification can be applied. However, we suggest a black box approach be used since fine-grained components are lightweight and provide feature specific functionality whose code need not be changed.

3.5.7 Product Line

The general idea of a software product line is that the new product is formed by taking components from the base of existing legacy code using variation mechanisms such as parameterization or inheritance. Thus, building a new product (system) becomes more a matter of assembly or generation than creation; integration rather than programming. This form of reuse among product lines, have been studied by various researchers such as [87][68][70][74][91][81]. Among all the product line initiatives, the most related work is that of the two methods developed by the Software Engineering Institute (SEI). These methods are supposed to extract existing assets from the core of an existing product line. The Mining Architectures for Product Lines (MAP) method addresses assets at the architecture level, while the Options Analysis for Reengineering (OAR) method addresses assets at the component level. While there are similarities in our motivations and those of MAP and OAR methods in reusing components, we differ in the following ways.

We are motivated in reducing the complexity gap by considering problem and solution domain, while MAP and OAR work in the solution domain only. Both MAP and OAR are not focused on incremental evolution while that is our intent. One of our goals is to reduce the maintenance cost of the feature to be evolved. It is not clear from the literature that the MAP and OAR consider this factor. MAP and OAR are more focused in the new product lines that use the extracted components from the legacy system while one of our goals is to plug the component in the original legacy system as well.

3.5.8 Previous experience with components and evolution

We mention two of our previously related works in this subsection, both of which are experience reports and provided preliminary motivation to work with components and evolution.

First, one of our previous works, carried out as part of the case study on AMS Output Engine, has contributed to research efforts in Software Architecture and CBSE [7]. This work sought to evolve earlier legacy systems so that recent ideas on architectural evolution could be applied. The experience report describes a simple technique: abstracting the communication between two components into a connecting-component. Using this abstraction, a stand-alone executable was easily converted into an ActiveX Component (DLL). This research demonstrated that architectural analysis helps to achieve business objectives. The methodology described in [7] forms the basis for our motivation in this dissertation because it is: 1) is incremental; 2) improves the architectural integrity of the legacy system by replacing implicit communication between system components with explicit, documented connecting-components; and 3) results in a better-documented architecture.

Second, to further stress the importance of CBSE encapsulation and reuse techniques we briefly describe the concept of component integration and extension [8], which was applied in the Input Engine of AMS, In [8], we have shown that new features can be integrated and extended into the original component by using CBSE techniques. Component integration and extension techniques improve code reusability among product lines and decrease maintenance costs for legacy code [8]. Component integration and extension techniques will encapsulate functions that implement features in context.

Encapsulating features into components will improve code reusability and will thus reduce the maintenance costs for legacy systems.

3.6 Program Understanding

Program slicing is another area that has potential for Feature Engineering and our component refactoring is inspired by this research. The notion of program slicing began with Weiser [93]. Since then, several researchers have modified and expanded the concept of a program slice by proposing additional methods for determining slices. Current research frontiers on program slicing are covered by Tip [35]. In abstract terms, a program slice is a subset of a program representation that is based upon some preset criteria. Traditionally, the criteria are formulated as program statements that affect the value of a variable at a particular place in the program text. This formulation of the criterion dictates that a backwards slice be computed from the source statement in question. The notion of forward slices has also been explored. There are several ways that a program slice can be calculated, with one common technique relying upon program-dependence graphs.

The slicing described so far is known as static slicing, because it relies only upon the program text. Researchers have also explored dynamic slicing, which takes into account program execution on a particular input set. In general, dynamic slicing produces smaller slices, which is a benefit to the isolation of program faults. Program dicing is a term used to describe the intersection of multiple slices. Sloane expands the traditional notions of program slicing by generalizing the slicing criteria [12]. His approach relies upon marking an abstract syntax representation of the program using tree decoration capabilities inherent in attribute grammars. One of the advantages of Sloane's approach is that it can easily be used to produce syntactically complete program slices that could be executed.

Program slicing can be expanded to incorporate Feature Engineering. By feature slicing, one could extract a subset of the system that interacts with a particular feature. This would be of critical importance in maintaining individual features, for exploring feature interactions, and for constructing feature relationships in an existing system. Presumably, the intersecting feature slices would indicate potential interactions among feature implementations. This notion was carried out to locate program features and their interactions but mainly for testing and debugging purposes and not evolving system features.

3.7 Regression Testing

Rothermel and Harrold [42] group a variety of selective regression testing approaches into three categories. *Safe* approaches require the selection of every existing test case that exercises any program element that could possibly be affected by a given program change. *Minimization* approaches attempt to select the smallest set of test cases necessary to test affected program elements at least once. *Coverage* approaches attempt to assure that some structural coverage criterion is met by the test cases that are selected. All three categories have been extensively studied by researchers [46][127][48][41][40][88][120][43][124] from a theoretical point of view to either minimize or prioritize test cases.

While minimizing and prioritizing is important, there has been little discussion on applying regression testing in industrial environments, specifically for evolutionary reasons. While researchers are mostly concerned with reducing the number of test cases for the testing process, other important issues in using regression testing in an industrial environment, such as considering regression test case in identifying feature implementation, remain an oversight. Regression testing contain important information in the form of input that reflects the end-user feature and the feature will be invoked.

3.8 Separation of Concerns and Aspect Oriented Programming

Two theories related to our work are the separation of concerns and Aspect-Oriented Programming (AOP). A software system consists of a set of artifacts, such as requirement specifications, designs, and code. Each artifact consists of descriptive material in some formalism, the purpose of which is to model needed concepts in a manner appropriate for that artifact. The formalisms differ for different projects, different phases, and different artifacts perhaps even within an artifact. Different artifacts often share the same concepts, with each concept potentially described in a different way, and with different details, in different artifacts. For example, the word *expression* in the requirements and the term *class expression* in the design.

Many kinds of concerns are important during the software lifecycle. Dimensions of concern help to organize the space of concepts and units. Common dimensions of concern are data or object (leading to data abstraction) and function (leading to functional decomposition). Others include feature (both functional, such as "evaluation," and cross cutting, such as "persistence"), role, and configuration. As illustrated by examples in their work, Tarr and Sutton explain that some dimensions of concern derive from the domain, often aligning with important domain concepts, while others come from system requirements, from the development process, and from internal details of the system itself [103]. In short there are a number of dimensions of concern that might be of importance for different purposes (e.g., comprehension, traceability, reusability, evolution potential), for different systems, and at different phases of the life cycle. However, even Tarr and Sutton admit that a large part of their theory is unproven, and we believe their approach will encounter great difficulties when applied to an existing legacy system.

The AOP community has focused on identifying cross-cutting concerns that appear throughout numerous modules of a system implementation [39][128]. These *aspects* are treated as first-class entities that are "woven" together into the primary modularization to create a final working system. We have found it possible to encapsulate features that are likely to change into fine-grained components, thus avoiding the code-weaving phase of AOP. Also, our fine-grained components are truly reusable whereas aspects appear to only be usable in the context of the original modular decomposition.

3.9 Summary

In this chapter, we discussed the related work as it relates to our research. We looked at several areas such as Software Evolution, Architectural Reconstruction, Feature Engineering, Product Lines, Requirements Analysis, CBSE, Program Understanding, Locating Systems Features, Regression Testing, Separation of Concerns and AOP. Although Turner [25] had identified the problem we are addressing in our work, that was outside the scope of his work. Furthermore, four [5][125][121][99] researchers have described points which are related to our work as far as identifying program features is concerned, however; their motivation is restricted to program understanding and not to the evolution methodology we have developed. To date, no software evolution technique has been proposed that addresses the important issue of evolving legacy code using CBSE and Feature Engineering. We believe that if legacy code is modernized using Feature Engineering and CBSE then many organizations can benefit from the resulting technique.

In Chapter 4, we will discuss the four models that are part of our evolution methodology namely ***Feature Model***, ***Fine-Grained Component Model***, ***Budget Analysis Model*** and ***Formal Model***.

4 Models

Our methodology depends on two important models.

> ***The Feature Model*** defines what a feature is, how it is implemented, how it interacts with other features, and how it is related to other features within the source code.
>
> ***The Fine-Grained Component Model*** describes the constituents of the refactored components using interfaces, properties, and methods.

To support the results of our dissertation we also rely on two additional models:

> ***The Budget Analysis Model*** lists and describes the elements that are necessary for performing the cost-benefit analysis of our evolution methodology.
>
> ***The Formal Model*** provides the theoretical foundation for our evolution methodology. The formal model is supported by the data model.

4.1 Feature Model

As we have already discussed, end-users often view a system in terms of its provided features. Intuitively, a feature is an identifiable unit of system functionality from the end-user's perspective. Examples of features include the ability of a word processor to spell check or the ability of an accounting system to generate a balance sheet statement for a given fiscal year. Software developers are expected to translate such feature-oriented requests into system design. *Feature Engineering* addresses the understanding of features in software systems and defines mechanisms for carrying a feature from the problem domain into the solution domain [26].

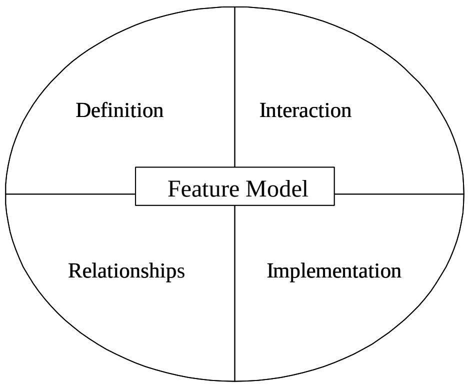


Figure 4.1: Elements of Feature Model.

Our feature model consists of following four elements as shown in Figure 4.1:

Feature definition, what a feature is.

Feature implementation, where and how features are implemented within the source code.

Feature interaction, how a feature interacts with other features.

Feature relationships, how a feature is related to other features.

4.1.1 Feature Definition

We developed the following definition by integrating and extending existing definitions from [82][26]:

> *A **feature** is a group of individual requirements that describes a unit of functionality with respect to a specific point of view relative to a software development life cycle (Figure 4.2).*

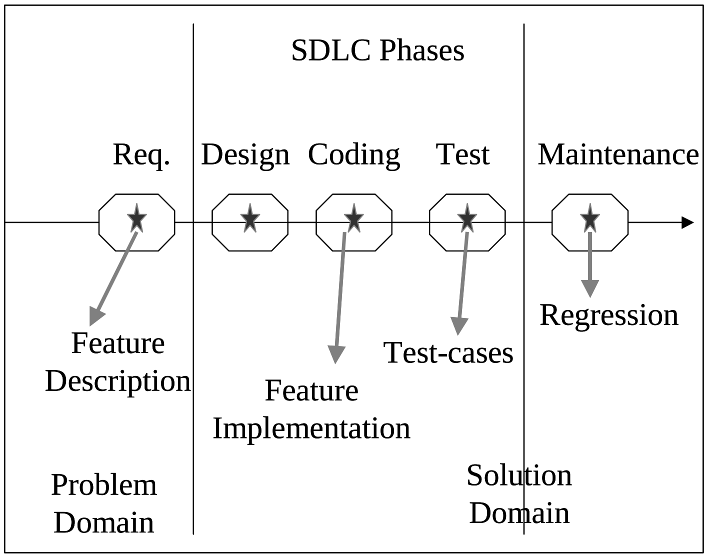


Figure 4.2: Definition of a Feature.

This definition is rooted in the problem domain but shows how a feature can be used in software evolution. For example, a system might support a feature that performs complex calculations in batch mode without user interaction. To an end-user, this feature is a time saver because input can be stored in a file or a database to be used at a later time. At the same time, testers might employ this feature to enable regression testing between two versions of the system; developers might design a specific set of modules to process user input without user interaction to analyze code coverage. A code-profiling tool executing regression test cases exercising that feature can locate the *feature implementation*, and evolution of that feature can commence.

Feature	Functions	Critical Evolution Viewpoint
1	Many	Solution domain
Many	1	Problem domain
1	1	None exists
Many	Many	N/A – Must be decomposed

Table 4.1: Feature/Functions Relationships.

4.1.2 Feature Implementation (FI)

End-users comprehend a system through its features but are unaware of the specific way in which these features are implemented. Software developers view the same system in terms of data types, local and global control, reusable functions, and units of testing and maintenance. Table 4.1 outlines how a feature might be implemented within function(s). When addressing feature implementation we must consider following two scenarios:

- When function(s) and data (local and/or global variable) implements only one feature
- When function(s) and data (local and/or global variable) implements more than one feature

If the function and data implements only one feature than the evolution is trivial. While our models can certainly address the first scenario mentioned above, we are more interested in the second scenario because it is more likely that a function is involved in the implementation of more than one feature. Thus, when we mention feature implementation we assume that the function implements more than one feature. We define feature implementation as following:

> *A **feature implementation (FI)** is the set of statements (including data) within all functions that execute when that feature is invoked. The feature is invoked by one or more test cases.*

When a single feature implementation contains code from many functions then the critical viewpoint regarding evolution is the solution domain because the feature “cross-cuts” the software [87]. Such code is often highly coupled and deeply embedded within the legacy system. When many related features are implemented by a single function then understanding the problem domain is critical for successful evolution. When a feature is implemented by a single function, evolution can be straightforward; a many-to-many relationship must be decomposed further for evolution (Table 4.1).

Given that a function(s) implements more than one feature, there are five cases that capture the essence of feature implementation. In the following example assume that there is a function f_x that is only involved in the implementation of two features FE_1 and FE_2.

4.1.2.1 Case I: Non-interacting (unrelated) features

Figure 4.3 show a function f_x (the large rectangle) implementing two features FE_1 and FE_2 represented as ovals. Even though these two features are implemented in a single function, they do not share any lines of code (LOC) or variables. That is, $FE_1 \cap FE_2 = \emptyset$. At this level of abstraction, it is not important how much of f_x is being executed.

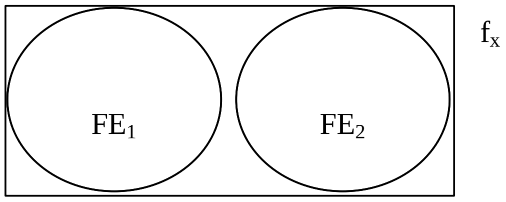


Figure 4.3: Two Features in Function (f_x) but Not Interacting.

4.1.2.2 Case II: Partially interacting features

Figure 4.4 show two features FE_1 and FE_2 sharing LOC or variables in a function. This type of interaction is common and we will discuss this in further detail in Sections 4.2, and 4.3. That is, $FE_1 \cap FE_2 \neq \emptyset$.

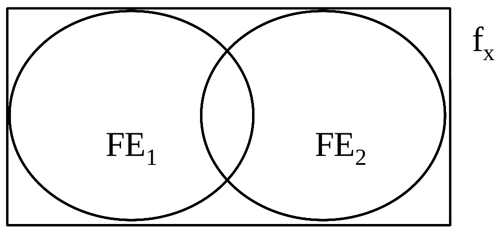


Figure 4.4: Two Features Partially Interacting in Function (f_x).

4.1.2.3 Case III: Fully interacting features

Figure 4.5 show two features FE_1 and FE_2 are fully interacting by sharing LOC and variables. These features are tangled, as there is no apparent distinction between shared LOC and variables using dynamic slicing. Our case study shows how to identify relationships and interactions among fully interacting features. That is, $FE_1 = FE_{2.}$

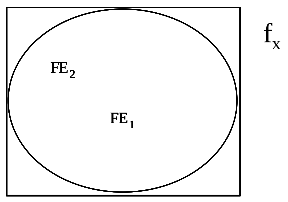


Figure 4.5: Two Features Fully-Interacting in Function (f_x).

4.1.2.4 Case IV: Interacting sub-features

Figure 4.6 show that FE_1 is a subset of FE_2. This could mean that FE_2 is sub-feature of FE_1 or FE_2 is composed of $FE_{1.}$ There are several possibilities in this scenario and we will discuss them in section 4.1.5 and 4.3. Dynamic slicing cannot fully identify the code of either feature thus a closer look at the feature relationships is required. That is, $FE_1 \subset FE_{2.}$

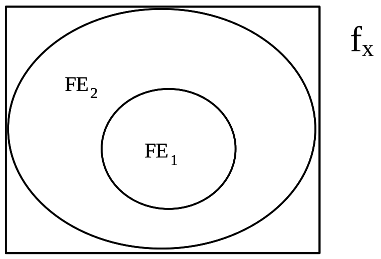


Figure 4.6: Interacting Sub-Feature in Function (f_x).

4.1.2.5 Case V: Interacting super-features

This is just the opposite of case IV as shown in Figure 4.7. That is, $FE_2 \subset FE_{1.}$

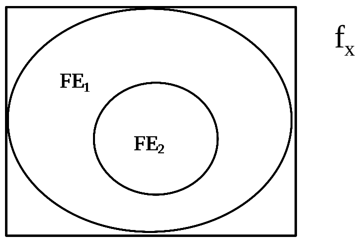


Figure 4.7: Interacting Super-Features in Function (f_x).

4.1.2.6 Summary

At this level of abstraction we ignore the often complicated control flow within a function f_x. Given a function that is involved with no more that two features these five cases (non-interacting, partial-interacting, fully-interacting, interacting-sub and interacting-super features) describe the possible interactions among features.

4.1.2.7 Regression Testing

We propose a novel use of dynamic slicing [18] that uses regression test cases to identify where a feature is implemented in the legacy system and to incrementally refactor the code base to create fine-grained components that can be individually evolved and reused.

Not every feature is evolved during system evolution, nor should each feature be encapsulated in a fine-grained component. We follow a heuristic we call "***The law of two***": if a feature can be used in another system, its implementation becomes a candidate for reuse. From this candidate set, the organization must still select specific features to evolve. These features must be associated with the existing test cases. Once the features are associated with their test cases, our feature model identifies in which functions the features are implemented and what is the feature/function interactions exists. We have identified two scenarios to associate the test case to features:

- Knowledge of the mapping of the test case to the features exists either via domain knowledge or in testing artifacts.

- Knowledge of the mapping of the test case to the features does not exist. In such a case, the input values of each regression test case can be analyzed using clustering techniques.

Domain Knowledge

There is no substitute for domain knowledge in legacy systems. Through using domain knowledge, it is possible to identify test cases that represent a particular feature or a group of features. It is also possible to construct test cases from scratch to exercise a feature. Typically, in an industrial environment the testers have full knowledge of which test cases are used to exercise what features. Our case study assumes that we have this knowledge.

Documentation

Legacy systems also have rich regression test suites that consist of hundreds of test cases. In some cases, test suites are well documented so we can identify easily the test cases used to exercise a given feature.

Clustering and textual pattern analysis

Our simple technique for grouping test cases to find the feature they represent is based on the premise that related test cases exercise either a feature or closely related features. We describe a simple technique to cluster these related test cases in this section. There are several clustering techniques described in the literature.

> ***Clustering analysis** is the organization of a collection of patterns (usually represented as a vector of measurements or a point in multidimensional space) into clusters based on similarity* [89].

Although, Jain et al. describes several clustering techniques, they (including other researchers) have not applied clustering techniques to group related test cases. The purpose of our research is not to explore the clustering techniques but to use them to identify the test case and feature mapping in the event that no documentation of domain knowledge exists. We begin by describing the test cases used in this case study and then provide a simple model that can be used to cluster or logically arrange the test cases that represent the features that need evolution. To illustrate the clustering heuristics consider 10 test cases with 5 sets of items that are considered the most important user inputs (Table 4.2). We analyze the user input and give an ordinal value to each of the valid user inputs for a given *Item*. For example, if item number 1 had ten valid user inputs then the user input was given a numeric value of 1 through 10 respectively. We create a matrix of test cases and *Items* as shown in Table 3. We then use existing tools such as Microsoft Excel™ to calculate statistical measures that can provide some insight on a group (or cluster) of related test cases. For example, if we consider two test cases T4 and T6 (assuming that only items 4 and 5 vary while others are exactly the same) we calculate the regression and standard deviation values to find the best-fit lines. T4, T6, T8 and T2 can be grouped together because their regression values are 2.4, 2.3, 2.2 and 2.1, which is much higher than other test cases indicating that they can be grouped together. Similarly, test cases T1, T3, T5, T7, T9 and T10 can be grouped together because they vary by item 1 and item 5. We can use any of the existing clustering algorithms in this step, but for simplicity, we use regression and standard deviation as our measure to help us define the best fit for the lines. It is possible to use just regression as a measure. However, we suggest that both regression and standard deviation be used because it is quite possible that in a large set of data, two unrelated test cases may end up getting the same value. Using standard deviation as an additional check can help identify such cases. Using such heuristics we can group the test cases into two broad groups; group 1 that exercise feature 1 consists of T4, T6, T8 and T2 and group 2 that exercise feature 2 consists of T1, T3, T5, T7, T9 and T10 in this example (Table 4.2). Likewise, Table 4.3 and Table 4.4 show the result of applying a RankSort clustering algorithm on the test case and items matrix. Note that Table 4.2 and Table 4.4 result in identical clusters as far as mapping test case and features is concerned. In addition, textual pattern analysis can also be used to group these related test cases because test cases often have textual input. Using some pattern searching and developing a simple utility program, one can group the related test cases based upon pre-defined criteria. We found that grouping these test cases into broad categories can help identify the mapping between test cases and features in cases when domain knowledge or documentation is not available. The Pseudo-code for determining clusters for any matrix is shown in Figure 4.8.

Test Cases	Item 1	Item 2	Item 3	Item 4	Item 5	Regression	Std Dev
T4	1	1	1	9	9	2.4	4.38
T6	1	1	1	8	9	2.3	4.12
T8	1	1	1	9	8	2.2	4.12
T2	1	1	1	8	8	2.1	3.83
T1	1	3	3	3	4	0.6	1.1
T5	2	3	3	3	3	0.2	0.45
T3	2	3	3	3	1	-0.2	0.89
T7	3	3	3	3	2	-0.2	0.45
T9	3	3	3	3	1	-0.4	0.89
T10	4	3	3	3	1	-0.6	1.1

Table 4.2:Test Cases vs. Items.

Test Cases	Item 1	Item 2	Item 3	Item 4	Item 5
T1	1	3	3	3	4
T2	1	1	1	8	8
T3	2	3	3	3	1
T4	1	1	1	9	9
T5	2	3	3	3	3
T6	1	1	1	8	9
T7	3	3	3	3	2
T8	1	1	1	9	8
T9	3	3	3	3	1

Table 4.3: Test-case and Items before RankSort.

Test Cases	Item 1	Item 2	Item 3	Item 4	Item 5
T2	1	1	1	8	8
T6	1	1	1	8	9
T8	1	1	1	9	8
T4	1	1	1	9	9
T1	1	3	3	3	4
T9	2	3	3	3	1
T5	2	3	3	3	3
T9	3	3	3	3	1
T7	3	3	3	3	2

Table 4.4: Clustering after RankSort.

```
Create Matrix (M):
    For (Test Case) 1 to I
        For (Input Item) 1 to J
        M(I,J) =Convert (Input Item) to Valid Numeric Value
        Next (Input Item)
    Next (Test Case)
    Return (M)

Apply Clustering Algorithms on (M):
    Choose Any:
         Regression + Standard Deviation
        RankSort
        Any Other

Analyze Modified (M):
    Identify Groups (clustered I)
```

Figure 4.8: Pseudo-code to Determine Clusters.

4.1.3 Features and Functions

We need to identify the percentage of lines of code coverage of a specific feature within a function because this relationship can provide useful information regarding how and where a feature is implemented. This feature/function relationship can be achieved via test cases as these test cases represent the features. In order to successfully identify feature implementation within a function via test cases, we follow the three-step process:

- Identify test case(s) that represent that feature.
- Run the profiler to obtain test case and function execution traces in terms of lines of code.
- Determine UNION of all the lines of code within a function can identify all the lines of code executed in a function by a test case representing that feature. We can calculate the percentage coverage.

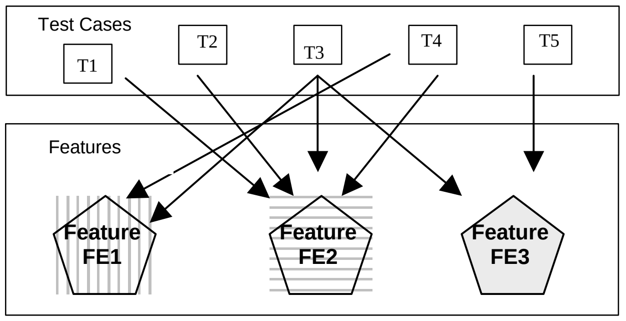


Figure 4.9: A Feature may be Invoked by Several Test Cases.

The feature/function relationship via test cases is either one:one or one:many. When a feature is represented by a single test case (one:one relationship) the code profiler can result in feature/function relationship rather easily as each test case is run and its execution traces are collected. These execution traces consist of lines of code executed in a given function. Even though a function itself can implement more than one feature since each test case represents only a single feature, the identification of lines of code and the coverage percentage within a function is rather trivial in this case because there is no need for the UNION step mentioned earlier. This is because most code coverage tools provide the percentage coverage information.

One:many relationship is more realistic and it is shown in Figure 4.9. It shows that a feature is invoked by many test cases. This case is non-trivial because we must first group the test cases that represent the same feature.

To identify the feature/function relationship, we discuss the three-step process in detail:

4.1.3.1 Step 1: Map test case and features

Test case and feature mapping in a matrix are as shown in Table 4.5. The shaded portion means that the test case can invoke that feature. The shaded cells simply represent that the feature is invoked by the test case. As discussed in Chapter 2, we obtain this mapping from the testers and the end-users.

	Test Case						
Features	T1	T2	T3	T4	T5	T6	T7
FE_1	■			■	■	■	
FE_2		■			■	■	
FE_3		■			■	■	■
FE_4	■		■		■	■	

Table 4.5:Test Case and Feature Mapping.

4.1.3.2 Step 2: Run test case and profiler

Test cases are run with the profiler and the results shown in Table 4.6. Each cell contains the LOC executed by the profiler in function. The next part of this step is to identify the LOC exercised by all test cases in a given function. At this point, we introduce the features that are invoked by all the test cases as shown in

Table 4.7. The next sub-step in this process is to eliminate the OR by making a UNION of all the LOC within the function that implements a feature. The UNION enables us to determine all the lines of code executed by the feature within the function. This is shown in Table 4.8. Finally, we calculate the percentage coverage for the illustration purposes assuming 10 lines of code per function, as shown in Table 4.9.

Functions/Test Cases	T1	T2	T3	T4	T5	T6	T7
f_x	1,2,5,10	0	0	0	0	1,2,3,4	1,2,3,4
f_y	1,2,3,5,8,10	1,2,3,4,5,6,7,8,9,10	0	1,2,3,4,5	0	1,2,3,4	0
f_z	0	0	1,2,3,4,5	1,2,3,5,8,10	0	0	0
f_a	0	1,2,3,4,5,6,7,8,9,10	0	0	0	0	0
f_b	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10
f_c	1,2,3,4,5,6	1,2,3,4,5,6,10	0	0	1,2,3,4,5,6	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10

Table 4.6: Test Case and Function Relationship by Profiler.

Test Case	T1, T4, T5, T6	T2, T5,T6	T2, T5, T6,T7	T1, T5,T6
Functions/Features	FE_1	FE_2	FE_3	FE_4
f_x	1,2,5,10 OR 1,2,3,4	1,2,3,4	1,2,3,4 OR 1,2,3,4	1,2,5,10 OR 1,2,3,4
f_y	1,2,3,5,8,10 OR 1,2,3,4,5 OR 1,2,3,4	1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4	1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4	1,2,3,5,8,10 OR 1,2,3,4
f_z	1,2,3,5,8,10	1,2,3,4	1,2,3,4	1,2,3,4
f_a	0	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10	0
f_b	1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4,5,6,7,8,9,10
f_c	1,2,3,4,5,6	1,2,3,4,5,6,10 OR 1,2,3,4,5,6	1,2,3,4,5,6,10 OR 1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4,5,6,7,8,9,10 OR 1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6

Table 4.7: Test Case, Features, Function and LOC.

Test Case	T1, T4, T5, T6	T2, T5,T6	T2, T5, T6,T7	T1, T5,T6
Functions/Features	FE_1	FE_2	FE_3	FE_4
f_x	1,2,3,4,5,10	1,2,3,4	1,2,3,4	1,2,3,4,5,10
f_y	1,2,3,4,5,8,10	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,8,10
f_z	0	1,2,3,4	1,2,3,4	1,2,3,4
f_a	0	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10	1,2,3,4
f_b	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6,7,8,9,10
f_c	1,2,3,4,5,6	1,2,3,4,5,6,10	1,2,3,4,5,6,7,8,9,10	1,2,3,4,5,6

Table 4.8: UNION of all LOC for a Feature Implementation.

Test Case	T1, T4, T5, T6	T2, T5,T6	T2, T5, T6,T7	T1, T5,T6
Functions/Features	FE_1	FE_2	FE_3	FE_4
f_x	60%	40%	40%	60%
f_y	70%	100%	100%	70%
f_z	0	40%	40%	40%
f_a	0	100%	100%	40%
f_b	100%	100%	100%	100%
f_c	60%	70%	100%	60%

Table 4.9: Percentage LOC (Feature-Function Relationship).

4.1.3.3 Step 3: Develop heuristics

Table 4.9 provides information regarding how features and functions may be related. Similar analysis regarding feature and data will be explored in Section 4.1.5.2. The types of information we can deduce from Table 4.9 are as follows:

Sub-features

While features are visible to end-users, they invariably consist of sub-features that may or may not be visible to the end-user. These sub-features are usually present when a feature is tested by many test cases.

Feature implementation (FI)

Table 4.9 provides information on implementation of a given feature in many functions. This information is very useful for our methodology. FI is the number of functions the feature is implemented in.

CORE

If a function(s) is executed 100% of the time for all features then we define that function to be part of CORE. Such functions are candidates for a shared library. Typically, functions that manipulate strings, round numbers, handle database connections etc. are part of CORE. These functions are most always stateless.

Base-line Architecture

When the system is invoked in a batch mode as discussed in Section 2.2.3, all the test cases execute 100% of certain functions that are not part of any feature. These functions are typically part of system initialization, system shutdown and setting up global variables. It is important to understand that these functions are not part of CORE but are part of system architecture and global control structure. Turner [26] also calls such functions as a base-line architecture. These functions not shown in Table 4.9. Base-line architecture is product specific and contains specific caller-callee sequence, and is unlikely to be reusable into another components. We argue that such code does not change often and is not a candidate for evolution into a component using our methodology. Using domain expertise and results from the profiler, the base-line architecture is identified and subtracted from the code to be evolved. However, if the base-line architecture itself is considered for evolution then it is important to realize that this architecture is represented at a coarse-grained (not fine-grained) level. Thus, such evolution activities will result in either wrapping or re-architecting the entire legacy system, both of which is certainly outside the scope of this dissertation as discussed in Section 1.5. Table 6.7 shows example of functions that are part of base-line architecture.

Neighboring features (K)

Using Table 4.9, we see how features may interact within a function. Traversing through the matrix, one can identify the features that are interacting within a function. We start with a given feature and traverse down to each function where the coverage is greater than 0%. Once the feature implementation is identified, we traverse in the horizontal direction to identify the coverage of other features in that function (greater than 0%). Thus, traversing down, across and then up can provide which are the neighboring features (see Figure 4.10). These neighboring features form relationships that we will discuss in Section 4.1.5.

Evolution threshold (T)

Our methodology can be used to evolve any feature. However we found that rewriting is preferable to features that cross-cut across many functions and interact with a lot of features. Using Table 4.9, heuristics concerning threshold can be developed that can identify features that are good candidates for evolution and that are not.

We use Table 4.9 to identify neighboring features, the number of functions where the feature is implemented, and then we calculate the average coverage percentage within the function. This provides us a threshold regarding evolving the feature, and provides heuristics on whether to continue with the evolution methodology or not. We realize that the high and low values used in Table 4.9 to determine whether or not to continue with our methodology depend on the particular feature(s) and legacy system(s). In our experience, we found that our methodology works best when K = 3, FI = 17 and C = 80% (a more detailed analysis is provided in Section 7.1.1). As a rule-of-thumb, we can say that if K is high than the feature we are trying to evolve cross-cuts through many other features. Conversely, if K is low than the feature we are trying to evolve is trivial. Likewise, if a feature is implemented in many functions (i.e. high FI) than the feature is likely to be scattered in numerous functions. If FI is low (perhaps 1 or 2) than it is a trivial case as the feature is totally contained in a low number of functions. Lastly, if the average coverage of the feature within a function is low than it certainly does not make sense to evolve that feature. However, whether to continue or not is a function of K, FI and C all of which depend on the feature being evolved. Figure 4.10 provides Pseudo-code for calculating K, FI, C and CORE.

	Evolution Threshold (T)			
Feature to be evolved	**Neighboring feature (K)**	**Number of functions (FI)**	**Average Coverage (C)**	**Continue?**
FE_n	High	High	High	No
FE_n	High	High	Low	No
FE_n	High	Low	Low	No
FE_n	High	Low	High	Possibly
FE_n	Low	High	High	Possibly
FE_n	Low	High	Low	No
FE_n	Low	Low	High	Possibly
FE_n	Low	Low	Low	No

Table 4.10: Evolution Threshold (T).

```
Map Test Case and Features:
        For Each (Test Case)
                Identify Features Represented
        Next (Test Case)
        Return Matrix (Test Case and Features)

Run Test Case with Profiler:
        For Each (Test Case)
                Run Profiler to obtain LOC in Functions
        Next (Test Case)
        Return Matrix (Test Case and Functions)

Map Features/Functions:
        For Each (Feature)
                Use Matrix (Test Case and Features) and Matrix (Test Case and Functions)
                UNION LOC representing a Feature in Functions
                Calculate Percentage Coverage of a Feature in a Function
        Next (Feature)
        Return Matrix_Features_Functions

Develop Heuristics:
        Select Feature Column to Evolve from Matrix_Features_Functions

        Calculate F1:
                For Row = 1 to MaxRow (Matrix_Features_Functions)
                        If Matrix_Features_Functions (Row, Column) >0% Then
                                F1 = F1 + 1
                        End if
                Next Row
                Return F1

        Calculate K:
                For Row = 1 to MaxRow (Matrix_Features_Functions)
                        If Matrix_Features_Functions (Row, Column) > 0% Then
                                For Columns = 1 to MaxColumns (Matrix_Features_Functions)
                                If Columns != Column Then
                                If Matrix_Features_Functions(Row, Columns) 0% Then
                                        K = K +1
                             End if
                            End if
                           Next Columns
                      End if
                Next Row
                Return K

        Calculate C:
                For Row = 1 to MaxRow (Matrix_Features_Functions)
                        PercentageCoverage = Percentage Coverage +
                   Matrix_Features_Functions(Row, Column)
             Next Row
            C = PercentageCoverage / MaxRow (Matrix_Features_Functions)
            Return C

Calculate CORE:
                Array IsCORE(MaxRow (Matrix_Features_Functions))
             For Row = 1 to MaxRow (Matrix_Features_Functions)
                For Columns = 1 to MaxColumns (Matrix_Features_Functions)
                        If Matrix_Features_Functions(Row, Columns) = 100% Then
                                IsCORE (Row (Function)) = TRUE
                        Else
                                IsCORE (Row (Function)) = FALSE
                                Break;
                        End if
                Next Columns
            Next Row
                Return IsCORE(Row(Function))
```

Figure 4.10: Pseudo-code for Heuristics.

4.1.4 Feature interactions

A legacy system has many features. These features must interact with each other to provide wider system functionality. When features interact with each other, they have an "effect" on the system. Depending upon the state of the legacy system, this effect can be either positive or negative (resulting in errors). We must distinguish between intended interactions between features, interactions between features that are not intentional but don't result in errors (or may even have positive side-effects) and unintended and undesirable feature interaction not known in advance and leading to faulty applications. Figure 4.11 shows a classification of feature interaction and their side effects.

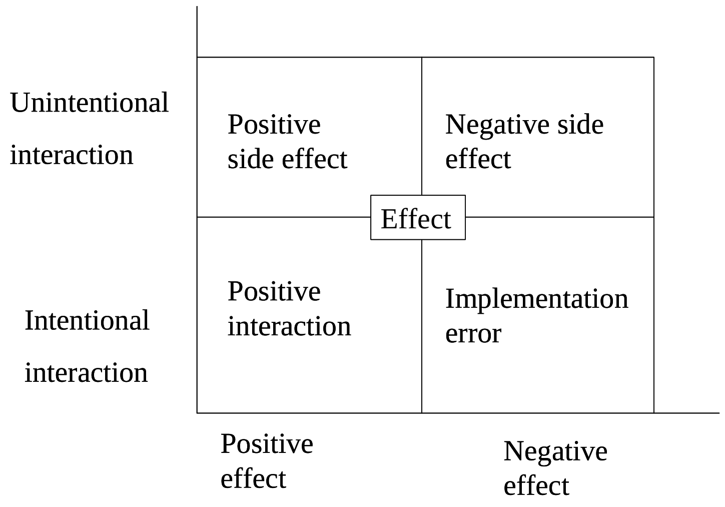


Figure 4.11: Classification of Feature Interaction.

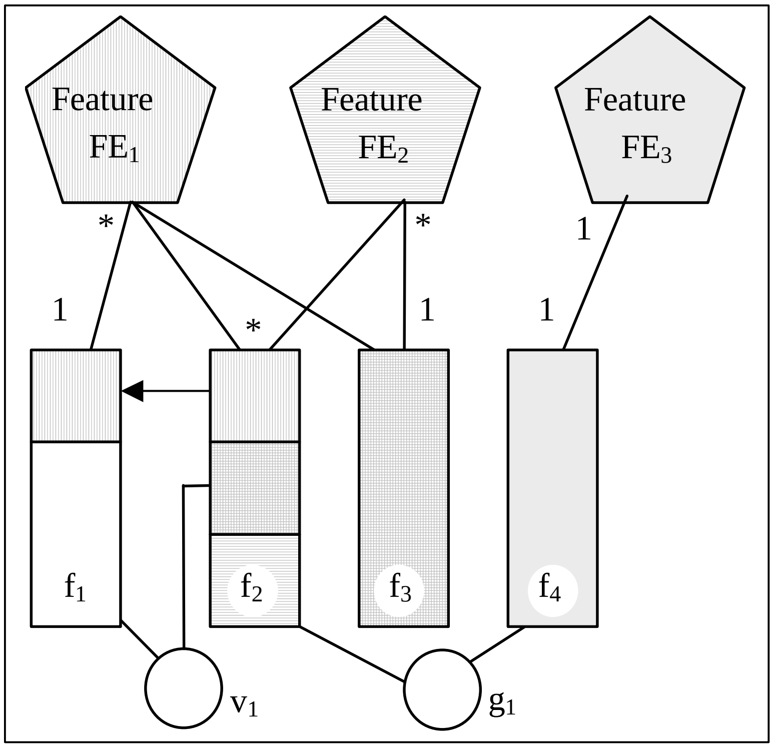


Figure 4.12: Feature Interaction via Functions and Data.

In Figure 4.12, functions are represented as rectangles, variables (both local and global) as circles, and features as pentagons. FIs are shaded using the same pattern as their corresponding feature (shown by the lines between pentagon and rectangle). A feature implementation is the set of shaded regions among the function rectangle. When two or more feature implementations share common data or functions, there are four key interactions.

4.1.4.1 Shared Stateless Function (SS)

A stateless function [72] can be shared between two FIs. For example, all statements in function f_3 are executed when both FE_1 and FE_2 are exercised and f_3 does not access any local or global data.

4.1.4.2 Shared State-Full Function (SSF)

A state-full function [72] can be shared between two features. Refactoring may be complex, involving analyzing global variable access and control structures. Function f_2 accesses global variable g_1 and since f_2 is part of both FI_1 and FI_2, there is an implicit interaction between FE_1 and FE_2.

4.1.4.3 Dependent Data (DD)

An FI may be dependent on the data that is updated by another FI. For example, f_1 and f_2 access the local variable v_1 leading to an interaction between FE_1 and FE_2.

4.1.4.4 Dependent Function (DF)

An FI may be dependent on a function that is part of another FI. Function f_2 calls function f_1 (shown by the arrow in Figure 4.12) when FE_1 is exercised but not when FE_2 is exercised (note the consistent shading). The remaining statements in f_1 (shaded white) are associated with another feature not shown and FE_1 interacts with that feature. When a feature is fully contained in a single function, the implementation could be equally complex. Such a function may be stateless or it could depend on global data (as is the case with f_4 in Figure 4.12). As each feature is exercised, *code-profiling* (or similar) *tools* identify the code slices associated with each feature, providing the details necessary to identify interactions between features.

4.1.5 Feature Relationships

Turner identified several relationships among interacting features [26]. In this section, we integrate and extend Turner's idea of feature relationships into our feature model. Understanding feature relationships allows us to better:

1. Interpret feature interactions as feature relationships refine the concept of interaction by providing specification through calling sequence.
2. Refactor the existing FI into fine-grained component(s).
3. Define the communication among fine-grained component(s) that will compose the large reusable unit.

We will discuss how these relationships are implemented within FIs.

4.1.5.1 Categories

We expand this concept into direct and indirect relationships among interacting features and map it into FI. We categorized the feature relationships in two broad categories as shown in Figure 4.13:

1. ***Indirect relationships*** are problem domain relationships and are abstract in nature. These relationships are important when talking to the end-user and usually exist at the application level rather than at the function level.

The end-user comprehends the system to be composed of several features and has a perspectives with respect to the software functionality. Within the indirect relationship, a feature may be a *composed*, *generalized*, or *specialized* part of another feature. These relationships are usually visible to the end-user view and reside in the problem domain.

2. ***Direct relationships*** are solution domain entities. These relationships may be *visible* to the end-users and typically consist of several *sub-features.* These relationships have concrete FIs associated with them. For example, in an application when the user performs "*file open*" command, the data is loaded from a database field and displayed on the screen. Within the code, the data may pass through series of transformation, such as error checking, checking dependency on other fields, and change its appearance. To an end-user, this feature may be that of a simple "*file open*", but this feature is composed of several sub-features such as error-checking, dependency-checking, and transform-view. Both error-checking and transform-view *require* dependency-checking to set certain state. These relationships can be identified by inspecting the feature implementation. Within the direct feature relationships, a feature relationship with another feature may be that of *shared, altered, required, conflict,* and *compete.* It is to be noted that both compete and conflict are example of features that are implemented using multiple operating system

threads. In contrast, feature relationships of type *required, shared and altered* are examples of features that are implemented using single threads.

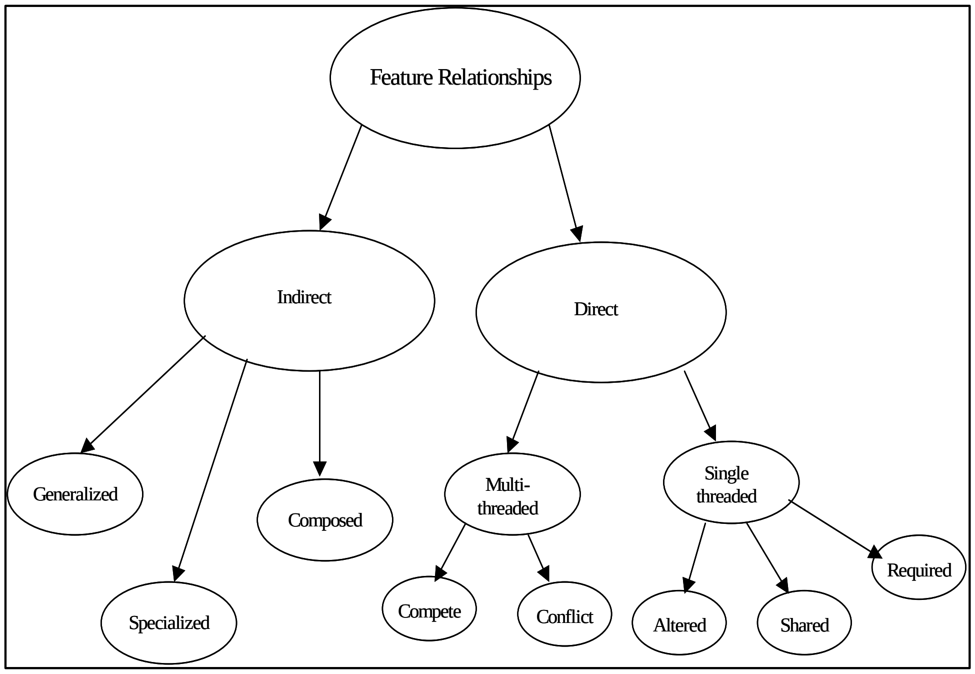


Figure 4.13: Feature Relationships.

We define each one of the feature relationships:

> *A* ***composed relationship*** *shows how a feature is composed of several sub-features. An example of a composed relationship is that a bank account consists of savings and checking accounts.*
>
> ***Generalized and specialized relationships*** *usually co-exist and they depend of particular point of view and granularity. An example of generalized feature is an application that can integrate assets and liabilities. An example of specialized feature is an application that can integrate executive benefits and life insurance, where executive benefit is the liability to be funded by the life insurance asset.*

> *When a feature is required to be present for other features to function, it is known as* ***required relationship****. For example, in order for paste feature to work, the cut/copy feature must exist.*
>
> *When a group of feature share resources (global data, objects or other implementation) with other feature(s) then a* ***shared relationship*** *among features exists. For example, Windows™ clipboard shares the text copied to it with other applications.*
>
> *When a feature's state (global data, object or implementation) is altered by another feature then there is an* ***altered relationship*** *between features. For example, a textbox turns red in color when an error is identified (in its content).*

Most contemporary programming languages do not allow creating multiple threads within a function. Thus, the feature relationships *compete and conflict* are found at an application level rather at a function level. Our feature model addresses feature interaction issues at a finer-granularity (i.e. at a function level). We suggest that compete and conflict relationships be addressed at a higher level of abstraction (i.e. at the architectural level) rather than by our feature model. Furthermore, multithreaded systems probably need to be rewritten rather than evolved because of the inherent complexity in maintaining them. Direct relationships are most commonly found in the solution domain.

4.1.5.2 Determining Feature Relationships

While indirect relationships are important, their purpose is mainly to communicate with end-users. There is no FI associated with indirect relationships because these relationships are abstract in nature. Since we are interested in evolving the FIs into fine-grained components, our methodology is focused on direct relationships. The following three elements are important to understand how features are related:

1. ***Granularity*** identifies any neighboring features associated with the feature that we are trying to evolve. There are two levels of granularity, inter-function and intra-function. In the inter-function, the FI is exclusively in a function and that function does not implement any other FI. The neighboring features communicate via global data. Identification of FI is simple as features are already contained in independent functions. These FI may not need any evolution. In the case of intra-function, the function may have several FI and these FI may be interacting via global and/or local data. Our methodology addresses feature interaction at an intra-function level.
2. ***Order of execution*** identifies which neighboring feature is executed first.
3. ***Variable analysis*** identifies which variables (local and global) are used among neighboring features and how. Along with the order of execution, variable analysis identifies which global or local variables within the neighboring features changed due to the execution.

We have developed techniques to identify feature relationships based upon order of execution, change in the state of variables, feature interaction and feature implementation. Below are five scenarios that allow us to identify direct feature relationships:

4.1.5.3 Altered and Required via DD

Both altered and required relationships have been shown in Table 4.11. The sequence of execution is shown in the leftmost column. The program declares a global variable g_1. Assume that FE_1 *sets* the value of this global variable and FE_2 and FE_3 *use* this variable only after it has been set. Thus, FE_2 and FE_3 require FE_1.

The altered relationship assumes FI to be in one function (f_x). This FI could be any of the five cases discussed in Section 4.1.2. FE_1 *declares* and *sets* the value of a local variable v_1. FE_2 changes the value of v_1. Thus, there is an altered relationship between FE_1 and FE_2.

Execution Sequence	Declare	Set	Use	Change
Program	g_1			
FE_1	v_1	v_1,g_1	v_1	
FE_2			g_1	v_1
FE_3			g_1	

Table 4.11: Altered and Required Relationship via DD.

4.1.5.4 Altered and Required via SSF

Like the scenario shown in Section 4.1.5.3, the alteration of a variable can also happen in an SSF. Furthermore, the required relationship depends either on a local or a global variable, (also seen in Table 4.12).

Execution Sequence	Declare	Set in f_y	Use in f_x	Change f_x or f_y
Program	g_1			
FE_1	v_1	v_1,g_1	v_1	
FE_2			g_1	v_1
FE_3			g_1	

Table 4.12: Altered and required relationship via SSF.

4.1.5.5 Shared via DD and SSF

There is a subtle difference between a required and shared relationship. The root of this difference is in the execution sequence. While it is necessary for a required relationship to be executed sequentially, this constraint is not required by the shared relationship. Thus, FE_2 and FE_3 can execute at any point in time as long as they simply use (share) the state of either local or global variable set by FE_1. This is true for both DD and SSF as shown in Table 4.13 and Table 4.14 respectively.

Execution Sequence	Declare	Set in f_y	Use in f_x	Change f_x or f_y
Program	g_1			
FE_1	v_1	v_1,g_1	v_1,g_1	
FE_2 OR FE_3			v_1,g_1	

Table 4.13: Shared Relationship via DD.

Execution Sequence	Declare	Set in f_y	Use in f_x	Change f_x or f_y
Program	g_1			
FE_1	v_1	v_1,g_1	v_1,g_1	
FE_2 OR FE_3			v_1,g_1	

Table 4.14: Shared Relationship via SSF.

4.1.5.6 Compete via DD

Features can compete with each other. In Table 4.15, FE_2 and FE_3 compete to change the values of variables v_1 and g_1 that was *set* by FE_1. The execution sequence of FE_2 and FE_3 is not important; only their intent to *change* the values of v_1 and g_1 is. The scenario shown in Table 4.15 can also exist with SSF and DF.

Execution Sequence	Declare	Set in f_y	Use in f_x	Change f_x or f_y
Program	g_1			
FE_1	v_1	v_1,g_1		
FE_2 AND FE_3				v_1,g_1

Table 4.15: Compete Relationship via DD.

4.1.5.7 Conflict via SSF

Features can be in conflict with each other. This conflict happens when they are trying to *set, use* and *change* local/global variables at the same time. This is shown in Table 4.16. Although, a scenario with SSF is shown DD and DF can also exhibit the same scenario.

Execution Sequence	Declare	Set in f_y	Use in f_x	Change f_x or f_y
Program	g_1			
FE_1	v_1	v_1,g_1	v_1,g_1	v_1,g_1
FE_1 OR FE_2 OR FE_3			v_1,g_1	v_1,g_1

Table 4.16: Conflict Relationship via SSF.

4.1.5.8 Summary

If a feature can be used in another system, its implementation becomes a candidate for reuse. When features are represented by many test cases, FI can be identified by the UNION of lines of code within the function(s). If test case and feature mapping is unknown, simple clustering techniques such as RankSort can help. Running the entire regression test case provides several interesting heuristics and information on sub-features, FIs, CORE, base-line architecture, neighboring features and threshold. Threshold data provides whether to continue with the methodology or not. Features interact with each other via global and local data. There are four ways how features interact; SS, SSF, DD and DF. Feature interactions allow us to identify relationships among features. Understanding feature relationships allow us to refactor FI into explicit fine-grained components. Our methodology addresses evolution issues with single-threaded direct relationships of type required, shared and altered.

4.2 Fine-Grained Component Model

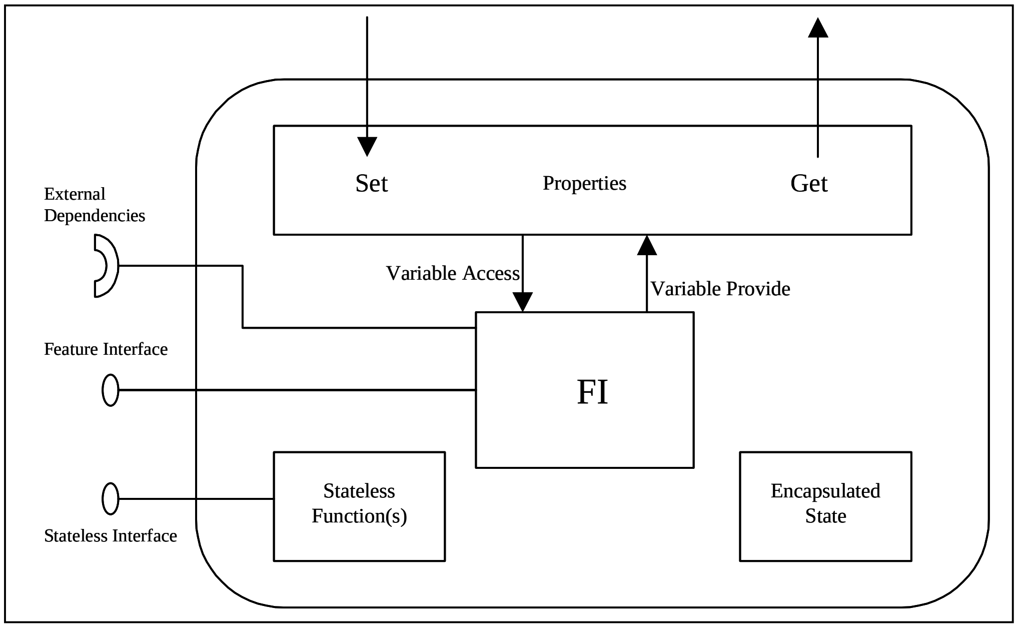


Figure 4.14: Fine-Grained Component Model.

Our fine-grained component (FGC) model is technology-independent and can be implemented using any of the contemporary technologies such as Microsoft ActiveX/COM or SUN JavaBeans. While an FGC can maintain like an EJB Session Bean, it may require basic data that is passed to it through its *Properties*. Since our evolution methodology is incremental, the FGC model encapsulates a feature implementation that can be invoked by its public interface. A FGC can provide data back to the legacy system via *Property Get*. A FGC can also implement any SS and their interface. Finally, a FGC can access any external dependencies (such as SSF, CORE, public functions within the legacy system or even other components) via specifying the external functions as shown in Figure 4.14. Likewise, the stateless functions can also have external dependencies (not shown in the figure).

An FI is often scattered across many system functions and may access local or global data. FIs can be identified and encapsulated into fine-grained components using the component model shown in Figure 4.14.

We borrow the definition of component and component model from [38]:

> *A **component** is a software element that conforms to a **component model** and can be independently deployed and composed without modification according to a composition standard.*
>
> *A **component model** defines specific interaction and composition standards.*

Our fine-grained component model has the following aspects Properties, Feature Implementation, Stateless Functions, and Encapsulated State:

4.2.1 Property Set

Our feature model explains the importance of global and local variables when evaluating feature relationships, as discussed in Section 4.1.5. Essentially, when we refactor the FI code in the legacy code we disable the old code within the legacy code. This disabled code requires access to several local and global variables from the legacy system. *Property Set* is a way to pass these variables to the refactored FI. Thus, *Property Set* must be called prior to invoking the FI.

4.2.2 Property Get

Like *Property Set*, the FI can change the state of certain local and global variables that the legacy system may need to continue to function properly. Using *Property Get*, the legacy system retrieves the values of these local or global variables.

4.2.3 Feature Implementation (FI)

The FI from the legacy code is refactored and encapsulated here. This may contain several functions, classes, local data and other data structure as the feature implementation. This implementation provides an interface, which is called by the legacy system and other product lines. This FI can call other fine-grained components, CORE or any other externally dependent functions as well. It acts as the single point of entry for the feature thus providing explicitness to a feature functionality encapsulated in FGC.

4.2.4 Stateless Function(s)

The FI may need Stateless functions (SS) that are not part of CORE, and other FI may not call that SS. In such cases, this SS can be part of fine-grained component. Its interface is exposed and can be called by the legacy system (or any parent application). Like FI, the SS can also call other fine-grained components, CORE or any other externally dependent functions.

4.2.5 Internal State

The fine-grained component may maintain its own state. This is analogous to an EJB Session Bean. This state is maintained by variables local to the component. Maintaining state has its advantages and disadvantages. It allows for better performance as the fine-grained component retains the values of its variables from one call to another. This saves the recalculation/resetting of variables. However, in a multiple-user environment, maintaining state can overload the server resources because state is stored in memory. State can also be serialized in a database, which usually provides a good compromise between performance and load issues discussed earlier. Our fine-grained component model allows for maintaining the state but leaves the implementation to the developer.

4.2.6 External Dependencies

SSF, CORE and other components can be called “out” of the fine-grained component to access any data needed via this interface. This interface can be implemented using “events” to access any state set by an SSF. Typically, external dependencies are a list of declaration of functions and other components that the FI or the SS may need within FGC.

4.3 Evolving Feature Implementation into Fine-Grained Components

Once we identify a FI using code profilers and similar tools such as χSuds [1] and NuMega's *TrueCoverage*™[37], we refactor that FI into a fine-grained component.

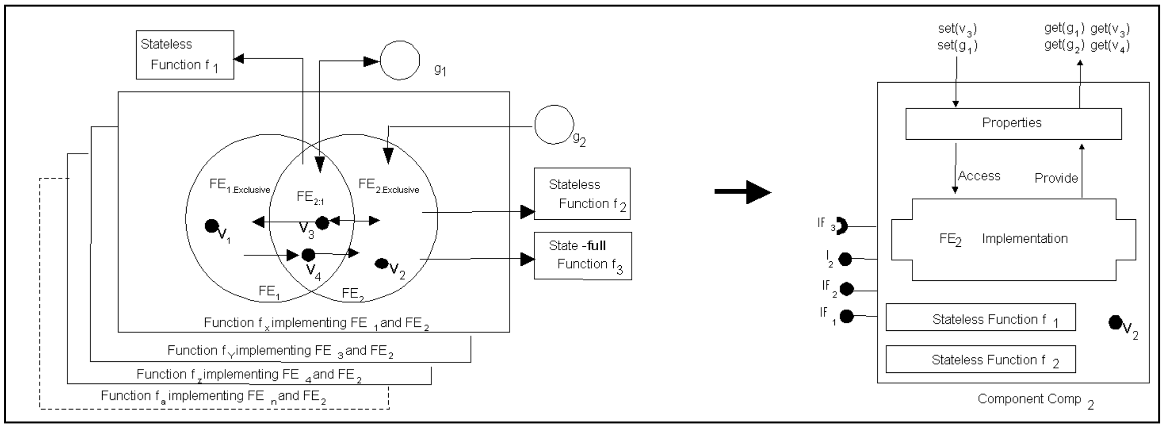


Figure 4.15: Evolving FI into a Fine-Grained Component.

In the fine-grained components developed in this dissertation, the interaction between components is clearly specified by the interfaces. Components can also access functionality using stateless interfaces. The FI is shielded from specific variable implementations by using the interface for external access; over time, the variable implementation will be replaced with explicit linkages to external interfaces.

The first step is to isolate each function that contains code belonging to the target FI. This analysis is often complex if because local variables, global variables, and dependent functions can be shared between FIs as discussed in Section 4.1.5. Our component model attempts to “share” the functions as well as the data that is scattered across various functions through explicit interfaces.

The left part of Figure 4.15 shows a single function f_x whose code is shared between FI_1 and FI_2. Similarly function f_y is involved in **FE_3** and **FE_2.** The purpose of cascading functions is to show that **FE_2** is spread in many functions, and interacts with other features. This simple example highlights all characteristics of our model. Common code and variables include: calls to SS f_1, global variable g_1, and local variables v_3 and v_4. Extracting FI_2 into $comp_2$ involves several artifacts. Function f_1 can easily be extracted because it is stateless. Double arrowheads on the arrow to g_1 show that it is both read and updated by FI_2. Local variables v_3 and v_4 are used by both FIs but FI_2 only reads v_4 (as shown by arrowhead), while v_3 is both updated and read by FI_2; v_4 is set by FE_1 but v_4 is used by $FE_{2:1}$. FI_2 also accesses global variable g_2, SS function f_2, and SSF f_3. There are several important regions in Figure 4.15.

- **FE_1**: The complete code in f_x that belongs to FE_1.
- **FE_2**: The complete code in f_x that belongs to FE_2.
- **$FE_{2:1}$** $FE_{2:1}$ is the shared code among **FE_1** and **FE_2** that is responsible for the cross-cutting problem associated with features that makes evolution of

legacy systems extremely difficult. When two or more feature implementations share variables and functions, as shown above, one must evaluate how they share code and data. The region $FE_{2:1}$ implicitly defines feature relationships because either global or local variables are used or changed (see Section 4.1.5.2). It is also important to understand the relationships among features during an evolution exercise. A detailed analysis of feature relationships that can be found in $FE_{2:1}$ is provided later in this chapter.

- **$FE_{1.Exclusive}$**: The complete code in f_x that belongs exclusively to FE_1 and is not shared with any other FIs including $FE_{2:1}$.
- **$FE_{2.Exclusive}$**: The complete code in f_x that belongs exclusively to FE_2 and is not shared with any other FIs including $FE_{2:1}$.

$Comp_2$ in Figure 4.15 encapsulates FI_2 and has several public interfaces, represented by circles attached by lines to $Comp_2$ to enable original code to access the moved artifacts. $Comp_2$ maintains data previously local to f_x, replaces global variable references with an interface that treats such data as properties, and contains stateless and state-full functions. Public interface I_2 is the primary interface for $Comp_2$. Stateless functions f_1 and f_2 are also encapsulated into $Comp_2$ and they can be accessed via the public interfaces IF_1 and IF_2. SSF f_3 is accessed with IF_3; through an outgoing interface, it is assumed that f_3 is not located inside $Comp_2$ but its state is accessed via IF_3. Local and global variables used by FI_2 can be accessed via `Get/Set` properties. Additionally, the `get` property provides a way to share local and global variables with other feature implementations. As related features are evolved, the interaction between fine-grained components will become increasingly specified and all implicit communication will vanish. Thus, we separate accessing variables from their implementation. When multiple features are extracted at the same time, many stateless functions will be common to several feature implementations; these will be encapsulated within a *core* component, rather than a fine-grained component, and will be treated as a shared library.

The interface for $Comp_2$ is a result of variables and functions that are needed for FE_2 implementation. We now discuss what constitutes the FE_2 implementation both at the function and at the component level:

FE_2 implementation at the function level consists of code in function f_x that is exercised only for FE_2 (defined as $FE_{2.Exclusive}$) plus the code implementation that is shared between FE_2 and FE_1, (defined as $FE_{2:1}$). $FE_{2.Exclusive}$ is simple to identify and typically it will be separated by explicit control structures such as IF…THEN…ELSE or SWITCH…CASE statements because it is unique to a particular feature implementation. When evolving the exclusive code there are two possible routes developers can take; they can simply cut and paste this code into the component, or this code can be refactored and then implemented into the FE_2 implementation of $Comp_2$. Typically, the challenging part is to understand rather than to identify $FE_{2.Exclusive}$ because as mentioned above code profiling tools will identify this unique code but understanding remains implicit many times.

The more complicated case arises when we are dealing with $FE_{2:1}$ because the code is sequentially executed in this shared part of f_x making it hard to isolate the code associated with either FE_1 or FE_2. The net result is that the test cases for feature 1 and feature 2 will reveal the same code in $FE_{2:1}$. At this point domain knowledge may be needed to understand the feature relationships for refactoring and evolution. For example, it is possible that the FE_2 implementation is dependent on the presence of FE_1. In such instances, it is possible that both feature implementations be evolved at the same time.

Although there is no substitute for the domain knowledge, our feature model identifies various relationships that may exist among features and can address the issue discussed above. Using the results from Section 4.1.5, we can understand the relationships among the features that can provide the local and the global variables involved. These variables (as discussed in Sections 4.1.2 and 4.1.5) can be used to:

1. Identify FE_1, FE_2, and $FE_{2:1}$
2. Form the *Property Get/Set* of $Comp_2$

Evolving a FI into a component requires identifying the neighbouring features within a function and code exclusive to the feature to be evolved. The variables that are required to execute (or updated) the FI become the properties of the component.

4.3.1 Evolution Considerations

To provide heuristics for evolving $FE_{2:1}$, we now discuss three possible scenarios are discussed:

4.3.1.1 Scenario I - Understanding T(K,FI,C)

If FE_2 is scattered in *FI* functions and its average coverage percentage C in a given function is less than *n* in each of the functions, then the feature is probably not a good candidate for evolution because FE_2 cannot be encapsulated easily into a component and the cost of evolution will be relatively higher. Furthermore, the legacy system will continue to work so there is probably more need for refactoring than encapsulation in case there is a desire to reduce maintenance cost. The variables *K, C,* and *FI* can be application and/or domain specific. The application used as a case study in this dissertation uses a K =3 and C = 80%.

4.3.1.2 Scenario II - Evolving Unrelated Features

If FE_1 and FE_2 share a common implementation, and furthermore they are totally unrelated, then code for FE_2 can simply be extracted and put in $Comp_2$. FE_2 will have to be manually identified. In this case, FE_1 will remain functioning. Since the features share a common implementation and usually there is no control statement that segregates these unrelated features, the code profiler may identify that each feature is fully covered as shown below in Pseudo-code (Figure 4.16).

Function X (i,b) implements FE_1 and FE_2; there is nothing common between these two features and they are totally unrelated. However, the code profiler will identifies 100% coverage when FE_1 or FE_2 is analyzed independently. In such cases, code for FE_2 will have to be manually identified and then moved into the component CompFE2. An in-depth analysis of grouping related features is provided in Section 4.3.1.3.2.

```
Function X (int i, boolean b)

        Code for Feature 1
        Use i
        Use b
        Rest of Feature 1 Code

        Code for Feature 2
        Use i
        Use b
        Rest of Feature 2 Code

End Function
```

```
Function X (int i, boolean b)

        Code for Feature 1
        Use i
        Use b
        Rest of Feature 1 Code

        Call CompFE2.X(i,b)

End Function
```

Figure 4.16: Example of Unrelated Features in One Function.

4.3.1.3 Scenario III - Evolving Related Features

If $FE_{2:1}$ implements two or more features and they are "related closely" to each other, then we can make a copy of the function with an understanding that we will probably evolve other features (shared in F_x) at some later point in time. There are some configuration management issues with this case and must be handled carefully. At a given point in time only one feature is extracted and evolved, as the evolution methodology is incremental in nature. More details regarding feature relationships are discussed below.

4.3.1.3.1 Primitive Features

Before providing specific examples in Pseudo-code for each of the relationship types, we discuss simple cases of what happens when a function is not shared among features. Although these examples are trivial, they do provide background information on how functions that implement multiple related-features should be handled. In Figure 4.17, function X1 () implements code for Feature 2 and no other feature. In addition, this code does not update any local or global variables. The evolved function simply calls a method in component, CompFE2 (not shown in the figure). Note that the control flow within the legacy system is not modified.

```
Function X1 ()
        Code for Feature 2
End Function
```

```
Function X1 ()
        Call CompFE2.X1
End Function
```

Figure 4.17: Function Implementing Code for Only One Feature.

Figure 4.18 describes a dependent data example discussed earlier; FI_2 uses the global variable. Again, the control flow is not modified and code that implements FE_2 is simply encapsulated into method X2. An interesting point in this example is that compFE2 has a property for accepting the variable Y. FE_2 depends on the global variable Y. The setY property in compFE2 is used to pass the value of the global variable from the legacy system.

```
Function X2 ()

            Code for Feature 2
            Use Global Variable Y
            Rest for Feature 2 Code

End Function
```

```
Function X2 ()

            Call CompFE2.SetY(Y)
            Call CompFE2.X2

End Function
```

Figure 4.18: Implementation of Dependent Data.

A more involved example is shown Figure 4.19. When FI_2 updates a global variable Y, a get property is needed to update the state of global variable in the legacy system. Note that the legacy system first passes the global variable into the component before calling X3.

```
Function X3 ()

            Code for Feature 2
            Change Global Variable Y
            Rest for Feature 2 Code

End Function
```

```
Function X3 ()

            CompFE2.SetY(Y)
            Call CompFE2.X3
            Y = CompFE2.GetY

End Function
```

Figure 4.19: Feature Updates Global Variable.

These examples assumed that the function does not include any other feature implementation. While these examples are good for showing the concept, in reality this is hardly the case, as a single function can be included in several features. One such example is shown in Figure 4.20. Function X4 () is involved in both FI_1 and FI_2, furthermore IsOdd () is used by both features. The value of the dependent data v determines which feature will be invoked. In addition, FI_2

changes the value of global variable Z. In evolving FE_2, FE_2 must be considered in function X4 (); code that is common to both (and other) features must be identified and moved to relevant components. For example, function A is moved inside component compFE2 because it is called only by FE_1 and $FE_{2.}$ IsOdd () is moved into core because it is SS and all the features call it. Note that control flow that is common to both features remains in the evolved function.

```
Function X4 (int i, boolean b)

        Declare Local Variable v
        v= Function A(i)
        If b = True Then
          v = v + 1
        Else
          v = v + 2
        End if

        If IsOdd(v) Then
          Code for Feature 1
          Use v,i
          Change Global Variable XX
          Rest of Feature 1 Code
        Else
          Use Global Variable Z
          Code of Feature 2
          Use v,i
          Change Global Variable Z
          Rest for Feature 2 Code
        End if

End Function

Function A (int i)

End Function

Function IsOdd( int v)

End Function
```

```
Function X4 (int i, boolean b)

        Declare Local Variable v
        v= CompFE2.A(i)
        If b = True Then
          v = v + 1
        Else
          v = v + 2
        End if

        If CORE.IsOdd(v) Then
          Code for Feature 1
          Use v,i
          Change Global Variable XX
          Rest for Feature 1 Code
        Else
          Call CompFE2.SetZ(Z)
          Call CompFE2.X4(v,i)
          Z = CompFE2.GetZ(Z)
        End if

End Function
```

Figure 4.20: Single Function Implementing Several Features.

4.3.1.3.2 Determining Feature Relationships

Since some of the basic ideas have been described above, the feature relationships are now discussed. The relationships between features are implemented by functions. The problem domain relationships are discussed in detail, with emphasis on required, alteration and shared, since all three of these feature relationships have a tendency to be implemented in more than one function. Feature relationships must be clearly understood for the purpose of evolution. As an example, when a function is to be evolved and it implements more than one feature, it is quite possible that a code-profiler may not reveal the exact code associated with a given feature. In fact, the profiler may result in the exact same code for both features as shown in Figure 4.21. Function X () implements FE_1 and FE_2. However, the code-profiler results in the exact same lines of code as seen in the left block. FE_1 requires FE_2 because dependent data i is changed by FE_1 and used by FE_2. The global variable i is passed using GetI and SetI to both components namely, $compFE_2$ and $compFE_1$. When Feature 2 only is evolved, the Pseudo-code may look like the one in the center block. The right block represents the code after FE_1 and FE_2 both have been evolved. The purpose in showing all three stages of evolution is to show that the methodology can be applied to evolve just one or both the features.

```
Function X ()

          Use Global i
          Code for Feature 1
          Change i to Something
          Rest of Feature 1 Code

          Code for Feature 2
          Use i
          Rest of Feature 2 Code


End Function
```

```
Function X ()

          Use Global i
          Code for Feature 1
          Change i to Something
          Rest of Feature 1 Code
          Call CompFE2.SetI(i)
          Call CompFE2.X
          i =  CompFE2.GetI

End Function
```

```
Function X ()


          Call CompFE2.SetI(i)
          Call CompFE1.SetI(i)

          Call CompFE1.X
          Call CompFE2.SetI(CompFE1.GetI
          Call CompFE2.X


End Function
```

Figure 4.21: Example of Required Relationship.

The example below in Figure 4.22 shows a more traditional function that has more than one feature implemented. Note that code for Feature 2 is implemented only when the dependent data i is equal to Something. If i is not equal to Something, Feature 2 is never invoked. Thus FE_2 requires FE_1. The right block simply encapsulates the Feature 2 code into a component along with dependent data i and b.

```
Function X ()

            Declare Local Variable i
            Declare Local Variable b
            Code for Feature 1
            i = Initialize
            b = Initialize
            Rest of Feature 1 Code
            Change i to Something
            Change b
            More of Feature 1 Code

            If i = Something Then
                  Code for Feature 2
                  Use b
                  Rest of Feature 2 Code
            End if

End Function
```

```
Function X ()

            Declare Local Variable i
            Declare Local Variable b
            Code for Feature 1
            i = Initialize
            b = Initialize
            Rest of Feature 1 Code
            Change i to Something
            Change b
            More of Feature 1 Code

            If i = Something Then
              Call CompFE2.Set(i)
              Call CompFE2.Set(b)
              Call CompFE2.X()
            End if

End Function
```

Figure 4.22: Example of Required Relationship.

An alteration relationship is similar to required as shown in Figure 4.23. The function prior to its evolution is the one to focus on because the one on the right can have similar implementation as the required relationship. FE_1 alters FE_2 based upon SomeSpecificValue of b. Since, entire code of FE_2 is encapsulated, the evolved function in the right block looks similar to the required relationship; however, they are quite different.

```
Function X ()

      Declare Local Variable i
      Declare Local Variable b
      Code for Feature 1
      i = Initialize
      b = Initialize
      Rest of Feature 1 Code
      Change i to Something
      Change b
      More of Feature 1 Code

      If i = Something Then
              Code for Feature 2
              Use b
               If b = SomeSpecificValue Then
                  Do something different for Feature2
               Else
                  Rest of Feature 2 Code
               End if
      End if

End Function
```

```
Function X ()

            Declare Local Variable i
            Declare Local Variable b
            Code for Feature 1
            i = Initialize
            b = Initialize
            Rest of Feature 1 Code
            Change i to Something
            Change b
            More of Feature 1 Code

            If i = Something Then
                 Call CompFe2.Set ( i )
                 Call CompFe2.Set ( b )
                 Call CompFe2..X ( )
            End if

End Function
```

Figure 4.23: Example of Alteration Relationship.

The next example illustrates a shared relationship implementation. In Figure 4.24 this example, the state is shared among the two features via a shared-state-full function A (). Function A () holds the state in a static variable and that state is used by FE_2. Since FE_2 and FE_1 use function A (), the two features are related; function A () can be some part of component comp FE_2. Finally, the code for FE_2 is actually moved into a method called X ().

```
Function X (int c)

        Declare Local Variable i
        Call A(0)

        i = Initialize
        i = A(1)

        IF c = SomeValue Then
           Code for Feature 1
           Use i
           More of Feature 1 Code
        Else
           Code for Feature 2
           Use i
           More of Feature 2 Code
        End if

End Function

Function A(int i)

        Static Variable K
        If i = 0 Then
          Calculation Code for K
          K = SomeValue
        Else
          Return K
        End if

End Function
```

```
Function X (int c)

        Declare Local Variable i
        Call CompFE2.A(0)


        i = Initialize
        i = CompFe2.A(1)

        IF c = SomeValue Then
           Code for Feature 1
           Use i
           More of Feature 1 Code
        Else
           Call CompoFe2.Set(i)
           Call CompFe2.X
        End if

End Function
```

Figure 4.24: Example of Shared Relationship.

Even though conflict and competition are solution domain concerns, they usually do not share a common function as far as implementation is concerned as shown in Figure 4.25 and Figure 4.26. As a result, these two types of direct feature relationships can be profiled easily with the code-profiler; however, evolution may require configuration level changes at a higher granularity. For example, a conflict relationship exists when a batch process is trying to change the status of certain records in the database while the GUI is running. Changing the status is, in fact, a feature but since it can be called from both batch and GUI there is a conflict. Since both these features are implemented in separate functions, the code-profiler will identify them individually based upon the test cases. The evolution of these two direct relationships (conflict and competition) is outside the scope of this dissertation. The understanding is that these methods are called at different times so there will not be a problem.

```
Function X ()
        Update Rows in certain table of a Database
        via a Batch Process
End Function

Function Y()
        Update Same Rows in certain table of a
        Database via a GUI Process
End Function
```

```
Function X ()
        CompFe2.X
End Function

Function Y()
        CompFe2.Y
End Function
```

Figure 4.25: Examples of Conflict Relationships.

```
Function X ()
        Trying to access a shared memory region
        at time t
End Function

Function Y()
        Trying to access the same shared memory
        region at time t (as function X)
End Function
```

```
Function X ()
        CompFe2.X

End Function

Function Y()
        CompFe2.X

End Function
```

Figure 4.26: Example of Compete Relationships.

4.4 Budget Analysis Model

In this section, we describe a simple model that allows the project manager to quickly calculate the net gain or loss due to the application evolution methodology. While there are several cost models such as COCOMO and others [15][17][109][130] that can be used, we show a simple model to track costs relevant to our methodology. These items can be integrated into other cost models as well. These costs are evaluated in Chapter 6 using our primary case study. Other cost savings are also possible so this list is not exhaustive. Note that original regression test-suites can be used to test feature-based as well as the CORE components. It is important to note that testing feature-based and CORE components are two separate processes. We suggest that CORE be integrated first and then tested, followed by the feature-based (and CORE) components. The elements of our cost models are as following:

Element	Description
Cost of Mapping Features and Test-Cases	Time taken by the software team to identify and map features and test cases.
Cost of identifying code using test cases and profiler	Time taken by the software team to run the code coverage tool to identify feature implementation.
Cost of Refactoring	Time taken to analyze heuristics and FIs.
Cost of Developing Components	Time taken to develop feature-based fine-grained components
Cost of Developing CORE Component	Time taken to create the shared reusable library
Cost of Configuration Management (CM)	Time taken to develop CM activities among product lines
Cost of Testing	Time taken to test feature-based and CORE components
Cost of Training and Documentation	Time taken to develop users guide and train other members of the software process team
Savings from Solving Feature Problems	Time saved from fixing the feature specific problems. It can be viewed as what would it cost in absence of the methodology
Savings from improved architecture (reduced global variables, more explicit communication and better understanding of features)	Time saved in training a new hire. This element is hard to measure because it is always implied. We were unable to measure it at AFS.
Savings in reusing Core	Time saved in re-development efforts in other product lines
Savings in reusing feature specific component	Time saved in re-development efforts in other product lines
Net Cost (+)/Savings (-)	**Sum of all costs and savings. Negative number means a profit.**

Table 4.17: Budget Analysis.

4.5 Formal Model

The feature model and the fine-grained component model are supported by a formal model that we now describe. We use Relational Calculus as the basis of our formal model that was introduced by Codd [34]. Refer to **Appendix D** for Relational Calculus Preliminaries.

A feature as described earlier is a group of individual requirements that describes a unit of functionality. We also established that regression test cases are the way these features are exercised. The feature model describes how FIs (functions, local data and global data) is associated with the features.

Researchers [5][125][121][99] have proposed several execution slice-based heuristics to identify code that is *uniquely* related to a given feature. Although code so identified provides an excellent starting point for program understanding and evolution, it is not sufficient to capture relationships such as SS, SSF, DF and DD (see Section 4.1.4). To capture these relationships, we need to identify functions and data that are shared among features. One approach is to use the union of the FI of related test cases to find a set of functions and data. Theoretically, we may need to use all test cases for a given legacy system and feature. In practice, this is often not necessary because the evolution methodology we describe suggests three ways to group related test cases (see Section 4.1.3.1).

The reason for using test cases with respect to the feature being examined is to avoid FIs that have nothing to do with this feature. If this is not possible (i.e., every input with respect to this feature also exercises some other feature), we need to subtract code that is uniquely used to implement the other features from the code identified by the union of such test cases. A simple example explanation is as follows. Suppose a feature (say FE_1) cannot be exercised without also having another feature FE_2 exercised. Also, assume that FE_2 can be exercised by itself. Under this situation, a way to find code used to implement FE_1 is to first find code used to implement FE_2 and FE_1, then subtract the code uniquely related to FE_2.

The formal model presented in this section forms the mathematical basis for the Feature and Fine-Grained Component Model discussed earlier. Regression test cases, feature, feature implementation and fine-grained components are represented using Relational Calculus and First Order Predicate Logic.

4.5.1 Data Model

Figure 4.27 illustrates the data model that will be used as the basis for formalism using relational algebra. The data model also provides the basis for the Evolution Manager Utility described in Section 4.6. The data model contains the information regarding the legacy system, the feature function relationship, the feature interactions and finally the component definition. The data model can be used to trace feature relationships, interactions, and component evolution of a legacy system.

The data model can be divided into four parts:

- **System Information Part**: This part consists of the following six tables; Legacy_System, Release, Feature, Test_Case_Feature_Map, Function_List and Test_Cases. The system information part describes the information about the legacy system's release. A release is a production version of the legacy system. This part of the data model reflects a legacy system with many releases. Each release may have many associated features, functions and test cases. A feature can be represented by one or more test cases. Table 4.18 provides more details on the specific tables and their relationships of system information part of the data model. The purpose of these tables are to capture information about the legacy system, the data can be entered manually into the database or import routines can bring the data from another system/sub-system.

- **Feature/Function Part**: This part of the data model stores the results from the profiler and is related to the system information. The two parts are related by Function_ID in the tables Function_List and Feature_Function_Map. The feature/function part of the data model consists of Test_Cases_to_Function, Feature_Function_Map, Function_To_Vars and Variable tables. Essentially, the test cases and the FI is determined by using profiler and Feature/Function mapping is stored in the Feature_Function_Map table. Information about variables and their location within the function is kept within the Function_To_Vars and Variable tables. Table 4.19 provides more details on the specific tables and their relationships of system information part of the data model. These tables are central to the collection of information for the methodology.
- **Feature Interaction Part**: This part of the data model contains information about feature interaction. It consists of Shared_Stateful, Dependent_Data, Dependent_Function and Shared_Stateless tables. This part is related to the feature/function part via the Execution_ID field

within the Feature_Function_Map and all the four tables listed above. This part of the data model is populated by the analyzing the feature interactions among features. The code profiler identifies the FI and data that is associated with a feature. This FI may call functions (SS, SSF, DD, and DF, see section 4.1.4 for more details) that may be part of other FI. The table Shared_Stateful is populated if FI calls an SSF that is part of another FI. Likewise, Shared_Stateless is populated if FI calls an SS that is part of another FI, and so on. Currently, these tables are manually populated in the database but code can developed to identify SS, SSF, DD and DF from the calling FIs to automate this process. Table 4.20 provides more details on the specific tables and their relationships of system information part of the data model.

- **Component Definition Part**: This part of the data model contains information regarding the component definition that is the result from applying the methodology. It consists of Component, Component_Interface, Component_Property_Set and Component_Property_Get tables. This part is related to the feature interaction part via the Shared_Stateful_ID, Dependent_Data_ID, Dependent_Function_ID and Shared_Stateless_ID. Component definition part stores the information regarding property get, property set and the feature interface. Table 4.21 provides more details on the specific tables and their relationships of component definition part of the data model.

There are several purpose of this data model. First, it provides an intuitive understanding of the evolution process and maps the methodology steps to the physical tables. Second, it provides the foundation of our formal model (see Section 4.5). Third, it provides the foundation for the Evolution Manager Utility (see Section 4.6) that can be used to track the evolution process of our methodology.

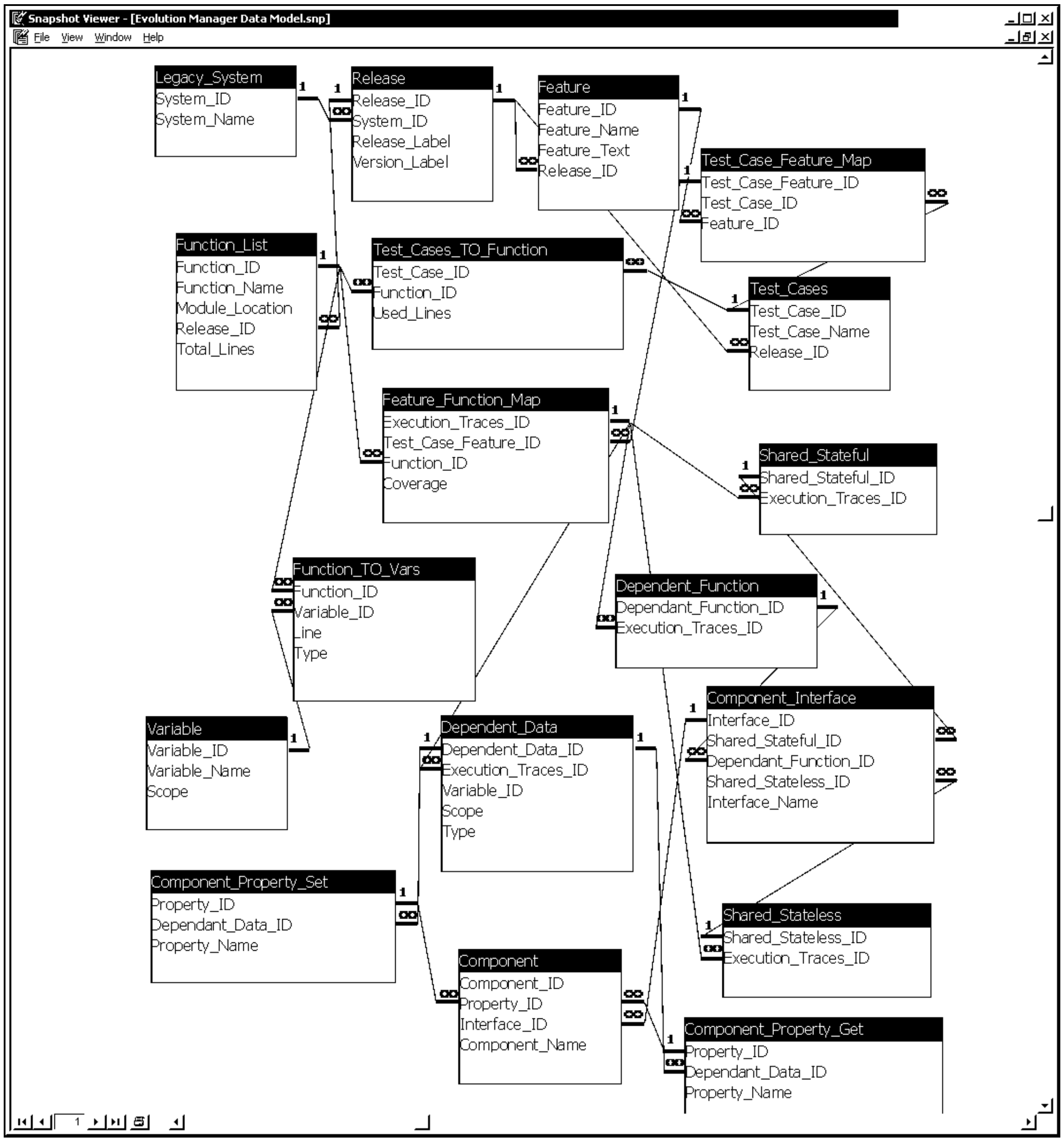


Figure 4.27: Data Model Used as Basis for Formalism.

Table	Related Table: Relationship	Keys	Relationship Type
Legacy_System	Legacy_System: Release	System_ID	One-To-Many
Release	Legacy_System: Release	System_ID	One-To-Many
	Release: Feature	Release_ID	One-To-Many
	Release: Test_Cases	Release_ID	One-To-Many
	Release: Function_List	Release_ID	One-To-Many
Feature	Feature: Test_Case_Feature_Map	Feature_ID	One-To-Many
	Release: Feature	Release_ID	One-To-Many
Test_Case_Feature_Map	Feature: Test_Case_Feature_Map	Feature_ID	One-To-Many
	Test_Cases: Test_Case_Feature_Map	Test_Case_ID	One-To-Many
	Test_Case_Feature_Map: Feature_Function_Map	Test_Case_Feature_ID	One-To-Many
Test_Cases	Test_Cases: Test_Case_Feature_Map	Test_Case_ID	One-To-Many
	Release: Test_Cases	Release_ID	One-To-Many
	Test_Cases: Test_Cases_TO_Function	Test_Case_ID	One-To-Many
Function_List	Function_List: Test_Cases_TO_Function	Function_ID	One-To-Many
	Function_List: Feature_Function_Map	Function_ID	One-To-Many
	Release: Function_List	Release_ID	One-To-Many
	Function_List: Function_TO_Vars	Function_ID	One-To-Many

Table 4.18: Data Model - System Information.

Table	Related Table: Relationship	Keys	Relationship Type
Feature_Function_Map	Feature_Function_Map: Shared_Stateless	Execution_Traces_ID	One-To-Many
	Feature_Function_Map: Shared_Stateful	Execution_Traces_ID	One-To-Many
	Feature_Function_Map: Dependent_Data	Execution_Traces_ID	One-To-Many
	Function_List: Feature_Function_Map	Function_ID	One-To-Many
	Feature_Function_Map: Dependent_Function	Execution_Traces_ID	One-To-Many
	Test_Case_Feature_Map: Feature_Function_Map	Test_Case_Feature_ID	One-To-Many
Function_TO_Vars	Variable: Function_TO_Vars	Variable_ID	One-To-Many
	Function_List: Function_TO_Vars	Function_ID	One-To-Many
Test_Cases_TO_Function	Function_List: Test_Cases_TO_Function	Function_ID	One-To-Many
	Test_Cases: Test_Cases_TO_Function	Test_Case_ID	One-To-Many
Variable	Variable: Function_TO_Vars	Variable_ID	One-To-Many

Table 4.19: Data Model - Feature/Function Part.

Table	Related Table: Relationship	Keys	Relationship Type
Dependent_Data	Feature_Function_Map: Dependent_Data	Execution_Traces_ID	One-To-Many
	Dependent_Data: Component_Property_S	Dependant_Data_ID	One-To-Many
	Dependent_Data: Component_Property_G	Dependant_Data_ID	One-To-Many
Dependent_Function	Dependent_Function: Component_Interface	Dependant_Function_ID	One-To-Many
	Feature_Function_Map: Dependent_Function	Execution_Traces_ID	One-To-Many
Shared_Stateful	Shared_Stateful: Component_Interface	Shared_Stateful_ID	One-To-Many
	Feature_Function_Map: Shared_Stateful	Execution_Traces_ID	One-To-Many
Shared_Stateless	Feature_Function_Map: Shared_Stateless	Execution_Traces_ID	One-To-Many
	Shared_Stateless: Component_Interface	Shared_Stateless_ID	One-To-Many

Table 4.20: Data Model - Feature Interaction Part.

Table	Related Table: Relationship	Keys	Relationship Type
Component	Component_Interface: Component	Interface_ID	One-To-Many
	Component_Property_G: Component	Property_ID	One-To-Many
	Component_Property_S: Component	Property_ID	One-To-Many
Component_Interface	Component_Interface: Component	Interface_ID	One-To-Many
	Shared_Stateful: Component_Interface	Shared_Stateful_ID	One-To-Many
	Shared_Stateless: Component_Interface	Shared_Stateless_ID	One-To-Many
	Dependent_Function: Component_Interface	Dependant_Function_ID	One-To-Many
Component_Property_Get	Component_Property_G: Component	Property_ID	One-To-Many
	Dependent_Data: Component_Property_G	Dependant_Data_ID	One-To-Many
Component_Property_Set	Component_Property_S: Component	Property_ID	One-To-Many
	Dependent_Data: Component_Property_S	Dependant_Data_ID	One-To-Many

Table 4.21: Data Model - Component Definition.

4.5.2 Preliminary Definition

Let:

LS be a Legacy System consists of functions and global data, LS = (F,G).

Define FE to be the set of all feature in LS and F is the set of all functions, $f_i \in F$. FE_i represents a specific feature. Let T be the set of regression test case that are part of LS, T = $\{t_1, t_2, t_3, t_4, \ldots\ldots t_n\}$, t_i refers to a specific test case within T.

The profiling of LS can determine which FIs are executed for any test case $t_i \in T$. Thus, we define a relation EXERCISES over $T \times F$ such that EXERCISES(t_i, f_i) is true if f_i is exercised by test case t_i.

We now define a relation that links test cases and features together. There are many different ways to view the features in FE but we are concerned with members of FE that can be represented by a subset of T. Thus we can define a relation REPRESENTS over $T \times FE$ such that REPRESENTS(t_i, FE_i) is true if test case t_i represents feature FE_i.

Minimal Constraint $\rightarrow \forall FE_i \exists t_j$ Such that REPRESENTS(t_j,FE_i)

Coverage Constraint $\rightarrow \forall t_i \exists FE_j$ Such that REPRESENTS(t_i,FE_j)

Non Pervasive Constraint $\rightarrow$ (Not $\exists$) $FE_i \forall t_j$ Such that REPRESENTS(t_j,FE_i), this means that no feature is tested by all test cases.

Let FI_i be a feature implementation as defined in the feature model and FE_i be a feature of LS that is exercised when K_i is executed, k_i is a set of test cases such that $k_i \subset T$. For example, $K_i = \{t_1, t_5, t_{11}\}$; note that $K_i \neq T$ by the non-pervasive constraint. We define the USES(f_i,f_j) relation over $F \times F$ that identifies when function f_i invokes function f_j. We are not ready to provide precise definition of the term feature implementation. Define FI_i as the implementation of FE_i in LS such that $FI_i = \{f \mid \forall\ t \in T, f \in F, \text{REPRESENTS}(t, FE_i) \wedge \text{EXERCISES}(t,f)\}$. Note that FI_i will be the way we refer to the implementation of FE_i.

4.5.3 Feature Interaction

Features interact because their underlying feature implementation overlap. Two features FE_i and FE_j interact functionally when $FI_i \cap FI_j \neq \emptyset$. Two features can also interact through data. If we define G to be set of global variables in LS, LOCALS (f_i) to be set of local variables within a function f_i, $D(f_i) = \{d \mid \text{USES}(f_i,d) \wedge (d \in \{G \cup \text{LOCALS}(f_i)\})\}$, and $\text{DATASCOPE}(FE_i) = \{ d \in D(f_i) \wedge f \in FI_i\}$ then the two features interact through data when $\text{DATASCOPE}(FI_i) \cap \text{DATASCOPE}(FI_j) \neq \emptyset$. Note that data interaction model is not powerful to capture alias or pointers to data (such as a SQL statement). We chose to ignore such details at this time since our methodology is capable of identifying the data that is required to create components. It is also useful to discuss the concept of neighboring features, that is, features that share their implementation with the target (feature to be evolved) feature. $NEIGHBOR(FE_i) = \{ FE_j \mid FE_j \neq FE_i \wedge (FI_i \cap FI_j \neq \emptyset)\}$

4.5.4 Classifying Functions

A function is Shared Stateless (SS) when $D(f) = \{\emptyset \wedge \exists\ FE_i, FE_j \mid f \in FI_i \cap f \in FI_j\}$

A function is Shared State-Full (SSF) when $D(f) = \{\emptyset \wedge \exists\ FE_i, FE_j \mid \exists\ d \in D(f) \wedge d \in DATASCOPE(FE_i) \wedge d \in DATASCOPE(FE_j)\}$

A function is Dependent Data (DD) when $\exists\ FE_i, FE_j, f_x, d, f_y \mid d \in DATASCOPE(FE_i) \cap DATASCOPE(FE_j) \wedge USES(f_x,d) \wedge f_x \in FI_i) \wedge USES(f_y,d) \wedge f_y \in FI_j \wedge f_x \neq f_y.$

A function is Dependent Function (DF) when $\exists\ FE_i, FE_j, f_x, f_y, f_z \mid f_x \in FI_i) \wedge f_y \in FI_j) \wedge USES(f_x, f_y) \wedge USES(f_y, f_z) \wedge f_x \neq f_y.$

4.5.5 Identifying Interactions within a Functions

We need to capture the interactions within a function between features. We define TRACE(t,f) = {N} where N is a natural number representing the lines of code executed within function f when test case t is exercised. The function definition can be defined to be $F_{Def} = N \times f$. Given that there may be a second feature whose implementation may overlap within f, we need to separate the code within the function f. We define $SCOPE(FE_i,f) = \forall\, t_i \bigcup TRACE(t_i,f) \wedge f \in FI_{i..}$

This allows us to define the code that is exclusive to the feature as $EXCLUSIVE(FE_i,f) = SCOPE(FE_i,f) - \bigcup SCOPE(FE_j,f)\ \forall\, FE_j \neq FE_{i.}$

4.5.6 CORE

We assume that all functions that belong to CORE are stateless and that they are exercised by all test cases. We define $CORE = \{f \in FI_i | \forall\, t \in T, EXERCISES(t,f) \wedge SS(t,f)\}$.

4.5.7 Threshold

As defined in Section 4.1.3.3 that Threshold consists of FI, K and C where FI is the number of functions (note that this is same as feature implementation), K is the number of neighboring features and C is the average coverage of a FE across all FIs.

Generalized Feature Set (GFS) for FE_i for a given function f_x= *Cardinality*($\forall T_i$ | REPRESENTS(T_i, FE_i) $\wedge$ EXERCISES(T_i,f_x)). It means set of features implemented in a given function.

When |GFS| = 1 it indicates that the function implements only one features

|GFS| > X represents *Evolution Threshhold*. It means that if a function implements more than X features in it, then it is probably not a good candidate for our methodology unless the function belongs to CORE.

Threshold can also be defined as a combination of (|FI|, |K|, C), where |FI| = *Cardinality*($\forall FI_i$, FE_i | FI_i), |K| = *Cardinality*($\forall FI_i$,FE_i |NEIGHBOR(FE_i)) and C = (*Cardinality*($\forall FE_i$ | EXCLUSIVE(FE_i,f))/|FI|)*100.

Figure 4.27 illustrates the data model that can be used as the basis for formalism using relational algebra and first order logic.

4.5.8 Summary

We presented a formal representation of our methodology in this section. Using relational calculus and first-order-logic we defined Feature Implementation, Feature Interactions, Classification of Functions into SS, DD, DF and SSF, and CORE. Most important aspect of our formal model is the fact that it is based on relational data model described in Section 4.5.1. We identified a weakness in representation of data when the data is a pointer or an alias (such as a SQL statement). We also represented important elements of our methodology such as

GFS, Threshold and Neighboring features. The next section describes the evolution manager utility that is based on the data model and it proved to be a useful tool in our case study.

4.6 Evolution Manager Utility

We developed a utility called *evolution manager* based upon the data model discussed in Section 4.5. Figure 4.28 shows an overview of evolution manager functions. The key features of evolution manager utility are:

- Feature function relationships based upon test case and features, and test case and functions
- Feature function relationship in terms of coverage percentage
- Exclusive coverage of a feature within a function
- Calculate Threshold T(FI,K,C)
- Variable usage (set or use) by a feature within a function
- Feature implementation in terms of which lines of code and variables implement the feature
- Several tracking reports such as feature lists, function lists, or features within a release etc

The following results from the profiler are imported into the evolution manager:

- Line(s) of code executed in each function by a test case
- Local and global Variables used and updated in each function

In addition to the information imported from the profiler, the evolution manager's data model accounts for following data (user input):

- Test cases used in each release, provided by testers
- Mapping of test case and features, provided by testers

Using SQL statements and matrix calculations, the evolution manager generates reports that allow us to identify feature implementation. The feature implementation is then used to refactor the code to create component. The utility does not refactor the code but provides tracking and identification of feature implementation based upon information discussed above. Among the key reports that the utility provides are feature/function mapping, feature exclusive lines of code in a function, threshold and recommendation for component's properties. The evolution manager utility can be used to help automate the methodology by taking the following steps:

- Populate the system information part of the data model either by importing data from another system or by manually entering the data through some simple user interfaces screens.
- Develop logic to identify feature/function interaction by identifying SS, SSF, DD and DF; note that this will involve programming language-dependent logic. This will result in populating the feature interaction part.
- Develop logic to populate the component definition part, this can be achieved by identifying which variables are set/used by FI.

Appendix G provides detail on the evolution manager utility and implementation of the example discussed in Chapter 5.

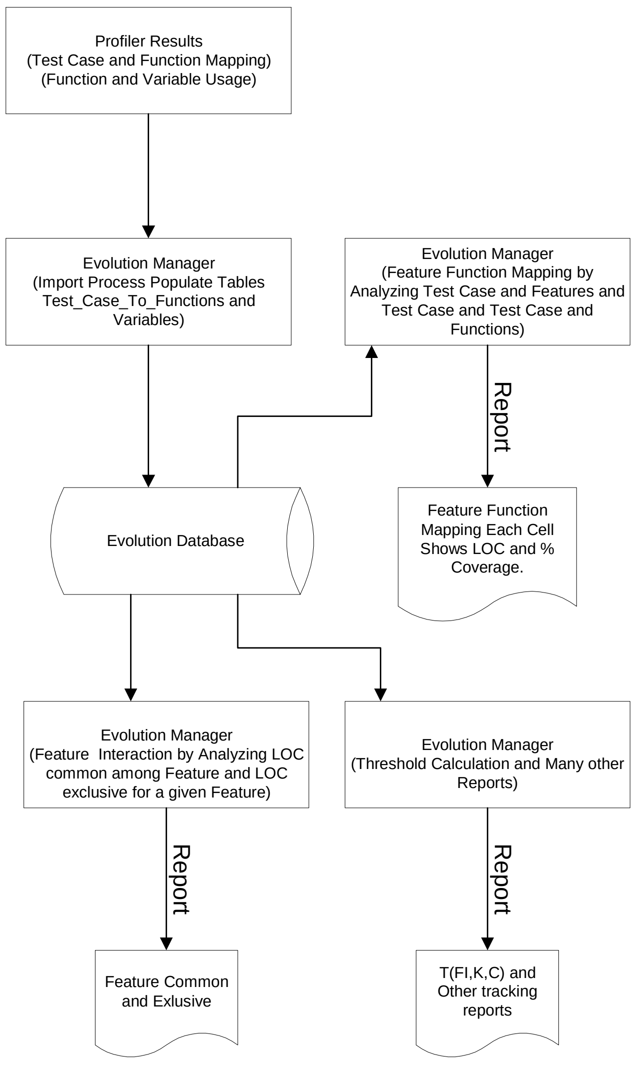

Profiler Results
(Test Case and Function Mapping)
(Function and Variable Usage)
Evolution Manager
(Import Process Populate Tables Test_Case_To_Functions and Variables)
Evolution Manager
(Feature Function Mapping by Analyzing Test Case and Features and Test Case and Test Case and Functions)
Report
Evolution Database
Feature Function Mapping Each Cell Shows LOC and % Coverage.
Evolution Manager
(Feature Interaction by Analyzing LOC common among Feature and LOC exclusive for a given Feature)
Evolution Manager
(Threshold Calculation and Many other Reports)
Report
Report
Feature Common and Exlusive
T(FI,K,C) and Other tracking reports

Figure 4.28: Evolution Manager Utility.

4.7 Summary

Four models were discussed in this chapter, namely the feature model, fine-grained component model, the budget analysis model and the formal model. The feature model describes our definition and understanding of features. It address the feature interaction problem by considering relationships among features. We also discussed heuristics using feature implementation. The fine-grained component model describes our definition and understanding of components and component model. Both feature and fine-grained component models are used to evolve feature implementations into reusable components. We also described a simple budget analysis model and items that should be tracked so the methodology can be verified. Finally, we provided theoretical foundation of our feature and fine-grained component model in formal model.

5 A Simple Example

To illustrate indirect feature relationships and evolution methodology; we have extended a small example that first appeared in [5]. We extended the example to show how our evolution methodology can be used to encapsulate a group of related features. The purpose of this example is to show how our evolution methodology can be used to trace source code associated with test cases; how the traced code can be encapsulated into components; and finally, how these components can be reused.

As described in [5] and informally observed, an Automatic Teller Machine's (ATM) operational requirements are shown in Table 5.1.

ATM Operational Requirements	
1.	A customer must be automatically prompted for a Personal Identification Number (PIN).
2.	After the input of a PIN, the customer must be offered a set of operations: make a deposit, make a withdrawal, or check one account balance.
3.	After an operation has completed, the customer must have the opportunity to start another operation.
4.	At any point of an operation, the customer must be able to cancel the current operation and be asked whether to continue with another operation.
5.	After the operation is chosen, the customer must select the account on which to perform the operation: checking or savings.
6.	In the case of a withdrawal, the customer must enter a positive number that represents the amount to withdraw from the selected account. Furthermore, if the withdrawal is done on the checking account then the amount must be less than or equal to $300.
7.	In the case of a deposit, the customer must be able to insert bills of $5, $10, $20, $50 or $100 into the ATM. The corresponding account must be credited.
8.	In the case of a balance operation, the balance of the corresponding account must be displayed on the screen.
9.	When the series of operations is terminated, the customer must decide whether a receipt should be printed. Given a positive response, a receipt with the balance information of all accounts that have been affected during the transaction should be printed.

Table 5.1: ATM Operatioal Requirements.

This ATM function was implemented in Visual Basic (See Figure 5.1). In addition, a feature analysis and test-case analysis was also performed to map feature and test cases. Note that details of functions such as Make_Deposit () and Make_Withdrawal () are omitted for space reasons. Line numbers are used for reference as they will be utilized when profiling the code using test cases. An interesting observation regarding the ATM example is that there is an indirect composition relationship because an Account is comprised of Checking and Savings.

```
Global PIN
Global Account_Choice
Global Amount
Global Checking_Flag
Global Savings_Flag
Global Receipt_Choice
Global Customer_Rec

Function ATM()
1.      Print ("Enter PIN")
2.      Read(PIN)
3.      Customer_Rec = Get_Customer(PIN)
4.      Do
5.              Clear_Screen
6.              Print (1. Deposit, 2. Withdraw, 3. Balance, 0 to End)
7.              Read (Code)
8.              IF Code > 0 Then Print (1. Checking, 2. Savings)
9.              IF Code > 0 Then Read (Account_Choice)
10.             IF Code = 1 Then
11.                  Amount = Get_Money(Customer_Rec)
12.                   IF Amount > 0 And Account_Choice = 1 Then
13.                       Make_Deposit(Customer_Rec, Amount, "C")
14.                       Checking_Flag = True
15.                   ELSEIF Amount > 0 And Account_Choice = 2 Then
16.                       Make_Deposit(Customer_Rec, Amount, "S")
17.                       Savings_Flag = True
18.                   END IF
19.             ELSEIF Code = 2 Then
20.                    Clear_Screen
21.                    Print ("Enter Amount")
22.                    Read(Amount)
23.                    IF Account_Choice = 1 THEN
24.                            IF Amount <= 300 AND Amount >0 THEN
25.                                Make_Withdrawal(Customer_Rec, Amount, "C")
26.                                 Checking_Flag = True
27.                            ELSEIF Amount >300 THEN
28.                                 Print ("Error: Cannot withdraw more than 300")
29.                            END IF
30.                     ELSEIF Account_Choice = 2 THEN
31.                             Make_Withdrawal(Customer_Rec, Amount, "S")
32.                              Savings_Flag = True
33.                      END IF
34.             ELSEIF Code = 3 Then
35.                      IF Account_Choice = 1 Then
36.                          Display_Balance(Customer_Rec,"C")
37.                       ELSEIF Account_Choice = 2 Then
38.                          Display_Balance(Customer_Rec,"S")
39.                      END IF
40.             END IF
41.     While (Code is NOT Equal to 0)
42.     Print ("Do you want a Receipt?, 1. Yes, 2. No")
43.     Read(Receipt_choice)
44.     If Receipt_Choice = 1 Then
45.         IF Checking_Flag THEN
46.             Print_INFO(Customer_Rec,"C")
47.         ELSEIF Savings_Flag THEN
48.             Print_INFO(Customer_Rec,"S")
49.          END IF
50.     ENDIF
51.     Eject_Card()
End Function
```

Figure 5.1: ATM Function Implemented in VB.

There are several features and sub-features in this example. They are summarized in Figure 5.2:

FE1: Customer Session
FE2: Enter PIN
FE3: ATM Operations
FE4: End of ATM
FE5: Receipt (Print)
FE6: Get Receipt
FE7: Skip Receipt
FE8: Withdrawal
FE9: Checking Withdrawal
FE10: Savings Withdrawal
FE11: Abort Withdrawal
FE12: Deposit
FE13: Checking Deposit
FE14: Savings Deposit
FE15: Abort Deposit
FE16: Verify Balance
FE17: Checking Balance
FE18: Savings Balance
FE19: Abort Balance
FE20:Amount to Withdraw
FE21: Amount to Deposit
FE22:Verify Limits
FE23: Abort Withdrawal
FE24: Accept Money
FE25 Enter Digit

Figure 5.2: Summary of Features in ATM Function.

Again, the deposit feature comprises a deposit either in the checking account or the savings account. Similarly, withdrawal can also be viewed as an indirect relationship of type composition. The next step is to analyze test cases that exercise the features in ATM sub-system.

Test Case	Content	Mapped Features
t1	123 Deposit Savings 100 Finished Finished No	FE1,FE2,FE3,FE4,FE5,FE7,FE12,FE14,FE21,FE22,FE23,FE23,FE25
t2	123 Deposit Checkings 100 Finished Finished No	FE1,FE2,FE3,FE4,FE5,FE7,FE12,FE13,FE21,FE22,FE23,FE23,FE25
t3	123 Withdraw Checkings 20 Finished Finished No	FE1,FE2,FE3,FE4,FE5,FE7,FE8,FE9,FE20,FE25
t4	123 Withdraw Checkings 500 Finished Finished No	FE1,FE2,FE3,FE4,FE5,FE7,FE8,FE9,FE20,FE25
t5	123 Withdraw Savings 200 Finished Finished Yes	FE1,FE2,FE3,FE4,FE5,FE6,FE8,FE10,FE20,FE25
t6	123 Balance Checking Finished Yes	FE1,FE2,FE3,FE4,FE5,FE6,FE16,FE25
t7	123 Balance Savings Finished Yes	FE1,FE2,FE3,FE4,FE5,FE6,FE18,FE25
t8	123 Deposit Savings 50 Finished Finished No	FE1,FE2,FE3,FE4,FE5,FE7,FE12,FE14,FE21,FE22,FE23,FE23,FE25

Table 5.2: Summary of Test Cases and Features in ATM Sub-system.

Note that not all features are worth analyzing for the purpose of creating components. The main features that are of interest are Deposit, Withdrawal and Balance inquiries of checking and the savings account. The eight test cases are a complete set of test cases that exercise the three features of interest. For example, test case number 5 will withdraw $200 from savings account and will generate a receipt before giving the card back to the customer.

Before the components can be created, code must be located and features must be analyzed and prioritized. Some features are better candidates than others as far as their evolution is concerned. While feature analysis and prioritization can identify the reasons to evolve features, their implementation in terms of functions and variables is indeed very important. One such technique, described in this dissertation is to identify code associated with a feature is to run source-code profiler with the test cases. In this simple example, the following code was revealed after running all eight test cases (Table 5.3):

Test Case	Content	Lines of Code
t1	123 Deposit Savings 100 End Finished No	1-9,10,11,12,15-18,41-44,51
t2	123 Deposit Checkings 100 End Finished No	1-9,10,11,12,13-14,41-44,45-46,51
t3	123 Withdraw Checkings 20 End Finished No	1-9 10,19,20,21,22,23,24,25,26,30,33,41,42,43,44,51
t4	123 Withdraw Checkings 500 End Finished No	1-9, 10,19,20,21,22,23,27,28,29,30,33,41,42,43,44,51
t5	123 Withdraw Savings 200 End Finished Yes	1-9, 10,19,20,21,22,23,30,31,32,33,41,42,43,44,45,47,48, 49,50,51
t6	123 Balance Checking Finsihed Yes	1-9,34-36,41,42,43,44,45,46,50,51
t7	123 Balance Savings Finsihed Yes	1-9,34-36,41,42,43,44,45,47,48,49,50,51
t8	123 Deposit Savings 50 Finsihed Finished Yes	1-9,10,11,12,15-18,41-44,47-51

Table 5.3: Profiler Results on ATM Test Cases.

Once the code has been identified, it can then be refactored and evolved. There are several refactoring techniques that have been described in the literature. The purpose of this dissertation is not to describe the refactoring techniques but to use them in the evolution methodology. Note that there are functions that the ATM function calls with all the test cases. Since these functions are common to all test cases and features, they can be collected and bundled into a library called CORE, as shown in Figure 5.3:

```
Function PRINT(s)
End Function

Function Read(s)
End Function

Function Get_Customer(s)
End Function

Function Clear_Screen
End Function

Function Print_Info(Customer_Rec,s)
End Function

Function Send_Back_Card
End Function
```

Figure 5.3: CORE Library Functions.

Assuming that the deposit feature needs evolution, the following code snippet outlines how the code is identified, evolved into a component and called from the legacy function ATM (). Note that CORE is also shown as a part of the incremental evolution. The Deposit component has three set properties, namely Customer_Rec, Amount and Account Choice. The Customer_Rec is used to access the customer record, Amount property determines how much to deposit and Account Choice tells the component into which account (Checking or Savings) the deposit should be made. Note that the Account Choice variable can further be refactored into a CONSTANT variable for explicitness. The return values of the deposit component are represented by the get property. There are three get properties that the caller of the deposit component can use, Balance, Checking Flag, and Savings Flag. The Balance property tells the caller what the balance is after making a deposit to either account. The flag properties simply tell the caller which account was affected. The deposit functionality is encapsulated in the function deposit() of component deposit as shown below. The calling of the component is also shown in Figure 5.4 and

Figure 5.5.

```
Function ATM()
        CORE.Print ("Enter PIN")
        CORE.Read(PIN)
        Customer_Rec = CORE.Get_Customer(PIN)
        Do
                CORE.Clear_Screen
                CORE.Print (1. Deposit, 2. Withdraw, 3. Balance, 0 to End)
                CORE.Read (Code)
                IF Code > 0 Then CORE.Print (1. Checking, 2. Savings, 3. Cancel)
                IF Code > 0 Then CORE.Read (Account_Choice)
                IF Code = 1 Then
                  CORE.Clear_Screen
                  CORE.Print("Enter Amount")
                   Amount=CORE.Read(Amount)
                  ComponentDeposit.Amount=Amount
                  ComponentDeposit.PIN = PIN
                  ComponentDeposit.Account_Choice = Account_Choice
                  ComponentDeposit.Deposit
                  Balance = ComponentDeposit.Balance
                  Checking_Flag = ComponentDeposit.Checking_Flag
                  Savings_Flag = ComponentDeposit.Savings_Flag
                ELSEIF Code = 2 Then
                   Clear_Screen
                   Print ("Enter Amount")
                   Read(Amount)
                   IF Account_Choice = 1 THEN
                         IF Amount <= 300 AND Amount >0 THEN
                           Make_Withdrawal(Customer_Rec, Amount, "C")
                        Checking_Flag = True
                         ELSEIF Amount >300 THEN
                           Print ("Error: Cannot withdraw more than 300")
                         END IF
                   ELSEIF Account_Choice = 2 THEN
                         Make_Withdrawal(Customer_Rec, Amount, "S")
                               Savings_Flag = True
                    END IF
                ELSEIF Code = 3 Then
                    IF Account_Choice = 1 Then
                      Display_Balance(Customer_Rec,"C")
                 ELSEIF Account_Choice = 2 Then
                      Display_Balance(Customer_Rec,"S")
                    END IF
                END IF
        While (Code is NOT Equal to 0)
        CORE.Print ("Do you want a Receipt?, 1. Yes, 2. No")
        CORE.Read(Receipt_Choice)
        If Receipt_Choice = 1 Then
          IF Checking_Flag THEN
            CORE.Print_INFO(Customer_Rec,"C")
          ELSEIF Savings_Flag THEN
            CORE.Print_INFO(Customer_Rec,"S")
        END IF
        ENDIF
        CORE.Eject_Card()
End Function
```

Figure 5.4: Modified ATM using Deposit Component.

```
Property Set PIN
Property Set Account_Choice
Property Set Amount

Property Get Balance
Property Get Checking_Flag
Property Get Savings_Flag

Private  Function
Make_Deposit(Customer_Rec,Amount,Account_Type)
Balance is Updated here
End Function

Private Function Get_Money(Customer_Rec)
End Function

Public Function Deposit()
            Customer_Record = CORE.Get_Customer(PIN)
            IF Amount > 0 And Account_Choice = 1 Then
                Make_Deposit(Customer_Record, Amount, "C")
                Checking_Flag = True
             ELSEIF Amount > 0 And Account_Choice = 2 Then
                Make_Deposit(Customer_Record Amount, "S")
                Savings_Flag = True
            END IF
End Function
```

Figure 5.5: Deposit Component.

To show the power of incremental evolution methodology, Withdrawal and Display Balance is evolved. Figure 5.6 shows what the evolved function and the new components look like. Note that properties (get and set) are used to pass the global variables. The public function provides the interface to the caller. The private functions encapsulate the withdrawal functionality.

```
Function ATM()

        CORE.Print ("Enter PIN")
        CORE.Read(PIN)
        Customer_Rec = CORE.Get_Customer(PIN)
        Do
                CORE.Clear_Screen
                CORE.Print (1. Deposit,2.Withdraw,3.Balance, 0 to End)
                CORE.Read (Code)
                CORE.Print (1. Checking, 2. Savings, 3. Cancel)
                CORE.Read (Account_Choice)
                IF Code = 1 Then
                  CORE.Clear_Screen
                  CORE.Print ("Enter Amount")
                  Amount=CORE.Read(Amount)
                  CompDeposit.Amount = Amount
                  CompDeposit.PIN = PIN
                  CompDeposit.Account_Choice = Account_Choice
                  CompDeposit.Deposit
                  Balance = CompDeposit.Balance
                  Checking_Flag = CompDeposit.Checking_Flag
                  Savings_Flag = CompDeposit.Savings_Flag
                ELSEIF Code = 2 Then
                  CORE.Clear_Screen
                  CORE.Print ("Enter Amount")
                  Amount=CORE.Read(Amount)
                  CompWithdrawal.Amount = Amount
                  CompWithdrawal.Max_Amount = 300
                  CompWithdrawal.PIN = PIN
                  CompWithdrawal.Account_Choice = Account_Choice
                  CompWithdrawal.Withdrawal
                  Balance = CompWithdrwal.Balance
                  Checking_Flag = CompWithdrawal.Checking_Flag
                  Savings_Flag = CompWithdrawal.Savings_Flag
                ELSEIF Code = 3 Then
                    CompShowBalance.Pin = PIN
                    CompShowBalance.Account_Choice
                    Comp.ShowBalance
                END IF
        While (Code is NOT Equal to 0)
        CORE.Print ("Do you want a Receipt?, 1. Yes, 2. No")
        CORE.Read(Receipt_Choice)
        If Receipt_Choice = 1 Then
          IF Checking_Flag THEN
            CORE.Print_INFO(Customer_Rec,"C")
          ELSEIF Savings_Flag THEN
            CORE.Print_INFO(Customer_Rec,"S")
            END IF
        ENDIF
        CORE.Eject_Card()
End Fucntion
```

```
Property Set PIN
Property Set Account_Choice
Property Set Amount
Propert Set Max_Amount

Property Get Balance
Property Get Checking_Flag
Property Get Savings_Flag

Private  Function
Make_WIthdrawal(Customer_Rec,Amount,Account_Type)
Balance is Updated here
End Function

Private Function Get_Money(PIN)
End Function

Public Function Withdrawal()
        Customer_Rec = Core.Get_Customer(PIN)
         IF Account_Choice = 1 THEN
           IF Amount <= MAX_Amount AND Amount >0 THEN
             Make_Withdrawal(Customer_Rec, Amount, "C")
             Checking_Flag = True
           ELSEIF Amount >MAX_Amount THEN
              Core.Print ("Error: Cannot withdraw more than
MAX_Amount
           END IF
        ELSEIF Account_Choice = 2 THEN
           Make_Withdrawal(Customer_Rec, Amount, "S")
           Savings_Flag = True
        END IF
End Function
```

```
Property Set PIN
Private Property Set Account_Choice
Function Display_Balance(Customer_Record,Account_Type)
End Function

Public Function ShowBalance()
        Customer_Rec = Core.Get_Customer(PIN)
        IF Account_Choice = 1 Then
          Display_Balance(Customer_Rec,"C")
        ELSEIF Account_Choice = 2 Then
           Display_Balance(Customer_Rec,"S")
         END IF
End Function
```

Figure 5.6: ATM Function, Withdrawal and Show Balance Components.

The last and final part of the example shows how the components can be used to add new functionality in the old legacy ATM function. In addition to these components being reused, they also open door for potential use in the platforms such as Wire Transfer Application or Internet Banking. Note that the new transfer component is created by integrating the deposit and the withdrawal components Figure 5.7.

```
Function ATM_Evolved()

          CORE.Print ("Enter PIN")
          CORE.Read(PIN)
          Customer_Rec = CORE.Get_Customer(PIN)
          Do
                   CORE.Clear_Screen
                   CORE.Print (1. Deposit,2.Withdraw,3.Balance 4.Transfer, 0 to End)
                   CORE.Read (Code)
                   CORE.Print (1. Checking, 2. Savings, 3. Cancel)
                   CORE.Read (Account_Choice)
                   IF Code = 1 Then
                      CORE.Clear_Screen
                      CORE.Print ("Enter Amount")
                      Amount=CORE.Read(Amount)
                      CompDeposit.Amount = Amount
                      CompDeposit.PIN = PIN
                      CompDeposit.Account_Choice = Account_Choice
                      CompDeposit.Deposit
                      Balance = CompDeposit.Balance
                      Checking_Flag = CompDeposit.Checking_Flag
                      Savings_Flag = CompDeposit.Savings_Flag
                   ELSEIF Code = 2 Then
                      CORE.Clear_Screen
                      CORE.Print ("Enter Amount")
                      Amount=CORE.Read(Amount)
                      CompWithdrawal.Amount = Amount
                      CompWithdrawal.Max_Amount = 300
                      CompWithdrawal.PIN = PIN
                      CompWithdrawal.Account_Choice = Account_Choice
                      CompWithdrawal.Withdrawal
                      Balance = CompWithdrawal.Balance
                      Checking_Flag = CompWithdrawal.Checking_Flag
                      Savings_Flag = CompWithdrawal.Savings_Flag
                   ELSEIF Code = 3 Then
                        CompShowBalance.Pin = PIN
                        CompShowBalance.Account_Choice
                        Comp.ShowBalance
                   ELSEIF Code = 4 Then
                         CORE.Print (1. Checking, 2. Savings)
                         CORE.Read (Destination_Account_Choice)
                         CompTransfer.PIN = PIN
                         CompTransfer.Amount = Amount
 Transfe                 CompTransfer.Source_Account_Choice = Account_Choice
 --                      CompTransfer.Destination_Account_Choice =
                             Destination_Account_Choice
                         CompTransfer.Transfer
                          If Destination_Account_Choice = 1  Or Account_Choice = 2
                             Checking_Flag = True
                          End if
                          If Destination_Account_Choice = 2  Or Account_Choice = 2
                                 Savings_Flag = True
                          End if
                   END IF
          While (Code is NOT Equal to 0)
          CORE.Print ("Do you want a Ticket?, Yes, No")
          CORE.Read(Ticket_choice)
          CompTicket.PIN = PIN
          CompTicket.Ticket_Choice = Ticket_Choice
          CompTicket.Ticket(Checking_Flag,Savings_Flag)
          CORE.Send_Back_Card()
End Function
```

```
Property Set PIN
Property Set Source_Account_Choice
Property Set Destination_Account_Choice
Property Set Amount

Property Get Balance
Property Get Checking_Flag
Property Get Savings_Flag

Public Function Transfer()
          IF Amount > 0
             CompWithdrawal.PIN = PIN
             CompDeposit.PIN = PIN
             CompWithdrawal.Amount = Amount
             CompDeposit.Amount = Amount
             CompWithdrawal.Account_Choice =
                     Source_Acount_Choice
             CompDeposit.Account_Choice =
             Destination_Account_Choice
             CompWithdrawal.Withdrawal ( )
             CompDeposit.Deposit ( )
             CompBalance.PIN = PIN
             CompBalance.Account_Choice=Source_Account_Choice
             CompBalance.ShowBalance
             CompBalance.Account_Choice=
                     Destination_Account_Choice
             CompBalance.ShowBalance
          END IF
End Function
```

Figure 5.7: Integrating Withdrawal and Deposit into Transfer Component.

6 Case Study

We applied the ten-step methodology outlined in Section 2.1 to the Master System (AMS), a product of American Financial Systems (AFS). AFS is a 60-person software firm that develops software for the corporate-owned life insurance market. AFS has developed AMS over the past 14 years to integrate life insurance and executive benefits using mathematical and financial modeling. AMS was first developed using Microsoft BASIC. Over the years, Microsoft has evolved BASIC into the more modern programming language, Visual Basic (VB). AFS ensured that the latest Microsoft compiler technology was used with each successive version of AMS. AMS is typical of long-lived software systems in that it has evolved from its original DOS version to a more modern Windows version. Appendix A lists the most important features of AMS. Appendix B describes the overall architecture of AMS. To illustrate the results of our methodology, we focused on the *Input Processing* functionality of AMS. *Input Processing* validates and prepares data from user inputs (also called items) so AMS can perform complex calculations to generate various reports. To an end-user, *Input Processing* has two purposes, *Suppression* and *Error Processing.* *Suppression* is a feature that either shows or hides an item in the user interface based upon the value of another item. *Error Processing* is a feature that validates item values.

There are 450 items in AMS and many of them are interdependent. Upon closer examination of *Input Processing*, we found that AMS also makes several *Assignments* (user input is stored as strings and is later assigned to types such as Integer, Single, Double, or Array). While *Assignments* are a hidden feature to the end-user, developers must naturally consider all three features when evolving the *Input Processing* of AMS.

The AMS data model for *Input Processing* is a hierarchy of plan, employee, and policy level information. A plan can have many employees and an employee can have many life insurance policies. A database stores a *Master File Table* that contains the 450 *plan items* that constitute a plan. Individual *employee items* are stored in a *Census File Table* and can vary for each employee in the plan. The *Census File Table* is associated with the *Master File Table*. For example, a plan with 3 employees might store all common information in the *Master File Table*, while storing each employee's age in the *Census File Table.* About 75% of the *plan items* can vary from employee to employee. An AMS test case is created from the combination of *Master File* and *Census File* data. AFS maintains a regression test suite (Appendix C) of nearly 450 test cases with an average size of 10 employees per test case. Running all regression tests executes AMS nearly 7,500 times. AMS provides a batch facility for executing regression tests and storing output to a text file.

The interdependencies among plan items are quite complex. For example, the value of the `retirement age` item for an individual cannot be less than the `policy issue age` item; *Input Processing* must enforce this constraint when either value changes. In addition, if the `policy issue age` item is greater than 75 then other items should be suppressed because certain policies may not be issued to persons older than 45 in some states. There are numerous, more complicated interdependencies within AMS items too detailed to discuss here. When a user-input in an item invalidates a constraint, AMS must display a message indicating the specific problem (note that suppressed items are not involved in *Error Processing*).

6.1 Evolution reasons

After a series of discussions with AMS project managers, marketers, testers, and key end-users, we found three reasons to evolve *Input Processing*.

6.1.1 AMS occasionally freezes during *Input Processing*

Many *plan items* are interdependent and so is their shared error-processing code. For example, **Item 9** assigns certain key variables whose value will determine whether **Item 16** is valid. In the code fragment validating values for **Item 9**, shown in Figure 6.1, global variable `nItem` is set to 16 and `Process_Items` is called to check for errors in the assignment of the item identified by `nItem` (**Item 16)**. **Item 16's** code section (not shown) sets a global error flag, `nError_F`, to indicate whether **Item 16** has a problem, which in turn means **Item 9** is not ready.

It is easy for developers to forget to reset the value of `nItem` back to the value of the calling Item number (in this case **Item 9**) resulting in an unbounded recursion that freezes the system during user input.

```
    nItem = 16
    call Process_Items
    nItem = 9
    If nError_F = 1 then
          Set Up Error Variables
    End if
```

Figure 6.1: Fragment for Validating Values for Item 9.

6.1.2 The cost of adding a new item into *Input Processing* is high

AFS developers required an average of three days to add just a single item because of implicit communication via global variables and the spaghetti-like calling process of the dependent items. Developers adding a new plan item must add a field to the database tables and update the data dictionary. Then it is necessary to code the complex logic of item dependence across the three features, namely, *Assignments*, *Error Processing*, and *Suppression*. Developers must identify the list of items that need to be suppressed based upon the input value of the new item and any errors must be generated. When adding an item, the processing of key global variables would often change, causing unexpected side effects. For example, incorrectly setting the value of `nItem` brought back errors that were previously fixed. Adding new items would often require unrelated items to be suppressed since the *Suppression* and *Error Processing* features are dependent on the *Assignments* feature.

6.1.3 The lack of code reuse between the desktop and web version of AMS

Since the web-based version of AMS required similar logical processing of plan items, AFS wanted to extract a reusable component from the legacy system to use within both systems. AFS wanted to avoid the costs of maintaining two divergent code bases, so solving this problem proved to be the greatest motivation for this evolution effort.

6.2 Identify feature(s) with problems

The testers test the *Suppression* and *Error Processing* of each item by inputting the valid entries for the particular item. Due to interdependencies, test cases are designed to test the combined effect of items. For example **Item 9** may have a valid input of 1 and 2, the result (*Suppression* and *Error Processing*) of inputting 1 may be different than inputting 2. End-user views *Suppression* and *Error Processing* of each item (and some valid combinations) when inputting values in the item fields. Each item and the valid combinations of items can be viewed as a unit for testing *Suppression* and *Error Processing*. For identifying features with problems, we consider each item to be an independent feature. An example of how **Item 9** handles *Error Processing* and *Suppression* of other items is shown in Table 6.1. Table 6.2 shows a partial listing of valid input and their combinations resulting in 4 test cases; any other input combination should generate errors. It is also to be noted that the rest of the item inputs remain the same for these 4 test cases.

Input Processing **(each item and valid combinations with other items are represented by test cases)**		
Assignments **(hidden)**	***Error Processing*** **(visible)**	***Suppression*** **(visible)**
Item 9 has a valid input of 1 and 2.	Any value other than 1 and 2 should generate error. Certain inputs for **Items 5, 13,** and **119** should not be allowed if **Item 9** is 2, and they should generate errors.	An entry of 1 should un-suppress **Items 30, 44** and **144**

Table 6.1: An Example *Assignments, Suppression* and *Error Processing*.

Item	**Valid Input**			
9	1		2	
5	1	5	2	3
13	<12/12/1990	>12/12/1990	<1/1/1991	1/12/1991
119	1	2	3	5

Table 6.2: An Example of Valid Input Combination for Testing Item 9.

6.3 Map test cases to features

Each item is representative of *Input Processing* feature to the end-user. Testers and end-users work together to provide the test case and feature mapping, shown in Table 6.3.

Features/Test Case	T1	T2	T3	T4	T5	T6	T7	T8
Item_5			■	■	■			
Item_9			■	■				
Item_13					■			
Item_16	■	■						■
Item_19		■						
Item_26	■							■
Item_119						■		
Item_212					■			
Item_431							■	

Table 6.3: AMS *Input Processing* Test Case and Feature Mapping (selective listing).

6.4 Map features to functions

Functions/Test Case	T1	T2	T3	T4	T5	T6
Process_Items	1,2,3,4,5,6,7,8,9,100,101,102,105,106,107,108,141,142,143,144,151	1,2,3,4,5,6,7,8,9,100,101,102,103,104,141,142,143,145,146,151	1,2,3,4,5,6,7,8,9,90,91,92,93,94,95,96,97,98,99,100,119,120,121,122,123,124,125,126,130,133,141,142,143,144,151	1,2,3,4,5,6,7,8,9,90,91,92,93,94,95,96,97,98,99,100,119,120,121,122,123,127,128,129,130,133,141,142,143,144,151	1,2,3,4,5,6,7,8,9,90,91,92,93,94,95,96,97,98,99,100,119,120,121,122,123,130,131,132,133,141,142,143,144,145,147,148,149,150,151	1,2,3,4,5,6,7,8,9,134,135,136,141,142,143,144,145,146,150,151
Calc_N	1,2,3,4,5,6	1,2,3,4,5,6	1,2,3,4,5,6	1,2,3,4,5,6	1,2,3,4,5,6	1,2,3,4,5,6
Process_Asterisks	1,2	1,2	1,2	1,2	1,2	1,2
Year_Values_From_Series	1,2,3,4,5,6,7.8.9,10	1,2,3,4,5,6,7.8.9,10	1,2,3,4,5,6,7.8.9,10	1,2,3,4,5,6,7.8.9,10	1,2,3,4,5,6,7.8.9,10	1,2,3,4,5,6,7.8.9,10
COL_EDIT	1	1	1	1	1	1
Census_Item	1,2	1,2	0	0	0	0
Within_Bounds	1	1	1	1	1	1
Nth_Elem_in_String	0	0	1,2	0	1,2	0
Mort_Fnctn	0	0	0	0	0	1,2,3,4,5,6,7,8,9
Adjust_Ages	0	1,2	0	0	1,2	1,2
Query_Check	1	1	1	1	1	1

Table 6.4: Profiler Listing of Features and Test Cases (selective listing).

To map features to functions, we instrumented the source code of AMS (only need to do this once) using code-coverage software and ran all regression tests. Appendix C describes the AMS regression-testing tool and its capabilities in detail. We then analyzed the coverage results and grouped related test cases together that exercised the input-processing feature for each item. A partial list of our result is shown in Table 6.4 and Table 6.5. Recall that to obtain a feature function relationship we follow the three-step process (see Section 4.1.3):

1. Obtain a feature and test case matrix as shown in Table 6.3.
2. Run the profiler on each of the test cases as shown in Table 6.4 to obtain the LOC within each function.
3. Finally, we obtain feature function relationship by combining steps 1 and 2 as shown in Table 6.5. We used the code-coverage tool *TrueCoverage*™ from *NuMega*® which works with many programming languages such as VB, Java, C++, and some scripting languages. Since AMS uses batch

processing for its regression testing, it was easy to produce instrumented output against all the 250 regression test cases. However, these instrumented images were stored using *TrueCoverage*'s proprietary file format, so we had to manually export each file into Excel for further analysis. The *TrueCoverage* tool has a *merge utility* that aggregated the results of all 250 test cases that were instrumented. This *merge utility* revealed that 95% of AMS was covered using the 250 test cases. We are currently identifying whether the rest of the code is either unused or if there are hidden features within the system that are not being exercised. For each test case, we used *TrueCoverage* to identify the functions executed, the percentage of lines covered within each of these functions, and the variables used.

Features/Functions	Process_Items	Calc_N	Process_Asterisks	Year_Values_From_Series	COL_EDIT	Census_Item
Item_5	001,002,003,004,005,006,007,008,009,090,091,092,093,094,095,096,097,098,099,100,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133,141,142,143,144,145,147,148,149,150,151	001,002,003,004,005,006	001,002	001,002,003,004,005,006,010	001	
Item_9	001,002,003,004,005,006,007,008,009,090,091,092,093,094,095,096,097,098,099,100,119,120,121,122,123,124,125,126,127,128,129,130,133,141,142,143,144,151	001,002,003,004,005,006	001,002	001,002,003,004,005,006,010	001	
Item_13	001,002,003,004,005,006,007,008,009,090,091,092,093,094,095,096,097,098,099,100,119,120,121,122,123,130,131,132,133,141,142,143,144,145,147,148,149,150,151	001,002,003,004,005,006	001,002	001,002,003,004,005,006,010	001	
Item_16	001,002,003,004,005,006,007,008,009,100,101,102,103,104,105,106,107,108,115,116,117,118,141,142,143,144,145,146,147,148,149,150,151	001,002,003,004,005,006	001,002	001,002,003,004,005,006,010	001	001,002
Item_19	001,002,003,004,005,006,007,008,009,100,101,102,103,104,141,142,143,145,146,151	001,002,003,004,005,006	001,002	001,002,003,004,005,006,010	001	001,002
Item_26	001,002,003,004,005,006,007,008,009,100,101,102,105,106,107,108,115,116,117,118,141,142,143,144,147,148,149,150,151	001,002,003,004,005,006	001,002	001,002,003,004,005,006,010	001	001,002
Item_212	001,002,003,004,005,006,007,008,009,134,135,136,141,142,143,144,145,146,150,151	001,002,003,004,005,006	001,002	001,002,003,004,005,006,010	001	
Item_119	001,002,003,004,005,006,007,008,009,090,091,092,093,094,095,096,097,098,099,100,119,120,121,122,123,130,131,132,133,141,142,143,144,145,147,148,149,150,151	001,002,003,004,005,006	001,002	001,002,003,004,005,006,010	001	
Item_431	001,002,003,004,005,006,007,008,009,134,135,136,141,142,143,144,145,147,148,149,150,151	001,002,003,004,005,006	001,002	001,002,003,004,005,006,010	001	

Table 6.5: Feature and Function Mapping (selective listing).

6.5 Identify FI and CORE

Table 6.5 provides a selective and pre-sorted listing of all the relevant functions (FI) for the *input-processing* feature. In order to arrive at feature function listing, we must identify the base-line architecture and the CORE as these functions are executed for all test cases at all times. The main difference between the base-line architecture and the CORE is that the base-line architecture changes the state of global variables and other sub-systems such as databases, at an architectural level. CORE on the other hand contains stateless functions that are utility functions. A list of base-line architectural functions and CORE is shown in Table 6.7 and Table 6.8 respectively.

Running regression test cases for all items (representing the *Input Processing* feature) resulted in following information regarding FI for *Input Processing*:

- Since each item represented the *Input Processing* feature we aggregated the profiler information for all items. The aggregation process was similar to UNION of all lines of code process discussed in Section 4.1.3.2. Aggregated information revealed that *Input Processing* is implemented in 17 functions. A partial listing of those functions is shown in Table 6.4.
- Many items shared same lines of code indicating either a circular dependency as mentioned earlier or dependent on some base items (such as **Item 9**). The shared lines of code can be seen in Table 6.5.

- About 68% of *Input Processing* is implemented in Process_Items function. Another 16 functions were either used in implicit communication or were stateless in nature. The three neighboring features included *Assignment*, *Error Processing,* and *Suppression*. The threshold (FI = 17, K = 3, C = 80%) suggests that we continue with the refactoring.

We ran all the 250 regression test cases to see how *Input Processing* measured relative to the rest of the features. A partial result of the regression test case is shown in Table 6.6. We will discuss this matrix in more detail in Chapter 7. However, the interesting region to observe is the shaded area on the top left which represents part of *Input Processing*. FE_1, FE_2, and FE_3 represent *Assignment*, *Error Processing* and the *Suppression* sub-features respectively. `Process_Items` is represented by $f_{2.}$ Functions $f_{5,}$ f_6 and f_7 are shared stateless but only within the scope of *Input Processing* features, such examples being functions `query_check`, `calc_n` and `asterisks_item` (also shown in Table 6.4). Functions f_{18}, f_{19}, and f_{20} are CORE functions `file_exists`, `integer_maximum`, and `integer_minimum` also shown in the Table 6.8. Comparing *Input Processing* features (FE_1, FE_2, and FE_3) to the rest of the features shown in Table 6.8, we observe that *Input Processing* is a natural choice for evolution because of its threshold value (FI = 17, K = 3 and C = 80%) relative to the other features.

Test Cases	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	
Function\Features	FE_1	FE_2	FE_3	FE_4	FE_5	FE_6	FE_7	FE_8	FE_9	FE_{10}	FE_{11}	FE_{12}	Type
f_1	100	100	100	0	0	0	0	0	0	0	0	0	SS
f_2	80	80	90	0	0	0	0	0	0	0	0	0	DF
f_3	100	100	100	0	0	0	0	0	0	0	0	0	SS
f_4	70	75	80	0	0	0	0	0	0	0	0	0	DD
f_5	100	100	100	0	0	0	0	0	0	0	0	0	SS
f_6	100	100	100	0	0	0	0	0	0	0	0	0	SS
f_7	100	100	100	0	0	0	0	0	0	0	0	0	SS
f_8	0	0	0	80	90	0	0	0	0	0	0	0	DF
f_9	0	0	0	100	100	0	0	0	0	0	0	0	DF
f_{10}	0	0	0	0	0	11	12	0	0	0	0	0	SSF
f_{11}	0	0	0	0	0	33	44	0	0	0	0	0	SSF
f_{12}	0	0	0	0	0	12	15	0	0	0	0	0	SSF
f_{13}	0	0	0	0	0	15	15	0	0	0	0	0	SS
f_{14}	0	0	0	0	0	22	22	0	0	0	0	0	SS
f_{15}	0	0	0	0	0	32	38	0	0	0	0	0	SS
f_{16}	0	0	0	0	0	15	15	15	15	15	15	15	SS
f_{17}	0	0	0	0	0	22	22	22	22	22	22	22	SS
f_{18}	100	100	100	100	100	100	100	100	100	100	100	100	SS
f_{19}	100	100	100	100	100	100	100	100	100	100	100	100	SS
f_{20}	100	100	100	100	100	100	100	100	100	100	100	100	SS
f_{21}	0	0	0	0	0	32	38	12	56	89	66	63	DD
f_{22}	0	0	0	0	0	22	22	34	52	23	43	34	DD
f_{23}	0	0	0	0	0	32	38	22	44	33	45	32	DD

Table 6.6: Partial AMS Feature Function Matrix.

	BASE-LINE ARCHITECTURE	Purpose
	Input Engine	
1	MAIN	Required by the programming language
2	VERIFY_USER	Validate User
3	LOAD_LICENSE_FORM	Display licensing information
4	SET_RUNTIME_PARAMETERS	Load some key global variables
5	OPEN_CASE	Open Master and Census File
6	LOAD_MAIN_FORM	Master form is loaded in memory
7	OPEN_ITEM_FILE	Meta data is loaded
8	MERGE_USER_DATA	Merge user data into new structure
9	INPUT_PROCESSING	Assignment, Error Processing and Suppression
10	SETUP_HELP_FILES	Load Help files in memory
11	INITIALIZE_DISPLAY_VARIABLES	Setup more global variables
12	SETUP_USER_PROFILES	Load selected user preferences
13	DISPLAY_BUTTONS	Load and show icons in the main form
14	DISPLAY_MASTER_FILE	Show screens with master file data
15	DISPLAY_CENSUS_FILE	Show screens with census file data
16	SETUP_STATUS_BAR	Show status of current user selection
17	EXIT_INPUT_ENGINE	Close connections, reset variables and end
	Calculation Engine	
1	PROCESS_CASE	Setup global variables
2	MAIN_CALC	Start calculations
3	PROCESS_EACH_EMPLOYEE	Loads assumptions to be calculated
4	INITIALIZE_EACH_EMPLOYEE	Setup global variables for Employee
5	PROCESS_EACH_YEAR	Calculation for each year
6	INITIALIZE_EACH_YEAR	Setup yearly global variables
7	MANDATORY_CALCULATIONS_PER_YEAR	Compulsory calculations per year
8	PRINT_EACH_YEAR	Print stored variables
9	ACCUMULATE_RESULTS	Accumulate certain key variables
10	STORE_RUNTIME_ERRORS	Collect run time errors
11	CLOSE_RUN	Reset variables and database connection
12	EXIT_CALC_ENGINE	Exit engine and return control to Input Engine
	Output Engine	
1	PROCESS_CASE	Setup global variables
2	OPEN_DATABASE	Connect to the database that has data to print
3	EVALUATE_REQUESTS	Which reports to print
4	LOAD_REPORTING_DATA	Fetch data to be printed

Table 6.7: Base-Line Architecture of the Three AMS Engines.

	LIST OF CORE FUNCTIONS	Return Value
1	APPEND_TO_STRING (string source, string newstring)	String
2	ARRAY_TO_RECORD (string a())	Recordset
3	BACKUP_DATABASE (string sourcedatabase, string targetdatabase)	Boolean
4	BUBBLE_SORT (string a())	Sorted Array
5	CLOSE_DATABASE (string databasename)	Boolean
6	COMPARE_RECORDS (recordset a, recordset b)	Boolean
7	CONNECT_DATABASE (string databasename)	Boolean
8	DATABASE_EXISTS (string databasename)	Boolean
9	DATE_TO_AGE (date a, date todaysdate)	Integer AGE
10	DATE_TO_NUMBER (date a)	Double
11	DIRECTORY_EXISTS (string pathname)	Boolean
12	DOUBLE_MAXIMUM (double a, double b)	Double
13	DOUBLE_MINIMUM (double a, double b)	Double
14	DOUBLE_ROUND (double a, int n)	Double
15	ERROR_LOG (string filename, string message)	Boolean
16	EXECUTE_SQL (string sql)	Boolean
17	FIELD_EXISTS (string fieldname)	Boolean
18	FILE_EXISTS (string a)	Boolean
19	INTEGER_MAXIMUM (int a, int b)	Integer
20	INTEGER_MINIMUM (int a, int b)	Integer
21	IS_A_CURRENCY (string a)	Boolean
22	IS_A_VALID_USER (string username, string password)	Boolean
23	IS_ALPHA (string a)	Boolean
24	IS_BLANK (string a)	Boolean
25	IS_DATE (string a)	Boolean
26	IS_NULL (string a)	Boolean
27	IS_NUMBER (string a)	Boolean
28	OPEN_DATABASE (string databasename)	Boolean
29	READ_INI_FILE (string filename, object returned)	Object
30	READ_REGISTRY_KEY (string keyname)	Boolean
31	RECORD_TO_ARRAY (recordset a)	Array
32	REMOVE_FROM_STRING(string source, string target)	String
33	SINGLE_MAXIMUM (single a, single b)	Single
34	SINGLE_MINIMUM (single a, single b)	Single
35	SINGLE_ROUND (single a, int n)	Single
36	STRING_TO_DOUBLE_ARRAY (string a)	Double
37	STRING_TO_INTEGER_ARRAY (string a)	Integer
38	STRING_TO_SINGLE_ARRAY (string a)	Array
39	SWAP (variant a, variant b)	VOID
40	TABLE_EXISTS (string tablename)	Boolean
41	WRITE_INI_FILE (string filename, object)	Boolean
42	WRITE_REGISTRY_KEY (string keyname)	Boolean

Table 6.8: List of CORE Functions Extracted from AMS.

6.5.1 Variable Analysis

AMS program architecture has 1,205 global variables. All the global variables are declared and initialized at the program level. However, these global variables are shared among features and are changed dynamically to facilitate implicit communication. A summary of key variables is shown in Table 6.9. The input values are read into sI$() array which is used by many features within the AMS system. The sI$() array is converted into its numeric counterpart within the *Input Processing* feature by the Assignment sub-features. An example of this conversion can be seen in the statement `nContract_Number = Val(sI$(9))` in Figure 6.2. `nContract_Number`, which is a global variable, is then used throughout the AMS system. Each of the items uses, sets, and changes the state of several global variables thereby increasing coupling between the features. These global variables are also used for implicit communication between features. For example, `UNREADY()`, `nError_F`, `nError_Item` and `nItem` can be changed by items other than the ones who set them indicating a possible relationship among the items and features.

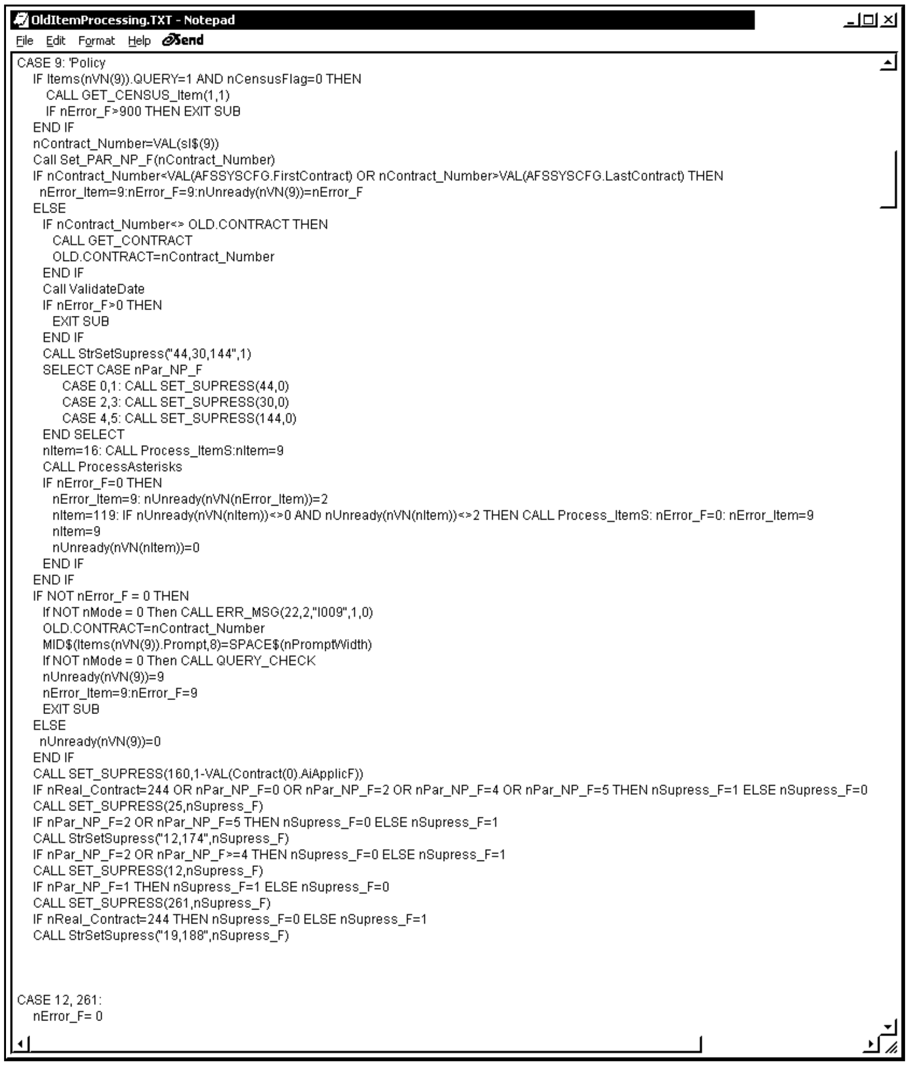
Figure 6.2 suggests that three types of relationships exist among the *Input Processing* features:

- *Error Processing* and *Suppression* depend on *Assignments*.

- *Suppression* state of certain items is altered depending upon *Assignments* of certain items.
- *Error Processing* and *Suppression* share global variables set by *Assignments*.

Variable	Purpose	Declare	Set	Use	Change
sI$()	Stores item information as read from database	Program	Program	*Input Processing* and other features	
Numeric Value of sI$(). All items are assigned into internal variables such as nContract_Number, nDB_Option etc.	Assignment of string arrays into numerical variables.	Program	*Input Processing* (Assignment Sub-feature)	*Input Processing* (*Error Processing*, *Suppression*) and Calculation Engine	
nError_F, nError_Item	Indicates if an item is in error	Program	*Input Processing* (*Error Processing*)	*Input Processing* (*Error Processing*, *Suppression*)	*Input Processing* (Other items)
UNREADY()	Indicates if an item is ready with its numerical value	Program	*Input Processing* (Assignment)	*Input Processing* (Assignment)	*Input Processing* (Other items)
nSuppress_F	Indicates if an item is suppressed	Program	*Input Processing* (*Suppression*)	*Input Processing* (*Suppression*)	*Input Processing* (Other items)
nItem	Indicates which item is being processed	Program	*Input Processing* (Assignment)	*Input Processing* (Assignment)	*Input Processing* (Other items)

Table 6.9: *Input Processing* Variable Analysis.



```
CASE 9: 'Policy
   IF Items(nVN(9)).QUERY=1 AND nCensusFlag=0 THEN
      CALL GET_CENSUS_Item(1,1)
      IF nError_F>900 THEN EXIT SUB
   END IF
   nContract_Number=VAL(sI$(9))
   Call Set_PAR_NP_F(nContract_Number)
   IF nContract_Number<VAL(AFSSYSCFG.FirstContract) OR nContract_Number>VAL(AFSSYSCFG.LastContract) THEN
    nError_Item=9:nError_F=9:nUnready(nVN(9))=nError_F
   ELSE
     IF nContract_Number<> OLD.CONTRACT THEN
       CALL GET_CONTRACT
       OLD.CONTRACT=nContract_Number
     END IF
     Call ValidateDate
     IF nError_F>0 THEN
       EXIT SUB
     END IF
     CALL StrSetSupress("44,30,144",1)
     SELECT CASE nPar_NP_F
         CASE 0,1: CALL SET_SUPRESS(44,0)
         CASE 2,3: CALL SET_SUPRESS(30,0)
         CASE 4,5: CALL SET_SUPRESS(144,0)
     END SELECT
     nItem=16: CALL Process_ItemS:nItem=9
     CALL ProcessAsterisks
     IF nError_F=0 THEN
       nError_Item=9: nUnready(nVN(nError_Item))=2
       nItem=119: IF nUnready(nVN(nItem))<>0 AND nUnready(nVN(nItem))<>2 THEN CALL Process_ItemS: nError_F=0: nError_Item=9
       nItem=9
       nUnready(nVN(nItem))=0
     END IF
   END IF
   IF NOT nError_F = 0 THEN
     If NOT nMode = 0 Then CALL ERR_MSG(22,2,"I009",1,0)
     OLD.CONTRACT=nContract_Number
     MID$(Items(nVN(9)).Prompt,8)=SPACE$(nPromptWidth)
     If NOT nMode = 0 Then CALL QUERY_CHECK
     nUnready(nVN(9))=9
     nError_Item=9:nError_F=9
     EXIT SUB
   ELSE
    nUnready(nVN(9))=0
   END IF
   CALL SET_SUPRESS(160,1-VAL(Contract(0).AiApplicF))
   IF nReal_Contract=244 OR nPar_NP_F=0 OR nPar_NP_F=2 OR nPar_NP_F=4 OR nPar_NP_F=5 THEN nSupress_F=1 ELSE nSupress_F=0
   CALL SET_SUPRESS(25,nSupress_F)
   IF nPar_NP_F=2 OR nPar_NP_F=5 THEN nSupress_F=0 ELSE nSupress_F=1
   CALL StrSetSupress("12,174",nSupress_F)
   IF nPar_NP_F=2 OR nPar_NP_F>=4 THEN nSupress_F=0 ELSE nSupress_F=1
   CALL SET_SUPRESS(12,nSupress_F)
   IF nPar_NP_F=1 THEN nSupress_F=1 ELSE nSupress_F=0
   CALL SET_SUPRESS(261,nSupress_F)
   IF nReal_Contract=244 THEN nSupress_F=0 ELSE nSupress_F=1
   CALL StrSetSupress("19,188",nSupress_F)



CASE 12, 261:
   nError_F= 0
```

Figure 6.2: Item 9 (Pre-evolution).

6.6 Refactor and create components

We identified the following problems in the *Input Processing* feature of AMS:

6.6.1 Identify problems

This step identifies problems associated with the feature implementation.

6.6.1.1 Circular dependencies

As Table 5 shows, **Item 9** is dependent on **Item 119** and **Item 119** is dependent on **Item 13**, which in fact is dependent on **Item 9**. We found eight such circular dependencies that were the ultimate cause of system freezes as verified by the defect tracking system for AMS.

Item	Dependencies (in order)
5	9, 56, 119
9	16, 119
13	5, 9, 22
19	158
119	13

Table 6.10: Example of Circular Dependencies

6.6.1.2 Readiness of dependent items

To solve the circular dependencies and determine an item's state during assignment, we found that the original developers used an array called `UNREADY`: when an item is dependent on another item that still needs to be evaluated, the original item is identified as being in the `UNREADY` state. Each item had a ready and unready state. The code fragment in Figure 6.3 illustrates this: **Item 5** is assumed to be ready by setting `UNREADY(5)` to 1. The item's value is then evaluated and the global `nError_F` is set to be greater than 1 in case of invalid input. The `UNREADY` state for **Item 5** will be set to the error flag's value indicating that the item is not ready. Items are processed sequentially so if another item dependent upon **Item 5** needs its value then the calling item will use `UNREADY(5)`. The implicit setting of item state resulted in bad patches to solve circular dependencies.

```
nUnready(5) = 1       ‘ 1 = ready
call Fix_Date(nItem)
if nError_F > 0 Then
  nUnready(5) = nError_F
end If
```

Figure 6.3: Dependent Items.

6.6.1.3 *Assignments* and *Suppression* intermingled with *Error Processing*

As items were evaluated for dependencies and error conditions the original code also set the values of internal program variables. AMS often uses a *time series* in most plan items. An example of a time series is "100,1,200,5" which means that from years 1 through 5, the value is 100 and from year 5 onwards it is 200. Time series presents complicated problems because the data needs to be evaluated over a period of time (or processed via the *Input Processing*) and errors can be present in any year. We found that internal *Assignments* were often used inconsistently and intermingled with *Error Processing* and *Suppression*.

Once we identify feature implementations, we refactor the code as outlined in Section 6.6.2. Refactoring removes global variables and converts implicit communication to explicit. Refactoring may require extensive analysis, especially if two or more features interact or interfere within a given source function. We have found that the refactoring results in fine-grained components with low coupling and high cohesion.

6.6.2 Refactor

For *Error Processing*, *Suppression*, and *Assignments* we refactored the code as follows:

6.6.2.1 Removed UNREADY array

The `UNREADY` array forced the *Assignments* and *Suppression* code to be highly coupled. We replaced this global array with a component that accepted a collection of errors. Then we developed routines (add, display, and delete) to access the collection for one individual or the entire census data.

6.6.2.2 Replaced recursive calls with sequential calls

In the original system, *Error Processing*, *Suppression* and *Assignments* were largely recursive. Essentially, a single large routine (Process_Items) inspected each item using a lengthy `case` statement; when an item needed to check dependencies for another item, a recursive call was made. After some analysis, we replaced this function with a simpler more sequential control flow

6.6.2.3 Separated *Assignments*, *Suppression*, and *Error Processing* code

After analyzing *Input Processing*, we were able to remove circular dependencies by first executing *Assignments* for certain core items. We found this was consistent with all three features.

6.6.3 Create Fine-Grained Components

To determine which code artifacts to encapsulate, we analyzed variable usage for all three features: *Error Processing (EP)*, *Suppression (S)*, and *Assignments (A)*. The result is shown in Table 6.11. (EP/S means variables involved both in EP and S).

	Pre-Evolution (Feature Related Variables)					Post-Evolution (Component Properties)			
Feature Interaction	Global variables	Local Variables	Stateless Functions	State-full Functions	Relation Type	Component Property GetValue	Component Property SetValue	Local To Component	AFS CORE Parameters
Error Processing	35	5	4	2	N/A	25	10	5	6
Suppression	14	8	6	4	N/A	10	12	6	4
Assignment	50	5	8	5	N/A	55	5	4	4
Error Processing and Suppression	11	3	3	3	Dependent Required	8	6	2	4
Error Processing and Assignment	20	5	4	3	Dependent	17	8	3	4
Suppression and Assignment	25	6	3	2	Dependent Alteration	18	12	4	2
Error Processing, Suppression and Assignment	8	9	2	2	Required	6	7	4	4

Table 6.11: Variable Analysis (Pre/Post evolution partial listing).

When creating fine-grained components, these variables and functions become properties of a component. The first two columns in Table 6 count the global (G) and local (L) variables involved in a particular feature implementation when related test cases are executed. Columns three and four show how many functions, both stateless (SS) and state-full (SSF), are covered. The component makes output values available using `GetValue (Parameter)`. Conversely, `SetValue (Parameter)` will set the property inside the component. Because we are refactoring, the sum of the first four columns for each row must equal the sum of the last four columns.

To define the interface for the fine-grained components, we must identify the possible relationships between features.

6.6.3.1 Relationships

In *Input Processing* we find the examples of the following types of direct relationships among features.

Dependent

In AMS all features share key item values. The code fragment in Figure 6.4 shows how key items are evaluated first and used in *Suppression* and *Assignments.* The variable *`QMarkInBPFA`* is set to true if **Item 16** has a `“?”`. We convert this variable into a read-only property of the *Assignments* component that can be read by other components.

```
Dim QMarkInBPFA As Boolean
Dim QmarkInUlPremType As Boolean
Dim XInBPFA As Boolean
Dim ISBEN As Boolean

QMarkInBPFA = isfloated(Values(16), False)
QmarkInUlPremType = isfloated(Values(174), False)
XInBPFA = XInItem(Values(16))
ISBEN = InStr(Values(26), ",BEN,") > 0 or
        InStr(Values(26), ",A/T.BEN,") > 0)
```

Figure 6.4: Dependent Feature Example.

Required

The function in

Figure 6.5 implements the relationship between *Suppression* and *Error Processing*. If an item is suppressed, then errors associated with it are unnecessary and can be removed. Because two features can directly interact with each other, the extracted fine-grained components will have clearly defined interfaces that

```
Public sub RemoveErrorsForSuppressedItems (
      suppressarray() as Integer, Errors as Collection)
dim x, itemNum as Integer
 dim s as String
 for x = Errors.count to 1 step -1
   itemNum = AFSCore.FVAL(Mid$(Errors.Item(x),
               InStr(Errors.Item(x), ">") + 1))
   if suppressarray(ItemNum) <> 0 then
      Errors.Remove(x)
   end if
 next x
End Sub
```

Figure 6.5: Required Feature Example.

Altered

The state of *Suppression* of a given item is altered by the entries in another item. For example, the *Suppression* state of **Item 98** in

Figure 6.6 can be modified with the right condition. Note that the *Assignments* component's properties are used to alter the *Suppression* state. If the UI changes the value for any field that can alter **Item 98**, the *Suppression* state is also altered. The global array `nSuppress()` is transformed into a read/write property of the *Suppression* component.

```
if Assignments.QMarkInBPFA or (Assignments.XInBPFA and
Assignments.SipFloat) or Assignments.ISBEN then
        nSuppress(98) = UnSuppressTheItem(nSuppress(98)) else
        nSuppress(98) = SuppressTheItem(nSuppress(98))
end if
```

Figure 6.6: Altered Feature Example.

Once feature relationships and properties are determined, we can create the component's interface.

6.6.3.2 Interfaces

Input Processing was refactored into six components: *Assignments*, *Error Processing*, *Suppression*, *Error Processing* Core, *Suppression* Core, and AFS Core. While *Assignments*, *Error Processing*, and *Suppression* perform specific duties of the three specified features, the core components manage data structures and contain stateless functions. In implementing these features, core items were evaluated first and each item was called sequentially instead of recursively. Feature relationships were identified and coded as shown earlier.

The list of interface for all six components that we created are shown in the Table 6.12. Public interfaces that are available to the client is shown in column 2 and column 3 lists all the components public methods. The *Error Processing* core and the *Suppression* core components are not important by themselves but they are important in conjunction with the *Error Processing* the *Suppression* components respectively. “Methods” column (column 3 in Table 6.12) implements the feature implementation, which will discussed later in this section. Note that only the public methods are listed in column 3.

Component	**Interface**	**Methods**
Assignments	clsAssignment	*Assignments*
Error Processing	clsErrorProcessing	ErrorChecking
Error Processing Core	clsEProcessingCore	AddError ClearError RemoveError RemoveErrorForSuppressed Item ClearAllErrors
Suppression	cls*Suppression*	SetTheSuppressCodes
Suppression Core	cls*Suppression*Core	SuppressTheItem UnSuppressTheItem
AFS Core	clsAFSCore	See Table 6.8

Table 6.12: Component Interfaces.

6.6.3.3 Properties

Once feature relationships were identified, the global variables used in the implicit communications were used to determine property set and get. The general logic for invoking a feature is to set the components properties, call its public method (feature implementation) and finally retrieve the results through the property get. It is the calling program's responsibility to set the components properties (both get and set). A list of properties, get and set for the six components is shown in the Table 6.13. It is to be noted that dictionary and collection objects are built-in data structures like array. They are used to set individual items values rather than setting them up individually. AFS CORE does not need any properties, as all its methods are public and stateless.

Component	Interface.Public Method	Property Set	Property Get
Assignments	cls*Assignments* .*Assignments*	sI$()	Dictionary (nItem, Value)
Error Processing	clsErrorProcessing .ErrorProcessing	Dictionary (nItem, Value)	Collection (nItem, Error_Message, Error_Code)
Suppression	cls*Suppression* .SetTheSuppressCode	Dictionary (nItem, Value) and Collection (nItem, Error_Message, Error_Code)	Dictionary (nItem, *Suppression*Code)
Error Processing CORE	clsErrorProcessingCORE .AddError .ClearError .RemoveError .RemoveErrorForSuppressItem .ClearAllErrors	Dictionary (nItem, Value)	Collection (nItem, Error_Message, Error_Code)
Suppression CORE	cls*Suppression*Core .SuppressTheItem .UnSuppressTheItem	Dictionary (nItem, Value) and Collection (nItem, Error_Message, Error_Code)	Dictionary (nItem, *Suppression*Code)
AFS CORE	clsAFSCORE .AFSCORE	None	None

Table 6.13: Component Properties.

6.6.3.4 Feature Implementation

Figure 6.7, Figure 6.8, Figure 6.9, Figure 6.10 and Figure 6.11 all show how the *Input Processing* component is implemented using *Error Processing*, *Suppression*, *Assignments* and supporting core (AFS CORE, *Suppression* CORE and *Error Processing* CORE) fine-grained components. *Assignments* component's properties are set by the calling program and then its public method Sub *Assignments* is called. Sub *Assignments* has following three tasks:

- to calculate base-items first, as these item values are used by other items.
- to sequentially calculate the value of each of the items.
- to update the local dictionary object which is accessed through the property *get* by the calling parent program.

Likewise, *Error Processing* and *Suppression* features also follow the similar implementation. However, their tasks are slightly different. *Error Processing* uses an items dictionary object and sets an internal collection object that can be accessed through property get. *Suppression* uses items dictionary object and error collection object to setup the *Suppression* state (0 or 1) of each item in an internal data structure called *Suppression*Dictionary to be accessed through the get property. The CORE items do not have any feature implementation. These components contain stateless components.

6.6.3.5 Stateless Functions

Figure 6.7, Figure 6.8, and Figure 6.9 all show a partial listing of stateless functions needed for the feature implementation. These functions can also be accessed by other components or by the calling parents. An example of a stateless function that is encapsulated within the *Assignments* component is Public

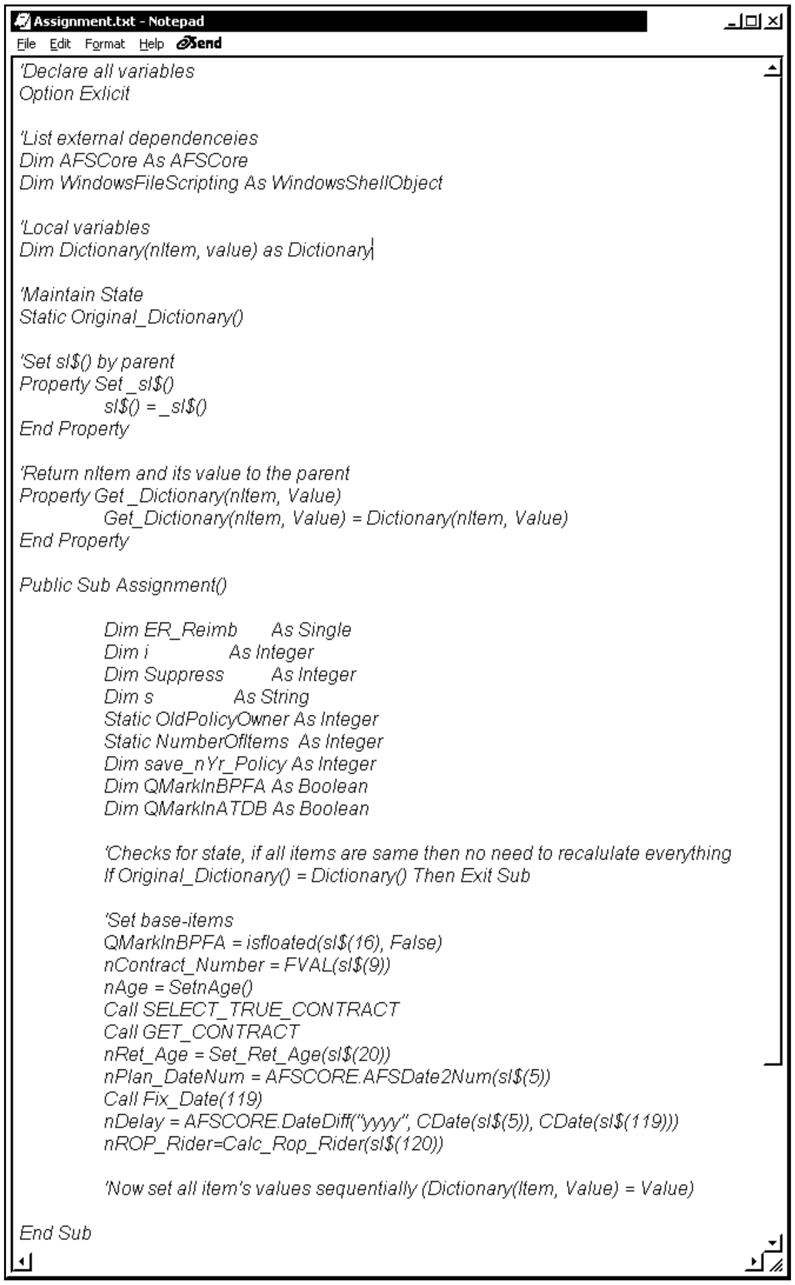
Function `Set_Ret_Ages()`. This function calculates retirement age based upon a date string that is passed to it. Although these stateless functions are accessible and can be used by other components and the program, having them encapsulated with the features provides a clearer understanding of the feature's scope and involvement.

6.6.3.6 Maintaining State

Figure 6.7, Figure 6.8, and Figure 6.9 all show that the components maintain their state for efficiency. For example, if the *Assignments* component public method is called a second time (intentionally or unintentionally) and the items values have not changed, then the public sub *Assignments* does not recalculate the entire set of item variables because the source (sI$()) has not changed. Similar implementation characteristics can also be seen in the *Error Processing* and the *Suppression* components.

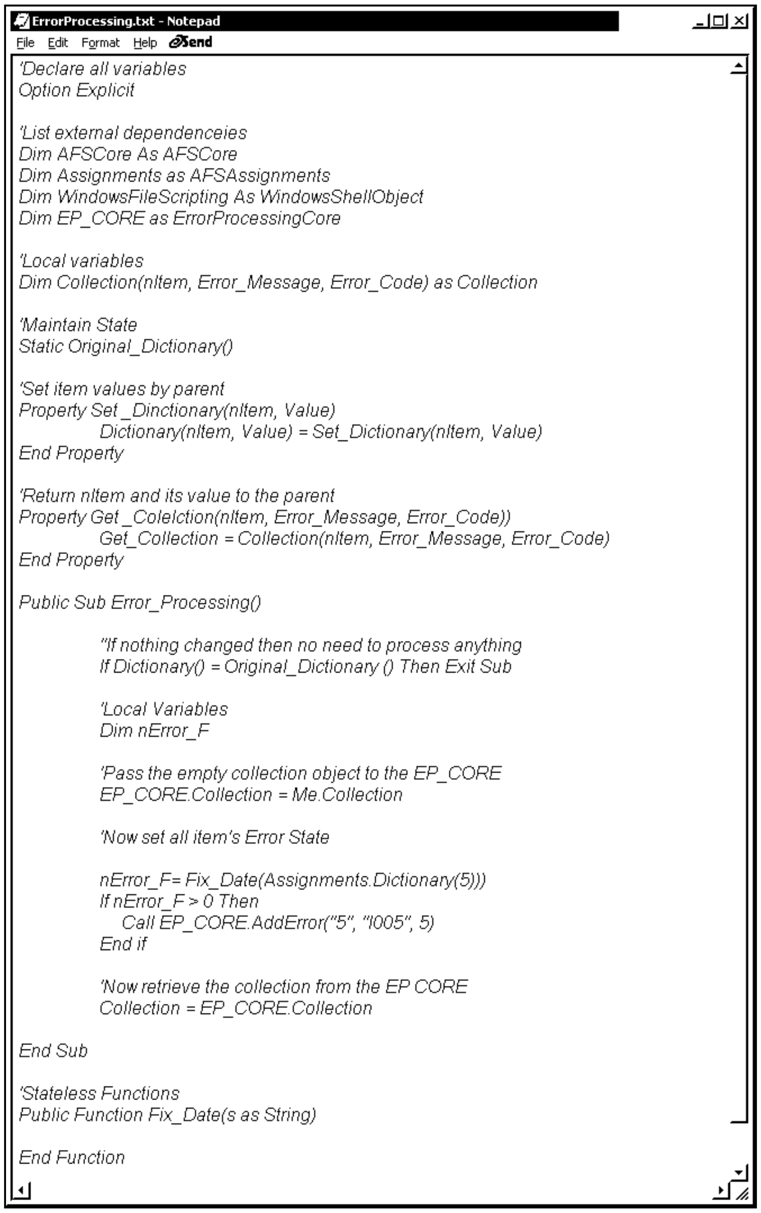
6.6.3.7 External Dependencies

At the top of Figure 6.7, Figure 6.8, Figure 6.9, Figure 6.10 and Figure 6.11 we list all the external components and functions the components are going to use such as AFS CORE and WindowsFileScripting object. VB provides built-in programming constructs such as collection and dictionary objects that are like indexed arrays.



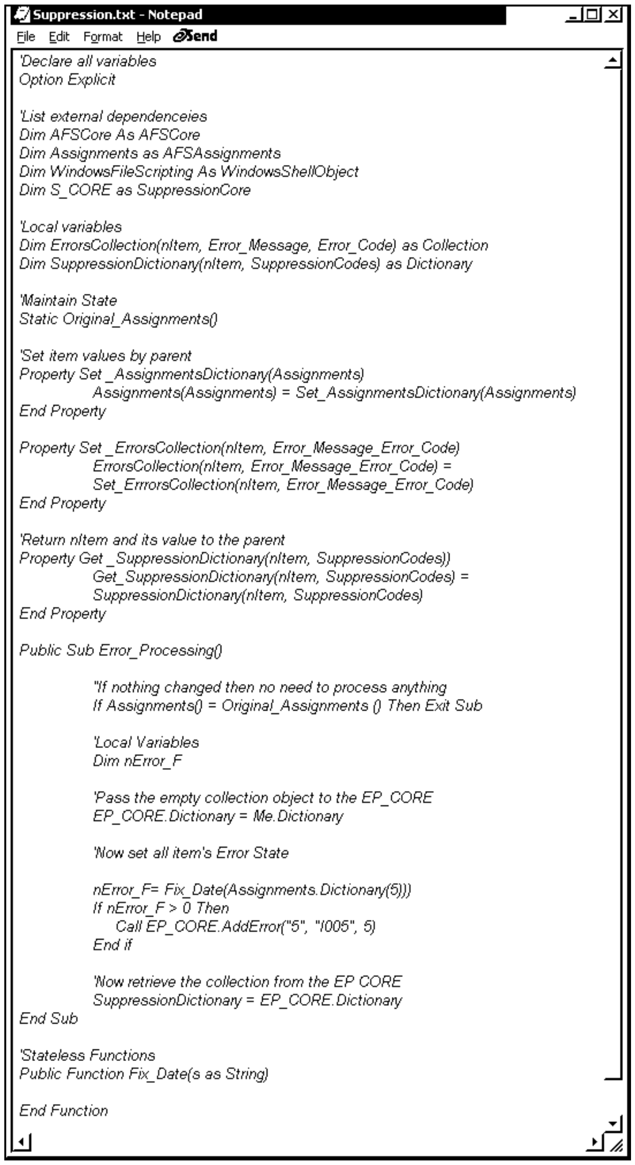
```
'Declare all variables
Option Explicit

'List external dependenceies
Dim AFSCore As AFSCore
Dim WindowsFileScripting As WindowsShellObject

'Local variables
Dim Dictionary(nItem, value) as Dictionary

'Maintain State
Static Original_Dictionary()

'Set sl$() by parent
Property Set _sl$()
        sl$() = _sl$()
End Property

'Return nItem and its value to the parent
Property Get _Dictionary(nItem, Value)
        Get_Dictionary(nItem, Value) = Dictionary(nItem, Value)
End Property

Public Sub Assignment()

        Dim ER_Reimb      As Single
        Dim i             As Integer
        Dim Suppress      As Integer
        Dim s             As String
        Static OldPolicyOwner As Integer
        Static NumberOfItems  As Integer
        Dim save_nYr_Policy As Integer
        Dim QMarkInBPFA As Boolean
        Dim QMarkInATDB As Boolean

        'Checks for state, if all items are same then no need to recalulate everything
        If Original_Dictionary() = Dictionary() Then Exit Sub

        'Set base-items
        QMarkInBPFA = isfloated(sl$(16), False)
        nContract_Number = FVAL(sl$(9))
        nAge = SetnAge()
        Call SELECT_TRUE_CONTRACT
        Call GET_CONTRACT
        nRet_Age = Set_Ret_Age(sl$(20))
        nPlan_DateNum = AFSCORE.AFSDate2Num(sl$(5))
        Call Fix_Date(119)
        nDelay = AFSCORE.DateDiff("yyyy", CDate(sl$(5)), CDate(sl$(119)))
        nROP_Rider=Calc_Rop_Rider(sl$(120))

        'Now set all item's values sequentially (Dictionary(Item, Value) = Value)

End Sub
```

Figure 6.7: Pseudo-code for *Assignments* Feature Implementation.



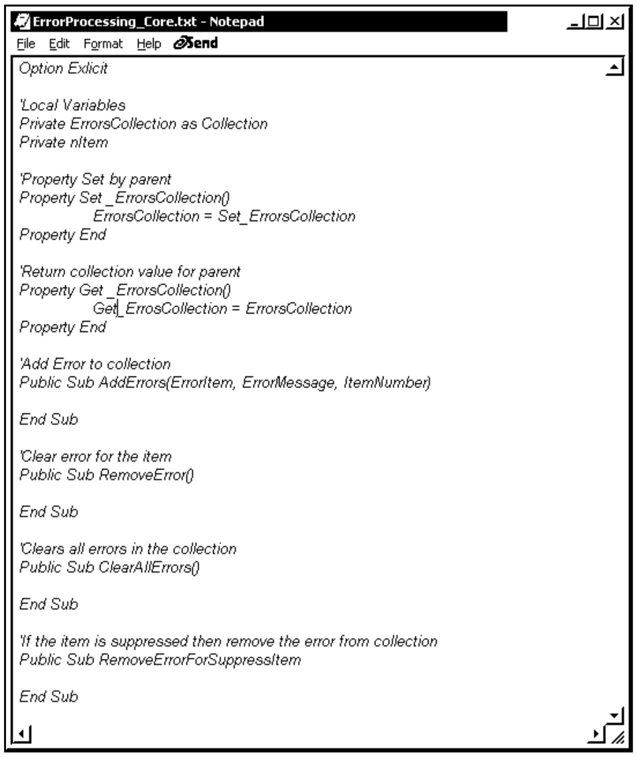
```
'Declare all variables
Option Explicit

'List external dependenceies
Dim AFSCore As AFSCore
Dim Assignments as AFSAssignments
Dim WindowsFileScripting As WindowsShellObject
Dim EP_CORE as ErrorProcessingCore

'Local variables
Dim Collection(nItem, Error_Message, Error_Code) as Collection

'Maintain State
Static Original_Dictionary()

'Set item values by parent
Property Set _Dinctionary(nItem, Value)
            Dictionary(nItem, Value) = Set_Dictionary(nItem, Value)
End Property

'Return nItem and its value to the parent
Property Get _Colelction(nItem, Error_Message, Error_Code))
            Get_Collection = Collection(nItem, Error_Message, Error_Code)
End Property

Public Sub Error_Processing()

            ''If nothing changed then no need to process anything
            If Dictionary() = Original_Dictionary () Then Exit Sub

            'Local Variables
            Dim nError_F

            'Pass the empty collection object to the EP_CORE
            EP_CORE.Collection = Me.Collection

            'Now set all item's Error State

            nError_F= Fix_Date(Assignments.Dictionary(5)))
            If nError_F > 0 Then
               Call EP_CORE.AddError("5", "I005", 5)
            End if

            'Now retrieve the collection from the EP CORE
            Collection = EP_CORE.Collection

End Sub

'Stateless Functions
Public Function Fix_Date(s as String)

End Function
```

Figure 6.8: Pseudo-code for *Error Processing* Feature Implementation.



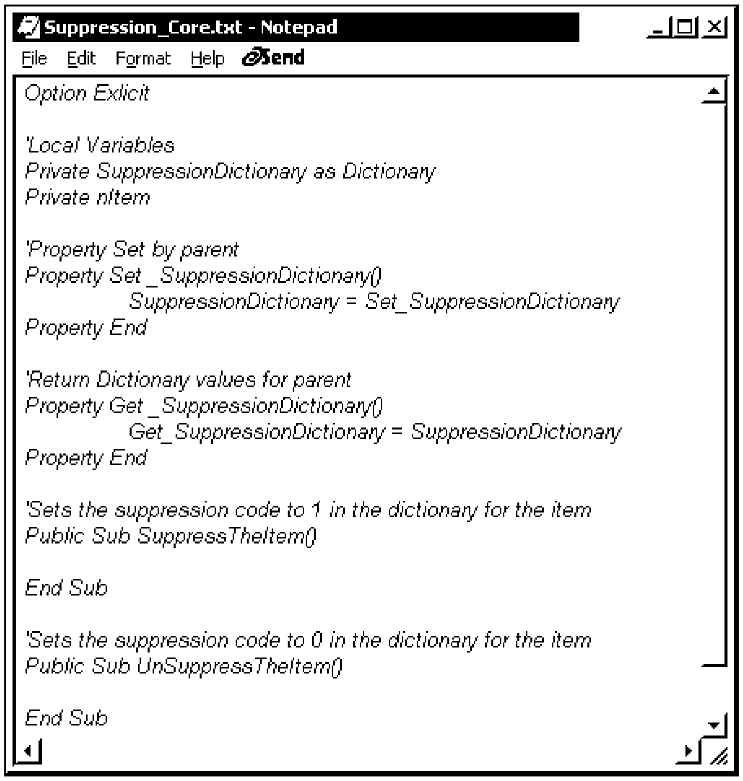
```
'Declare all variables
Option Explicit

'List external dependenceies
Dim AFSCore As AFSCore
Dim Assignments as AFSAssignments
Dim WindowsFileScripting As WindowsShellObject
Dim S_CORE as SuppressionCore

'Local variables
Dim ErrorsCollection(nItem, Error_Message, Error_Code) as Collection
Dim SuppressionDictionary(nItem, SuppressionCodes) as Dictionary

'Maintain State
Static Original_Assignments()

'Set item values by parent
Property Set _AssignmentsDictionary(Assignments)
        Assignments(Assignments) = Set_AssignmentsDictionary(Assignments)
End Property

Property Set _ErrorsCollection(nItem, Error_Message_Error_Code)
        ErrorsCollection(nItem, Error_Message_Error_Code) =
        Set_ErrorsCollection(nItem, Error_Message_Error_Code)
End Property

'Return nItem and its value to the parent
Property Get _SuppressionDictionary(nItem, SuppressionCodes))
        Get_SuppressionDictionary(nItem, SuppressionCodes) =
        SuppressionDictionary(nItem, SuppressionCodes)
End Property

Public Sub Error_Processing()

        ''If nothing changed then no need to process anything
        If Assignments() = Original_Assignments () Then Exit Sub

        'Local Variables
        Dim nError_F

        'Pass the empty collection object to the EP_CORE
        EP_CORE.Dictionary = Me.Dictionary

        'Now set all item's Error State

        nError_F= Fix_Date(Assignments.Dictionary(5)))
        If nError_F > 0 Then
            Call EP_CORE.AddError("5", "I005", 5)
        End if

        'Now retrieve the collection from the EP CORE
        SuppressionDictionary = EP_CORE.Dictionary
End Sub

'Stateless Functions
Public Function Fix_Date(s as String)

End Function
```

Figure 6.9: Pseudo-code for *Suppression* Feature Implementation.


ErrorProcessing_Core.txt - Notepad
File Edit Format Help Send

```
Option Exlicit

'Local Variables
Private ErrorsCollection as Collection
Private nItem

'Property Set by parent
Property Set _ErrorsCollection()
            ErrorsCollection = Set_ErrorsCollection
Property End

'Return collection value for parent
Property Get _ErrorsCollection()
            Get_ErrosCollection = ErrorsCollection
Property End

'Add Error to collection
Public Sub AddErrors(ErrorItem, ErrorMessage, ItemNumber)

End Sub

'Clear error for the item
Public Sub RemoveError()

End Sub

'Clears all errors in the collection
Public Sub ClearAllErrors()

End Sub

'If the item is suppressed then remove the error from collection
Public Sub RemoveErrorForSuppressItem

End Sub
```

Figure 6.10: Pseudo-code for *Error Processing* Core Implementation.



```
Option Exlicit

'Local Variables
Private SuppressionDictionary as Dictionary
Private nItem

'Property Set by parent
Property Set _SuppressionDictionary()
            SuppressionDictionary = Set_SuppressionDictionary
Property End

'Return Dictionary values for parent
Property Get _SuppressionDictionary()
            Get_SuppressionDictionary = SuppressionDictionary
Property End

'Sets the suppression code to 1 in the dictionary for the item
Public Sub SuppressTheItem()

End Sub

'Sets the suppression code to 0 in the dictionary for the item
Public Sub UnSuppressTheItem()

End Sub
```

Figure 6.11: Pseudo-code for *Suppression* Core Implementation.

6.7 Plug the fine-grained components into AMS

The last and final part of creating the component was to integrate all six components into one unit that performed *Input Processing* in an integrated environment. Using standard configuration management and compiler directives, old code in AMS was disabled to integrate the new components. Since the code profiler provides all the relevant functions, it was rather simple to insert the *Input Processing* component. The Pseudo-code for integrating the Input Processing component is shown in Figure 6.12. The compiler directive

`InputProcessing_Evolution` is used to enable the new components and disable the old code, including the global variables declared at the program level. This compiler directive is set at the program level and can be turn off easily in case the testers report adverse side effects due to the new components. Note that `UNREADY()` array is not used in the refactored code. The new components (AFSCORE, *Assignments*, *Suppression* and *Error Processing*) are declared globally at the program level so they can be used by other sub programs. The return value of Assignments components is then passed to other sub programs such as the calculation engine. Likewise, the return values of *Error Processing* components passed to the *Error Processing* GUI and the return value of the *Suppression* component is passed to the Main GUI sub programs respectively. The integrated component is shown in Figure 6.14.

```
Global sI$(), nItem

#IF InputProcessing_Evolution = True Then
        Global AssignmentsDictionary(nItem, Value)
        Global ErrorsCollection(nItem, Error_Message, Error_Code)
        Global SuppressionDictionary(nItem, Value)
        Global AFSCORE as AFSCORE
        Global Assignments as Assignments
        Global ErrorProcessing as ErrorProcessing
        Global Suppression as Suppression
#ELSE
        Global nError_F, UNREADY()
#END IF

Sub InputProcessing()

        #IF InputProcessing_Evolution = True Then

                'Assignment Component
                Set Assignments.sI$() = sI$()
                Call Assignments.Assignments
                AssignmentsDictionary(item,Value) =
                        Get Assignments.Dictionary(item,Value)

                'Error Processing Component
                Set ErrorProcessing.Dictionary(item,Value) = Dictionary(item,Value)
                ErrorProcessing.ErrorProcessing
                ErrorsCollection(nItem, Error_Message, Error_Code) =
                        Get ErrorProcessing.Collection (nItem, Error_Message, Error_Code)


                'Suppression Component
                Set Suppression.Dictionary(item,Value) = Dictionary(item,Value)
                Set Suppression.ErrorsCollection(nItem, Error_Message,
                Error_Code)=ErrorsCollection(nItem, Error_Message, Error_Code)
                Suppression.Suppression
                SuppressionDictionary(item,Value) =
                Get Suppression.Dictionary(item,Value)

        #ELSE
                'Following code will use, set or change global variables declared above
                'Items are called recursively
                For nItem = 1 to 450
                        Call Item_Hub_Code(i)
                Next nItem
        #END IF
End Sub
```

Figure 6.12: Integrating Assignments, Error Processing and Suppression Components.

6.8 Verify results

We performed a regression test of the *Input Processing* code to gather data pre and post evolution of these three features. We then compare the results to make sure we have not broken anything during the evolution process. The regression was automated by using GUI testing tools, and the AMS system has a built-in regression utility that sends the output to an ASCII text file. This text file was compared pre and post evolution to ensure that no side affects were introduced. In addition, the testers perform several tests to ensure proper working of the *Input Processing*.

6.9 Reuse

The Input Processing component is integrated in the WEB AMS in exactly the same fashion as the desktop AMS. The return value from the collection and dictionary objects is used by Microsoft server-side scripting language VBScript® because the WEB version of AMS uses VBScript® as opposed to the desktop version of AMS that uses VB. VBScript and VB behave similarly as far as integrating the *Input Processing* components are concerned. *Input Processing* components were deployed on the web server so the user interface layer can use *Assignments, Error Processing* and *Suppression* features. AFS CORE component was deployed on both servers namely, the web server and the application server. AFS CORE is also used by other product lines on all servers.

AFS Product lines	Components	Deployment
AMS (Desktop)	*Assignments, Error Processing, Suppression, Error Processing Core, Suppression Core and AFS CORE*	Desktop machines
AFS WEB	*Assignments, Error Processing, Suppression, Error Processing Core, Suppression Core and AFS CORE*	Web Server
AFS WEB	*AFS CORE*	Application Server
DTS and DTS WEB	*AFS CORE*	Web Server and Application Server
Sdev	*AFS CORE*	Web Server and Application Server

Figure 6.13: Resuing Fine-Grained Components in AFS Product Lines.

6.10 Measure Results

To measure the success of our methodology, we perform a validation against the evolution reasons. To reiterate, there were three primary reasons why we wanted to evolve the *Input Processing* into a component-based solution.

6.10.1 Solving the system-locking problem

The component-based implementation is a linear solution. In all three features, core items are evaluated first and then each of the items is individually evaluated. In addition, the communication between items is not done via the global variables. This communication is explicitly replaced by implementation of the feature relationship code discussed earlier. The original design was recursive in nature with no explicit condition to stop the recursion. The recursion was stopped implicitly by setting global variables or arrays that became error prone as more and more items with complex hierarchy and relationships were introduced. Replacing the recursive design with a linear design solved the system-locking issue.

6.10.2 Cost of adding a new item

The average time to add a new item and code all the relevant Assignment, *Error Processing* and *Suppression* logic took 3 days prior to applying the evolution methodology. After evolution, we collected data on adding 4 new items and the average time spent was about 1.25 days. The steps for adding a new item are as follows (Table 6.14):

Steps – Pre-Evolution	**Steps – Post-Evolution**
Add Item to Master File Table	Same
Add Item to Data Dictionary and assign its properties	Same
Create, Initialize and Assign Global Variable for Assignment	Code GetValue in Assignment Component.
Setup UNREADY Array	Not Needed
Code dependent items using recursion for *Error Processing* (mixed with Assignment and *Suppression* code)	Code *Error Processing* Component, but each item has its own spot rather than mixed with other items. Also, core items may also evaluated first but not necessarily.
Use Error Flag from Dependent Items to generate Errors	Errors are added to a collection. No global variables are needed.
Add Error Text into look-up tables	Same
Code dependent items using recursion for *Suppression*	Code *Suppression* component just like the *Error Processing* component.
Set *Suppression* code array	*Suppression* code array is automatically a part of interface of *Suppression* component.
Cost of debugging is high due to implicit communication and poorly implemented recursive routine	Virtually no debugging is necessary but must understand component interfaces

Table 6.14: Steps for Adding a New Item.

6.10.3 Reusability between AMS and the web version of AMS

There were six resulting components from this evolution exercise. Assignment, *Error Processing*, *Suppression*, *Error Processing* Core, *Suppression* Core and AFS Core. AFS Core is used in all of the AFS product lines (comprising 4 different projects), since it contains basic routines such as rounding functions and file I/O etc. The other five components are used in the two platforms of the AMS software, the desktop and the Internet. Table 6.15 shows cost and benefit involved in reusability. The net cost of the evolution exercise using this methodology so far has produced no loss or gain, but it is to be noted that there are certain hidden savings that we have not been able to capture until now, such as effect on training of new hires. It is expected that average savings identified in rows 14-16 (See Table 6.15) will eventually result in more favorable savings. We are currently in the process of gathering the data.

Effort	**Cost (+) /Savings (-) (Measured in months)**
Cost of Mapping Features and Test-Cases	+1
Cost of identifying code using test cases and profiler	+1
Cost of Refactoring	+2
Cost of Developing *Error Processing* Component	+1
Cost of Developing *Suppression* Component	+1
Cost of Developing Assignment Component	+1
Cost of Developing AFS Core Component	+1
Cost of Configuration Management	+1
Cost of Testing	+2
Cost of Training and Documentation	+1
Savings from solving system-locking problem (from 10/1/01 to 1/2/03)*	-1
Saving from improvement in adding a new Item (from 10/1/01 to 1/2/03)*	-1
Savings from improved architecture (reduced global variables, more explicit communication and better understanding of features)	N/A (data is being gathered**)
Savings in reusing AFS Core in 4 projects. Including cost of testing and integration.	-4
Savings in reusing *Suppression*, Assignment and *Error Processing* Component in desktop and Internet version of AMS. Including cost of testing and integration.	-6
Net Cost (+)/Savings (-) (from 10/1/01 to 1/2/03)*	**0**

Table 6.15: Budget analysis for input processing project.

* Calculated as opportunity cost, i.e. time we would have spent otherwise (from 10/1/01 to 1/2/03)

** This data will be reflected as cost of training a developer before and after the change. Due to challenging economic environment AFS has not hired a new trainee as of 1/2/2003. Thus, we will be gathering this data as AFS starts hiring new trainees.

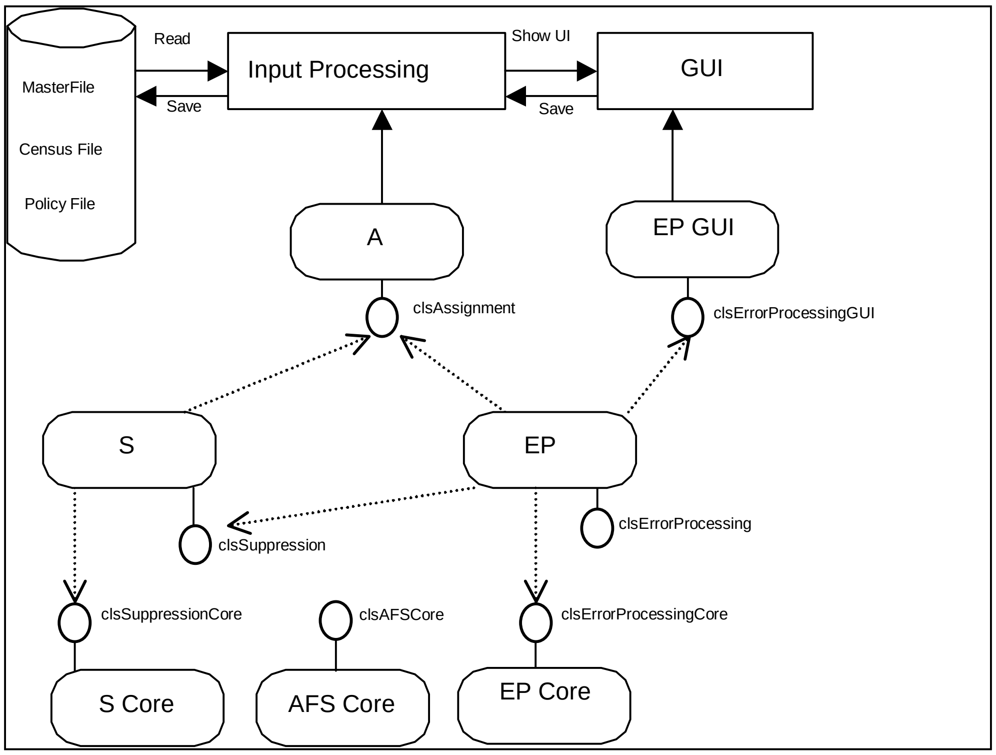


Figure 6.14: *Input Processing* Component Infrastructure.

6.11 Summary

We applied our ten-step methodology to identify and refactor the code to create reusable input processing component at AFS. We applied our feature model to identify the feature implementation. The original Input Processing feature represented the fully interacting feature implementation discussed in Section 4.1.2.3. The Assignment, Error Processing and Suppression code were intermingled. The feature relationships between the three features was identified by analyzing the implicit communication using techniques described in Section

4.1.5.2. The *input-processing* feature is comprised of Assignments, Suppression and Error Processing features. We created six reusable fine-grained components namely, clsAssignments, clsSuppression, clsErrorProcessing, clsSuppressionCore, clsErrorProcessingCore and clsAFSCORE, using our fine-grained component model. The clsSuppressionCore and clsErrorProcessingCore components support clsSuppression and clsErrorProcessing respectively. We plugged the components using compiler directives into AMS. We then verified the evolution reason by running regression pre and post evolution, reusing the component and fixing the system-lock problem. AMS and WEB-AMS shared all six components. AFSCORE component is being shared in all four AFS product lines. Using our budget model, we monitored and reported the cost/benefit of the entire effort. Our methodology shows a break-even in terms of costs and benefits incurred so far. There are some implicit benefits such as better understanding of features and reduced global variables, for which we are still collecting data. We consider our evolution initiative a success in applying our methodology to the Input Processing project. We will discuss lessons learned, our contributions and future work in the next chapter.

7 Conclusions

This chapter discusses the lessons learned in developing and applying our methodology, contributions made, and what avenues could be taken for future work.

7.1 Lessons Learned

In this section, based on our case study, we evaluate the benefits and limitations of our methodology.

7.1.1 Methodology Applicability

Although our methodology is programming language-independent and does not depend on specific code profiler tools, several factors affect the applicability of our methodology. These factors are as follows: organization's product-lines, maturity of software process, type of legacy system and refactoring choices.

Our methodology has been applied and tested in a scenario where there was one primary legacy system and other product-lines were being developed from scratch. In addition, the existing legacy system was experiencing maintenance costs of certain key features which were visible to the end-user. Furthermore, these features were also common across product-lines. While it is certainly possible to apply our methodology and refactor problematic feature implementation even in the case when these feature implementation are not common across product-line,. we argue that businesses will earn most return on investment (ROI) under the scenario when refactored feature implementation can be reused across multiple product lines. In AFS' case, we found that Input Processing feature implementation was experiencing high maintenance cost which when refactored was reused across two product AMS lines namely, the desktop and web-enabled.

Organizations that are looking to apply our methodology must have a mature software process in place. Maturity of software process can be determined by various factors such as CMM level, ISO 9001 level, availability of regression test process and ability to track costs of making changes to the legacy system. CMM level II and above, and ISO 9001 level I and above, both recommend availability of regression test process and ability to track costs when making changes. It is certainly important for organizations to be certified in CMM or ISO 9001, we suggest that our methodology will work best when there is a mature software process in place that has ability to perform regression test with each release and has ability to track costs of making changes.

Our methodology has been applied and tested in a function-based system with lots of global variables, functions and subroutines. While the methodology steps are programming language independent and type of legacy system independent, our methodology has not been applied on an object-oriented (OO) or a real-time legacy system. While it is possible to identify feature implementation of an OO system using a source code profiler, there are several complicated issues in OO such as polymorphism, overloading and inheritance that could result in same feature implementation of two features which would further complicate refactoring. Likewise, our methodology has not been tested on the systems with multiple thread of execution commonly found in real-time systems. We acknowledge the fact that our methodology will have to be customized in addressing evolution of OO and real-time legacy systems.

Once feature implementation is identified, our methodology provides a template for identifying lines of code that need refactoring. In addition, this template also provides the properties (Get and Set) for the refactored component. However, it is up to the developers to refactor the code into a component using the best possible design. Our methodology will work the best when the refactored unit has a lower maintenance cost itself.

7.1.2 Sensitivity relative to Average Coverage (C)

Not all features are an ideal candidate for this methodology. Using domain knowledge and enterprise initiatives, it is possible to identify features that either are a good candidate for reuse or have maintenance problems.

In Table 4.10, we have shown whether to continue with the methodology or not based upon the *Threshold* (T). Based upon the data collected from AMS (see Table 7.3) we believe that generally the average coverage (C) determines how sensitive a particular feature is for evolution. The end-users and testers identified the feature that needed evolution for our AMS case study. However, it is quite possible that selecting a feature for evolution the candidate features (note that the feature must still satisfy the law of two as discussed in Section 4.1.2.7) the decision for evolution is solely based on C. In this section, we will discuss how sensitive the case study Threshold values are. We found that the *Input Processing* had the following values for Threshold, Neighbouring Feature (K) = 3, Feature Implementation (FI) = 17 and Average Coverage (C) = 80%. Analyzing the feature/function matrix of *Input Engine* of AMS we found following:

If we reduced the C to be about 50% we found that K was increased to 8 and FI was increased to 19. If we reduced the C to be about 25% we found that K was increased to 21 and FI was increased to 99. Both the above finding indicates that we are dealing with less cohesive code, hinting that evolution would take longer (for a more detail analysis see Section 7.1.3). If C was increased to be greater than 90% we found that K was decreased to 2 and FI decreased to 1, indicating a trivial case. K=0, FI=0 and C=100% represent the CORE. The data is shown in

Table 7.1 and Figure 7.1. Note that Figure 7.1 shows C on the x-axis (represents the first data point of 10%, 2 means 25%, 3 means 50% and so on) and a scale on the y-axis which is used to plot FI and K. The values shown in

Table 7.1 are shown plotted in Figure 7.1. We can see from Figure 7.1 as C decreases the distance between FI and K increases indicating more time to evolve these features and likewise as C increases distance between FI and K decreases indicating lesser time to evolve. We found that C for our case study was best suited around 80%, which is the fourth data point in Figure 7.1 (suggesting an optimum for AMS's Input Engine given the budget in terms of time/resources). Note that different parts of the legacy system may be sensitive to C, a more empirical study is suggested as a part of future work (see Section 7.3.2).

Feature	K (Neighboring Feature)	FI (Functions)	C (Avg. Coverage)
Log	125	35	10%
Constraints	99	21	25%
Callback	8	19	50%
Input Processing	3	17	80%
Import/Export	2	1	90%
CORE	0	0	100%

Table 7.1: Coverage Sensitivity Data in AMS.

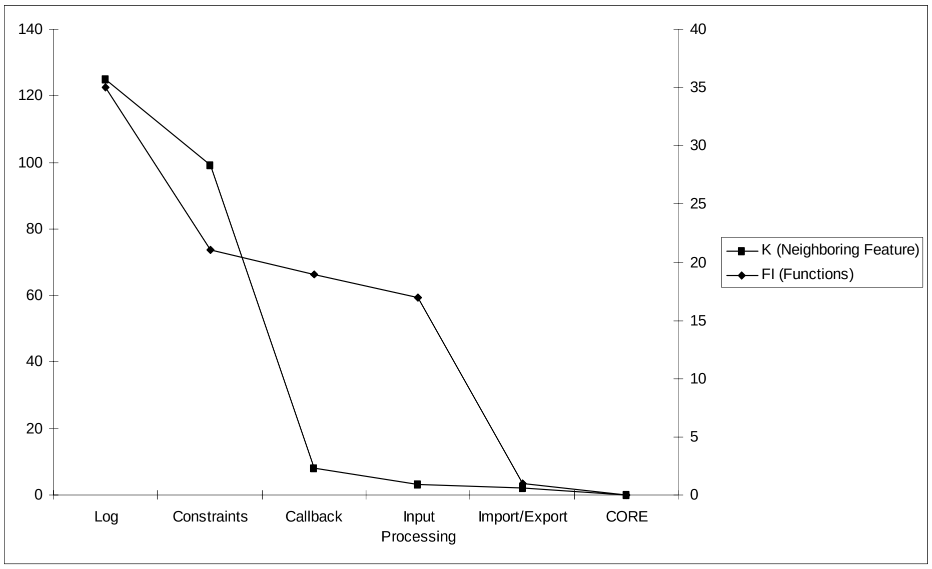


Figure 7.1: Coverage Sensitivity in AMS.

7.1.3 Selecting Evolvable Features

In the case of AMS, if a feature is spread out across many functions, and if the code execution is below 80% using selected test cases within each of the functions, the feature is not a good candidate. For example, the primary function of AMS is to integrate executive benefits and life insurance using complex non-linear algorithms. Life insurance acts as an asset to fund the executive benefits. There are many legal-, accounting-, insurance- and benefits-related constraints, which play an important role in the asset/liability match within AMS. Such constraints are scattered throughout AMS, and make up less than 20-25% in any given function. Our experience tells us that the constraints themselves will certainly not be good candidates for evolution because they do not change frequently, and they probably cannot be reused in other AFS product lines. Good candidates are those features that change often, are concentrated in fewer functions, depend on or share global variables as a means of communication and can be reused across product lines.

Our methodology provides several heuristics to avoid feature interaction issues by identifying closely related features. If two feature implementations are highly correlated, then it is certain that these features are intertwined, and a rewrite is probably warranted.

Heuristics on features that are evolvable can be further elucidated by close analysis of Table 7.2. This table displays the results of running the profiler with regression test cases, after the test cases have been mapped to the features. Each row represents a particular function; each function has been mapped to a pre-defined function type (SS, DF, DD, SSF – see List of Acronyms and Glossary), as indicated in the rightmost column. For example, f_1 is of the type Shared Stateless Function (SS). Each column represents a regression test case Tn (while our methodology supports multiple test cases representing a single feature we omit this detail for the purpose of this discussion); each test case is mapped to a feature (FE_n). For example, T1 is mapped to FE_1. The numeric value in each cell is the percentage of coverage for the specific feature by that function. For example, f_1 has 100% coverage for FE_1 in T1.

The concept of threshold is essential in this analysis of evolvability. For a given feature-function relationship, the threshold is based on the number of functions (FI), the number of neighboring features (K), and the percentage of coverage (C) for the feature in each function. The coverage level must be significant; we have selected 80% as the minimum for evolvability based upon our experience with the AMS. There are two special cases: (1) The trivial case: K = 1 and F = 1; the feature and the function are coterminous; (2) C = 0: zero coverage, hence there is no feature-function relationship. Optimum values for evolvability, for a given legacy system (LS), might be K = 3, FI = 17, C = 80%, with no cross-cutting and no trivial cases.

Another important concept in this analysis is traceability. This is the ability, using a code-profiling tool, to traverse the implementation code path in order to identify feature-function relationships. Traceability can be used to examine three types of relationships: (1) function-to-feature; (2) feature-to-function; (3) feature-to-feature. The NuMega ® True Time Code Profiler [56] was used in this analysis.

The following describes feature or function categories in terms of evolvability. Each category (in upper-case below) corresponds to a colored section in Table 7.2.

Test Cases	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	
Function\Features	FE$_1$	FE$_2$	FE$_3$	FE$_4$	FE$_5$	FE$_6$	FE$_7$	FE$_8$	FE$_9$	FE$_{10}$	FE$_{11}$	FE$_{12}$	Type
f$_1$	100	100	100	0	0	0	0	0	0	0	0	0	SS
f$_2$	80	80	90	0	0	0	0	0	0	0	0	0	DF
f$_3$	100	100	100	0	0	0	0	0	0	0	0	0	SS
f$_4$	70	75	80	0	0	0	0	0	0	0	0	0	DD
f$_5$	100	100	100	0	0	0	0	0	0	0	0	0	SS
f$_6$	100	100	100	0	0	0	0	0	0	0	0	0	SS
f$_7$	100	100	100	0	0	0	0	0	0	0	0	0	SS
f$_8$	0	0	0	80	90	0	0	0	0	0	0	0	DF
f$_9$	0	0	0	100	100	0	0	0	0	0	0	0	DF
f$_{10}$	0	0	0	0	0	11	12	0	0	0	0	0	SSF
f$_{11}$	0	0	0	0	0	33	44	0	0	0	0	0	SSF
f$_{12}$	0	0	0	0	0	12	15	0	0	0	0	0	SSF
f$_{13}$	0	0	0	0	0	15	15	0	0	0	0	0	SS
f$_{14}$	0	0	0	0	0	22	22	0	0	0	0	0	SS
f$_{15}$	0	0	0	0	0	32	38	0	0	0	0	0	SS
f$_{16}$	0	0	0	0	0	15	15	15	15	15	15	15	SS
f$_{17}$	0	0	0	0	0	22	22	22	22	22	22	22	SS
f$_{18}$	100	100	100	100	100	100	100	100	100	100	100	100	SS
f$_{19}$	100	100	100	100	100	100	100	100	100	100	100	100	SS
f$_{20}$	100	100	100	100	100	100	100	100	100	100	100	100	SS
f$_{21}$	0	0	0	0	0	32	38	12	56	89	66	63	DD
f$_{22}$	0	0	0	0	0	22	22	34	52	23	43	34	DD
f$_{23}$	0	0	0	0	0	32	38	22	44	33	45	32	DD

Table 7.2: Evolable Features.

Functions that are shared stateless (SS) and have 100% coverage for all test cases can be shared in a common library, as CORE. Functions 18-20 in Table 7.2 meet these requirements.

If a feature is contained within a few functions, and if the code execution is above 80% using selected test cases within each of the functions (above threshold), this feature is a good candidate to be EVOLVABLE. Features 1-3, implemented in Functions 1-7, and Features 4-5, implemented in Functions 8-12, meet these requirements.

A CROSS-CUT feature is dispersed in too many functions and the coverage values are below threshold. Features 6-7 illustrate this category. A cross-cut feature is not recommended for evolution because the impact of change can create several unforeseen errors and the cost will exceed the benefits.

The shaded area (Feature 8 through Feature 12, and function f_{16} through f_{23}) represents the inverse of cross-cutting, another type not recommended for evolution, NON-EVOLVABLE; while the features are implemented in a reasonable number of functions, the functions implement a large number of features, and the coverage values are below threshold.

The ZERO COVERAGE category includes all cells in the feature-function matrix, which have no coverage. These are the white cells in Table 7.2.

Our experience with AMS features is shown in Table 7.3 (partial listing). The *Input Processing* feature that was evolved, had a K=3, FI = 17 and C = 80%. We also found that there were features within AMS that were implemented within one function only and no other feature was implemented in that function. An example of such a feature is importing data from another database. Cash value calculation is an example of non-evolvable feature because its FI is shared by nine other features in a single function which means the impact of change will be unfavorable. Likewise, a cross-cutting feature, like a constraint to a non-linear equation solve, is also not a favorable candidate for evolution because it is dispersed in so many different features. Finally, we discovered CORE of 42 functions as shown in Table 7.3.

K	FI	C		
Number of Neighboring Features	Number of Functions	Avg. Coverage	Feature	Heuristics
3	17	80.00	Input Processing	Evolable
1	1	100.00	Import	Trivial
10	1	100.00	Cash Value Calc	Not-Evolvable
1	54	35.00	Solve Constraint	Cross-Cut
-	42	100.00	-	Core

Table 7.3: Heuristics (partial listing).

7.1.4 Sorting Feature Function Matrix

When regression test suite is run with a profiler, the amount of data can be overwhelming. AMS has over 250 test case files in its regression test suite. When the profiler generates the feature function coverage matrix, it is not sorted and the feature-function coverage data is scattered as shown in Table 7.4.

Unsorted	Features											
Functions	FE_1	FE_2	FE_3	FE_4	FE_5	FE_6	FE_7	FE_8	FE_9	FE_{10}	FE_{11}	FE_{12}
f_8	0	0	0	80	90	0	0	0	0	0	0	0
f_9	0	0	0	100	100	0	0	0	0	0	0	0
f_{10}	0	0	0	0	0	11	12	0	0	0	0	0
f_{11}	0	0	0	0	0	33	44	0	0	0	0	0
f_{12}	0	0	0	0	0	12	15	0	0	0	0	0
f_{13}	0	0	0	0	0	15	15	0	0	0	0	0
f_{14}	0	0	0	0	0	22	22	0	0	0	0	0
f_{15}	0	0	0	0	0	32	38	0	0	0	0	0
f_{16}	0	0	0	0	0	15	15	15	15	15	15	15
f_{17}	0	0	0	0	0	22	22	22	22	22	22	22
f_{18}	100	100	100	100	100	100	100	100	100	100	100	100
f_1	100	100	100	0	0	0	0	0	0	0	0	0
f_4	70	75	80	0	0	0	0	0	0	0	0	0
f_{21}	0	0	0	0	0	32	38	12	56	89	66	63
f_1	100	100	100	0	0	0	0	0	0	0	0	0
f_5	100	100	100	0	0	0	0	0	0	0	0	0
f_{21}	0	0	0	0	0	22	22	34	52	23	43	34
f_{19}	100	100	100	100	100	100	100	100	100	100	100	100
f_2	80	80	90	0	0	0	0	0	0	0	0	0
f_6	100	100	100	0	0	0	0	0	0	0	0	0
f_7	100	100	100	0	0	0	0	0	0	0	0	0
f_{20}	100	100	100	100	100	100	100	100	100	100	100	100
f_{23}	0	0	0	0	0	32	38	22	44	33	45	32

Table 7.4: Unsorted Feature Function Matrix.

When we look at the evolution threshold, we must cluster the data into group of related features and their FIs. We make use of ***RankSort*** algorithm to cluster the feature-function matrix. Ranksort allows sorting over multiple columns. A detailed description and analysis of Ranksort can be found in [107]. We have also implemented RankSort algorithm in our utility that was discussed in Chapter 5. The data shown in Table 7.4 is sorted using RankSort that then resembles Table 7.5.

Sorted	Features										
Functions	FE_1	FE_2	FE_3	FE_4	FE_5	FE_6	FE_7	FE_8	FE_9	FE_{10}	FE_{11}	FE_{12}
f_1	100	100	100	0	0	0	0	0	0	0	0	0
f_2	80	80	90	0	0	0	0	0	0	0	0	0
f_3	100	100	100	0	0	0	0	0	0	0	0	0
f_4	70	75	80	0	0	0	0	0	0	0	0	0
f_5	100	100	100	0	0	0	0	0	0	0	0	0
f_6	100	100	100	0	0	0	0	0	0	0	0	0
f_7	100	100	100	0	0	0	0	0	0	0	0	0
f_8	0	0	0	80	90	0	0	0	0	0	0	0
f_9	0	0	0	100	100	0	0	0	0	0	0	0
f_{10}	0	0	0	0	0	11	12	0	0	0	0	0
f_{11}	0	0	0	0	0	33	44	0	0	0	0	0
f_{12}	0	0	0	0	0	12	15	0	0	0	0	0
f_{13}	0	0	0	0	0	15	15	0	0	0	0	0
f_{14}	0	0	0	0	0	22	22	0	0	0	0	0
f_{15}	0	0	0	0	0	32	38	0	0	0	0	0
f_{16}	0	0	0	0	0	15	15	15	15	15	15	15
f_{17}	0	0	0	0	0	22	22	22	22	22	22	22
f_{18}	100	100	100	100	100	100	100	100	100	100	100	100
f_{19}	100	100	100	100	100	100	100	100	100	100	100	100
f_{20}	100	100	100	100	100	100	100	100	100	100	100	100
f_{21}	0	0	0	0	0	32	38	12	56	89	66	63
f_{22}	0	0	0	0	0	22	22	34	52	23	43	34
f_{23}	0	0	0	0	0	32	38	22	44	33	45	32

Table 7.5: Sorted Feature-Function Matrix.

7.1.5 Reusable Components and Web Applications

Section 2.2.2 discussed the importance of the Internet in evolving legacy systems. At AFS, while we already had our flagship product AMS in operation, we were developing its web counterpart which presented challenges and opportunities. In our search to reuse components across our product lines, we found that a web applications forces a system to be composed of smaller stateless units which in turn forces a monolithic legacy system to be decomposed into smaller more manageable units. Web applications are typically are comprised of three tiers namely web server, application server and database server as shown in Figure 7.2. In evaluating reusability between AMS and AMS-Web, we found that between the two product lines there were four common indirect features specifically; *Input Processing*, Reporting, Business Logic Calculation and Data Access as shown in

Figure 7.2. While all four features were converted into reusable units using the methodology; *Input Processing* feature was evolved using the methodology, Reporting and Business Logic features were evolved via wrapping and, Data Access feature is part of CORE.

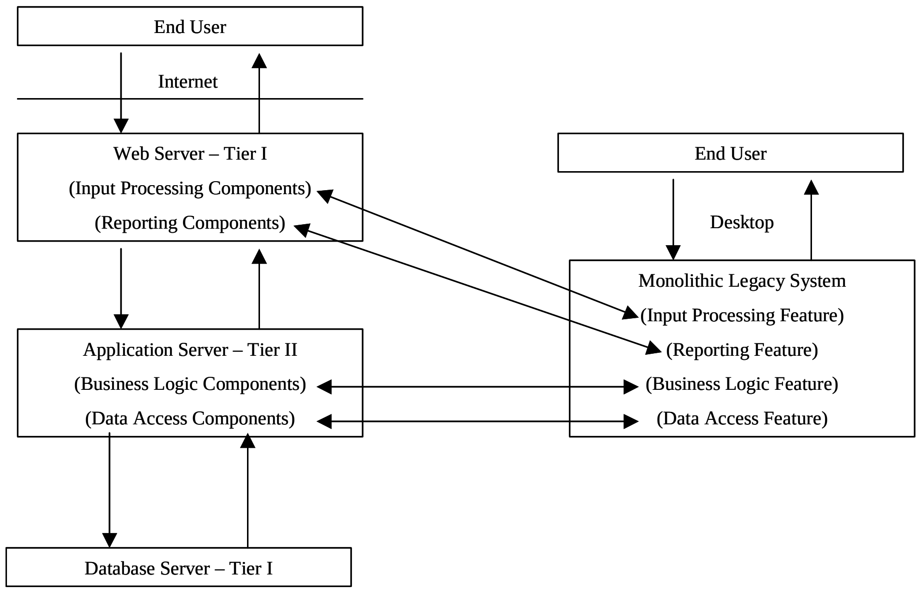


Figure 7.2: Reusable Components Between Desktop and Internet Application.

7.1.6 Issues In Reusing Components Across Product Lines

While reusing components is an excellent idea across product lines, we found several inherent problems:

7.1.6.1 Configuration Management

Configuration management and version control are key issues and must be addressed carefully when a component is shared across product lines. It is also important to decide how many versions of reusable components are going to be maintained and supported. We found that configuration is more complicated in a product line context for two reasons:

1. A change must be considered not from the point of view of a single product, but in terms of keeping the changed component used by all of the products that currently employ it.
2. It is more likely that it will be necessary to maintain separate versions of reusable components, as opposed to simply supplying the most recent one, as may suffice in one-at-a-time development.

At AFS we found that strong, centralized architectural control is key to product line development, but so is management of change and evolution. In the development phase, the architect answers to a single set of products, and their needs are often complex. But in a product line, the architect answers to users of all versions of the system, and keeping the product line intact is more important than making changes to accommodate a single product's needs. In order to address the above-mentioned issues we created a team of software developers whose main job was to manage the versions of the shared components.

7.1.6.2 Deployment

Given that fine-grained components needed to be integrated for use within product lines, there are two technology alternatives that we considered:

1. For the local component model (for the use within AMS) we compared Microsoft’s Active X and Sun’s JavaBeans.
2. For distributed component model (for the use within AMS-WEB) we compared Microsoft’ COM (COM/DCOM/MTS/COM+) and Sun’ Enterprise JavaBeans (EJB).

We decided to use Microsoft COM for deploying both AMS and AMS-WEB product lines for the following reasons:

a. Integrating Active X components in our existing enterprise framework is easier than JavaBeans since not all our developers know Java programming language.

b. Our enterprise framework and legacy code uses COM as an underlying technology.

c. We did not want to deal with the bridge technology that would try to connect COM and JavaBeans [62]. For example, it is possible to raise events from JavaBeans and catch them in Active X components.

However, there are known problems with this integration specifically when handling exceptions.

d. Our users use Windows operating system at-large so interoperability is not an issue for AFS.

e. EJB offers several nifty and powerful features but is quite complex. Because not all our developers are Object-Oriented literate and we have already spent so much in Microsoft technologies, we feel that COM is a better choice for our server-based implementations as well. We feel that COM is simple. Thus, we chose COM for deploying components on the server side as well.

Table 7.6 summarizes the comparisons. Although, it made good technical and business sense to use COM as an underlying technology for deploying our fine-grained components we faced the following challenges in implementing the reusable units in AMS-WEB:

1. Our fine-grained components had to be deployed in MTS (Microsoft Transaction Server) when using with AMS-WEB. While the installation of MTS was simple, it did add another layer of complexity for our developers. We also found that *Input Processing* components under MTS' performance was slower than the AMS version.

2. COM’s deployment data is stored in the Windows registry; this is cumbersome and presents several faults as far as portability is concerned. Registry entries can be exported and imported if a server is moved but this is an error prone procedure.
3. Server-based COM components cannot be configured as stateful or stateless. They're always stateless. Remote clients use DCOM to invoke methods in COM objects on server machines. For access from Internet-enabled clients via HTTP, the COM Executive is loaded into Internet Information Server (IIS) the Web server built into Windows 2000 OS. We had to enhance our fine-grained component model to maintain state.
4. The AMS-WEB versions of components had to be configured for security in the application server. The security settings required setting several parameters in the registry of Windows 2000 OS, which is cumbersome and errorprone.

Properties	EJB	COM
Component Language	Java only	VB, C++, Java, C# and Others
Platforms	All	Windows 2000
Middleware Vendors	30+	Microsoft
Legacy Integration	RMI/JNI, CORBA, Connectors	COM TI, MSMQ, OLE DB
Deployment method	XML descriptor file	GUI and Registry
Protocol	Any	DCOM
Component Persistence	Serialization	No
Stateless components	Yes	Yes
Stateful components	Yes	No
Persistent components	Yes	No
Method-granularity transactions	Yes	No
Middle-tier load balancing	Most vendors	Supported via app. server
Middle-tier data caching	Some vendors	No
Queued components	No	Yes
Single-vendor solution	No	Yes
Middleware comes with OS	No	Yes
Development tools	Choice of many	Microsoft Dev Studio

Table 7.6: EBJ and COM Comparison.

7.1.6.3 Training

Fine-grained components deployment on the server and configuration management required more training to our developers than we had originally anticipated. While we were successfully able to train our developers on our methodology and fine-grained components, we found that server-based deployment of fine-grained components required extra effort. We did not include training time for deployment on the server in our budget analysis model because this time was not an extra step required by our methodology. In other words, in absence of our methodology this step would have to be performed anyway.

7.1.7 Global Variables

We were pleasantly surprise to observe following:

1. Better definition of feature-based global variables
2. Reducing global variables when feature relationships are shared and required.

7.1.7.1 Explicit Definiton

We found that in all cases that the legacy system is full of global variables. These global variables are declared and initialized in numerous functions. Typically, FE_1 would set the value of a global variable g_1 and FE_2 or subsequent feature(s) may either use (shared/required feature relationship) or change (alter feature relationship) $g_{1.}$ This implicit communication is common in legacy system. As a side effect of our methodology, we renamed the global variables based upon the neighboring features and their relationships. Providing better name to the global variables implied two benefits:

1. It suggests explicit relationships among features thereby reducing confusion.
2. Changing the state of the global variable can provide some clue regarding the impact it may have on related feature(s).

Table 7.7 illustrates the concept discussed above. The new names of global variables G1 and G2 depends on feature relationships between the two features that they have been involved in.

Old Global Variable Name	Program	FE_1	FE_2	Feature Relationship	New Global Variable Name
G1	Declare	Set	Use	Shared	G1_FE1_FE2_S
G2	Declare	Set	Change	Altered	G2_FE1_FE2_A

Table 7.7: Global variable naming convention.

7.1.7.2 Reducing Global Variables

We found that the program does not need to declare the global variable if the relationships between the features is either shared or required. The global variables can be encapsulated within the feature-based fine-grained components, and be accessed as needed by the rest of the program.

7.1.8 Availability of Regression Tests

While we have no empirical studies to show that most systems have regression test suites to measure stability between releases, such test suites are important from a business perspective [10]. An informal survey of seven legacy systems revealed that all of them had adequate regression test suites. We therefore believe it is reasonable to assume that most businesses either have these test suites (although they may not refer to them as such) or are generating these test suites manually each time a new release is scheduled.

7.1.9 Automating Tasks

To instrument the source code we compiled the source code image with *TrueCoverage*™. Since the regression testing is already being done in batch mode, it was easy to get the instrumented output to compare against all 250 regression test cases. However, these instrumented images were in a *TrueCoverage*™ specific file format. *TrueCoverage*™ does provide an automated way to export the specific file format. We had to manually export each file into a standard file format (comma-separated values) just to import into a spreadsheet tool for further analysis. This process needs to be better automated and the *TrueCoverage*™ vendor has indicated that future releases will have this functionality.

7.1.10 Dead code and coverage

We assume that a comprehensive set of regression tests is available for identifying code associated with the given feature(s). In our case study, we found that even after executing all test cases, not all of the code associated with *Input Processing* was executed. We believe that the unexecuted code contained either hidden features or is dead code. For example, 12 routines were never called at all. Also, nearly 17% of the code was not executed in the original code. We put all the unused code in a separate file and documented it. Incremental feature evolution gives us the implementation of core (*AFS Core*).

7.1.11 Core and Reducing Dependence on Variables:

After refactoring the AFS Core component, we manually identified the parameters for each of the 42 stateless functions. Since AFS Core is being used in four AFS product lines, this effort was worthwhile because these 42 functions do not create any side effects and use no global variables. In addition to AFS Core, there are two additional supporting core components: *Suppression Core* and *Error Processing Core*. These supporting core components encapsulate the worker functions and states (i.e., business logic) used by *Suppression* and *Error Processing* components. The supporting core components are created to provide flexibility in future evolution if any underlying data structure is changed for managing *Suppression* or *Error Processing*. For example, *Error Processing Core* contains functions to add, remove, and edit errors to a collection object. In the future, if the collection object is replaced by an array or another structure, such encapsulation will allow AFS to change only the working functions and the interface for the business logic will remain the same. Therefore, each of the six components has well-defined interfaces with no side effects. Their properties and methods are categorized explicitly using `GetValue/SetValue`.

7.1.12 Performance

In refactoring the recursion into linear functions, the performance of AMS was unaffected. We observed a 2% decrease in execution time once *AFS Core* was introduced. We attribute this improvement to the removal of global variables and in-line code.

7.1.13 Component Interface Issues

Our methodology initially created components with too many interfaces. To resolve this issue, we used a Collection Object provided in the VB programming language to hide the list of these variables. Different programming languages may require a different implementation of methods and properties. Furthermore, the collection object was divided into two basic types, `GetValue/SetValue` with the parameter of the variable name as an index key.

7.1.14 Measuring Success

The true measure of a successful evolution methodology is in reduced future maintenance costs. We have only just begun the long-term task of collecting maintenance data on the refactored system. We found that the features we evolved for AMS as components can be reused in two platforms, both desktop and Internet. Although reuse involves integration, configuration management, and testing costs, the savings on development costs made this exercise highly successful. As briefly shown in Table 6.15, the net estimated cost of this project is one month’s salary for the AFS development team. Once long-term cost reductions are factored in, the resulting savings will be favorable. The performance of the refactored system is acceptable and it no longer freezes during input. Also, AFS is now using AFS Core in all four of its product lines (an unexpected side effect).

7.2 Contributions

In this dissertation, we have made the following contributions: First, an incremental methodology to evolve legacy code is developed that improves the maintainability of evolved legacy systems. Second, the technique describes a clear understanding between features and functionality, and relationships among features using our feature model. Third, the methodology provides guidelines to construct feature-based reusable components using our fine-grained component model. Fourth, we bridge the complexity gap by identifying feature-based test cases and developing feature-based reusable components.

7.2.1 Incremental Evolution Methodology

Our ten-step methodology is incremental in nature and can provide rapid results. Our methodology provides “exit-points” in case the developers/testers are not satisfied with any of the results. For example, heuristics discussed in this dissertation identify which features are good candidate for evolution and which ones are not. We have identified input and output criteria for each step of our methodology, and at any step if a parameter is missing the developer can stop the whole process without any side effect. Although we have not done so, this methodology can also be used to evolve multiple features at the same time.

7.2.2 Feature Model

Our feature model defines features in a way that considers evolution in mind. Our feature model provides a guide to identify the feature implementation within the source code. We have identified and provided solutions to various cases when features interact with each other and reveal the same code for feature(s) to be evolved. We provide a simple clustering technique to group test cases, which represent same indirect features. In addition, our feature model provides ample description on thorny issue of feature interaction and provides an intuitive way of addressing the issue by considering feature relationships. Another major contribution of our feature model is a technique to associate multiple test cases with a single feature and develop a feature/function relationship. Finally, our feature model forms the basis of providing heuristics to the user by providing insights on sub-features, feature implementation, CORE, neighboring features and evolution threshold.

7.2.3 Fine-grained Component Model

Feature Implementations (FIs) can be refactored into fine-grained components, which can then be reused across multiple product lines. Our fine-grained component model is simple to use and has minimum requirements in the sense that it allows the developers to provide better definition to the FI and the variables involved in invoking that FI. We provide guidelines for evolving a FI into a fine-grained component: A component’s properties can simplify complicated scenarios such as when a code profiler results in same code for more than one feature in a function.

Finally, our fine-grained component model reduces global variables if feature relationships are shared or required.

7.2.4 Complexity Gap

We bridge the complexity gap in two ways. First, we map the regression test cases to the features and create a feature/function matrix. This matrix is used to select evolvable features. Regression test cases reflect the end-user feature; they are already focused so it is not necessary to collect execution traces on all inputs or to divide the input sets into *invoking* or *non-invoking* category as proposed by other researchers. We suggest that regression test cases are the best choice for the input cases because regression test cases contain information regarding features. They can be used as a common entity between the end-user and the software team. Second, the fine-grained components are feature-based components as they implement a specific feature or a group of related features. Since these components are focused on specific features, we believe that they can represent end-user requirements in a much more explicit way thereby bridging the gap between user expectations and what the software can provide.

7.3 Future Work

American Financial Systems, Inc. has nearly ten years of longitudinal data on their legacy system. We are currently expanding our evaluation to model the development costs in adding, modifying, or removing system features. Now that AFS has refactored parts of its legacy system, we will carefully monitor their development and maintenance teams to determine the impact of the software evolution methodology. We hope that other organizations will be inspired by the success of AFS to carefully evaluate their regression test suites to determine the feasibility of creating their own reusable fine-grained components.

The work carried out in this research effort opens the door for several interesting as we now describe.

7.3.1 Metrics

The methodology presented in this dissertation can be further enhanced by including several metrics such as: quantifying the relationships between test cases, quantifying the relationship between the features, quantifying impact analysis when a feature implementation is altered, quantifying the relationship between the fine-grained components and the legacy system and, quantifying the complexity gap.

7.3.2 Threshold

The concept of threshold can be studied with respect to following:

1. Different parts of the legacy system: We applied our methodology in the *Input Processing*. The methodology can be applied to different areas of AMS (See Appendix B for more detail) such as:
    a. Calculation Engine

   b. Output Engine
   c. Utility functions such as import and export

2. Legacy systems with different architecture: Legacy systems can have various architectural styles such as pipe and filter, event based, implicit invocation, layered, repository oriented, table driven, blackboard and object oriented [29]. We applied our methodology in the *Input Processing* of AMS, which appears closest to that of pipe and filter. While our methodology is architectural-style independent, it will be interesting to see the results of applying our methodology to the legacy system with various architectural styles.

7.3.3 Multi-threaded features

Our methodology has been developed and tested with single-threaded features. Our case study included features with relationships shared, required, and altered. Typically, multi-threaded features implement competition and conflicting relationships [26]. While we can certainly identify features with these relationships, our methodology does not provide enough guidance in converting such Feature Implementations (FIs) into fine-grained components. More work needs to be done in this area.

7.3.4 Extending the evolution manager utility

The evolution manager utility that we discussed in Section 4.6 can be extended. The main purpose of our utility is to show that a Feature Implementation (FI) can be identified. This FI typically needs local and global variables, which in turn becomes the component’s interface. We evaluated various code profilers and used their output of in the utility. Extending the utility to automatically import the information on static and dynamic slicing from various code profiles would be a good improvement. The main purpose of our evolution manager utility is to identify FI in terms of Lines of Code (LOC) and variable involved. While we are able to show the power of relational database by modeling LOC and variables used, this utility does not yet provide any insights into refactoring. Extending the utility to provide refactoring insights can be helpful. There are several other enhancements that can be made to this utility such as maintaining release versions and adding more reports.

7.3.5 Object-Oriented Systems

The case study presented in this dissertation uses a legacy system that has lots of global variable and is not object oriented. It will be interesting to apply our technique on an object-oriented system with complex class hierarchies. There are several code profilers that are available for most of the object-oriented languages such as Java or C++. Indeed, many legacy systems are object-oriented. These legacy systems can benefit from our methodology if our methodology is extended to include object-oriented systems. Code profilers can gather information about the legacy system’s classes that implement features. These classes will have to be then analyzed using our feature model. Refactoring of these classes can result in some core classes that can then be shared. It appears that analyzing object-oriented system will be quite challenging and it presents its own research issues. We leave this interesting problem to be solved as part of future work.

7.3.6 Systems whose source code is unavailable

At this time it may appear far-fetched but our methodology could be extended to include techniques to analyze systems whose source code is not available. There are legacy systems whose source code is either lost or unavailable for one or more reasons. While our methodology uses source functions to provide heuristics on evolvable features, this can be changed to simply extract features by running regression test cases. Essentially, input and output can be compared to identify systems behavior. By comparing the input against the full regression test suite, the features can be identified and the system can be executed to generate output. This behavior can be statistically studied to identify features of interest. Once features are fully understood they can then be rewritten. CBSE techniques can be used to integrate newly created components into legacy system whose source is not available.

7.3.7 Real-time Systems

Our methodology has been applied in a legacy system that is used in integrating executive benefits with life insurance. The AMS legacy system is by no means a real-time system. If the real-time systems are instrumented to collect the output data then desired features could be studies, identified and refactored as needed.

This instrumentation can be in many forms such as code profiler, debug lines or log files. Many real-time systems have advanced logging capabilities. That could be used to analyze and refactor features of interests.

7.3.8 Tools to manage feature evolution

Figure 4.27 outlines the data model used as a basis for our formal model. We chose a relational model because it is intuitive and easily applicable in our context. The formal model provides a theoretical foundation to our techniques. While we have successfully applied our methodology, we believe that it is lacking a tool to manage the feature evolution process. This tool can be built from the idea presented in our formal model. The formal model is actually based upon the data model. This data model contains useful information regarding shared function and data. It can select the functions and data shared among features and can automatically copy them into the relational model as the developers are using this methodology. As this database grows, it will provide meaningful information for traceability and future maintenance. We believe that several tools can be developed that can use the data model presented in this dissertation.

7.3.9 Tools to automate selection of test cases

We have presented two techniques to select the regression test cases in our case study. These techniques are usage of clustering and textual pattern analysis. Future work can include development of some tools that can automate this task. Essentially, these tools can analyze the pool of heterogeneous data that is either part of regression test suite or is associated with the documentation that supports regression test cases. These tools can group the related test cases (thus related features) based upon certain rules.

7.3.10 Extending the budget analysis model

The budget analysis model can be extended to include several other variables that may be interesting from a project management perspective. While the budget analysis presented in the conclusion Section 6.10 includes variables that are sufficient to suggest that our approach is indeed worthwhile, it can certainly be extended to include costs more accurately using COCOMO or COCOTS model [15][17][109][130]. Furthermore, the budget analysis presented in Section 7.1 can include several other line items such as opportunity cost.

7.3.11 Extending the component and formal model

Our component model considers two features at a given point in time. See Figure 4.15, which shows view of the function being analyzed as f_x, f_y etc. We feel that there is an opportunity to analyze more than two features at a given time. This would increase the complexity of the analysis but we believe that the component model could be extended to use more than two features at a time.

Similarly, the formal model could be extended to include several metrics such as provided in Wong et al. [125]. These metrics could provide several interesting views such as what is the relationship between components properties and variables used in a feature prior to its evolution or a *scattering index* indicating how many functions a particular feature is scattered in and what is its relationship with other features within the function.

7.3.12 Using our methodology with tools other than code profilers

We have developed and used our methodology with source code profilers. There are other tools that could be used to collect the data we want, such as compilers with symbolic debug information or user-defined instrumentation.

7.3.13 Application of our methodology for program understanding

Although we argued rather rigorously in the related work section that motivation of our work is software evolution rather than program understanding, we believe that our methodology can be used to understand program as well. Using regression test cases and code profiler, the execution paths can be studied to understand which part of the program is being used more than the other and so on. Similarly, functions involved in a particular feature can be traced and watched for testing and debugging purposes.

7.4 Summary

We discussed lessons learned, our contributions and future work in this chapter. We presented heuristics for selecting the features that are best suited for evolution. We discussed the concept of threshold that allows us to select sutable candidates for evolution. Threshold consists of a function of number of neighbouring features, number of functions and average coverage. We discussed the importance of RankSort and clustering. Upgrading existing desktop applications into web-based application allows us to identify features that can be reused using our methodology. There are several issues in reusing components among applications such as configuration management, deployment and training personnel. Legacy systems that have large number of global variables can benefit from our methodology in two ways, first our methodology provides better definition of feature-based global variables and, second it reduces global variables when feature relationships are shared and required. Our methodology assumes availability of regression test suites and code coverage profilers. We believe that this constraint is not severe. We successfully applied our methodology in a large industrial application. Our contributions consists of an incrementally evolution methodology, a feature model and a fine-grained component model. These models are supported by formal model and budget analysis model. We bridge the complexity gap in two ways; first by mapping the regression test cases to the features and creating a feature/function matrix, second by creating feature-based fine-grained components. We have provided several avenues for future research such as developing metrics to measure feature/function relationships, collecting information and calculating threshold for various parts of the legacy system, extending our methodology for multi-threaded features, enhancing the evolution manager utility, applying our methodology for OO systems and extending our models.

We hope that we have convinced the reader that our methodology is easily applicable and measurable, incremental in nature and has solid theoretical foundation.

List of Acronyms

ADT	Abstract Data Type
AFS	American Financial Systems, Inc.
AMS	AFS Master System
AOP	Aspect-Oriented Programming
API	Application Programming Interface
C	Average Coverage
CBSE	Component Based Software Engineering
COTS	component-off-the-shelf
DD	Dependent Data
DF	Dependent Function
FE	Feature in a Legacy System
FE_i	a feature of Legacy System that is exercised when k_i is executed
FGC	Fine-grained component model
FI	Feature implementation as defined in the feature model
FI_i	An implementation of FE_i in LS
FOCS	Feature Oriented Classification of System
FODA	Feature Oriented Domain Analysis
FOP	Feature Oriented Programming

G	A set of global variables in a Legacy System
GUI	Graphical User Interface
g_i	The set of global variables involved in FI_i
k_i	A set of test cases such that $k_i \in T$
K	Neigboring Features
LOC	Lines of Code
LS	Legacy System
MAP	Mining Analysis of Product Lines
OAR	Options Analysis Re-engineering
SEI	Software Engineering Institute
SS	Shared Stateless Function
SS_i	The set of shared stateless functions in FI_i
SSF	Shared State-Full Function
T	Threshold
V	The set of local variables in FI_i
v_i	The set of local variables directly affected by FI_i

Glossary

χSuds. A tool developed by Telecordia Technologies where program features in the source code are tracked down to files, functions and lines of code.

Altered Relationship. When a feature's state (global data, object or implementation) is altered by another feature then there is an altered relationship between features.

Architectural reconstruction. Architectural reconstruction is the process where the "as-built" architecture of an implemented system is obtained from the existing legacy system.

Aspect Oriented Programming (AOP). An approach in which cross-cutting concerns that appear throughout numerous modules of a system implementation are identified and then integrated into the primary modularization to create a final working system.

Assignments. AMS's Input Processing's sub-feature which converts user input from strings to types such as Integer, Single, Double, or Array.

Base-line Architecture. Specific caller-callee sequence within a program, and is unlikely to be reusable into another components.

Black Box Technique: A binary executable form of the component is available and there is no extension language or API.

Budget Analysis Model: The budget analysis model presents the cost and the benefit of applying the methodology.

Clustering Analysis. It is the organization of a collection of patterns (usually represented as a vector of measurements or a point in multidimensional space) into clusters based on similarity.

Coarse-grained software evolution. Evolution focused on large-scale structural issues of a software system, such as global control structure, synchronization and protocols of communication between components.

Code refactoring. Improving the design of existing **software code** without altering the behavior.

Code profiler. A tool for analyzing software code which performs functions such as identification of performance bottlenecks and verification that code changes have improved performance.

Complexity gap. The gap between the problem domain the solution domain.

Component. A software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

Component Based Software Engineering (CBSE). An approach to software design that utilizes components as the core structural elements.

Component model. A model which defines specific interaction and composition standards.

Composed relationship. It shows how a feature is composed of several sub-features. An example of a composed relationship is that a bank account consists of savings and checking accounts.

Configuration management. Application of technical and administrative controls to characteristics, change processing, and implementation of configuration items in a software system.

CORE. Shared Stateless function(s) that are executed 100% of the time for all features then we define that function to be part of CORE. Such functions are candidates for a shared library.

Cross-cutting. Mean that a function can implement many features and these features share the same code/data.

Data Model. The data model is used to trace feature relationships, interactions and component evolution of a legacy system.

Dependent Data. An FI may be dependent on the data that is updated by another FI. This can be local or global variable.

Dependent Function. An FI may be dependent on a function that is part of another FI.

Evolution Manager Utility. Utility used in recording and tracing the methodology steps.

Error Processing. Error Processing is a sub-feature of AMS's Input Processing feature that validates item values.

External Dependencies. SSF, CORE and other components can be called "out" of the fine-grained component to access any data needed via this interface. External Dependencies is a list of external program/component declaration within a fine-grained component.

Feature. It is a group of individual requirements that describes a unit of functionality with respect to a specific point of view relative to a software development life cycle.

Feature Engineering. The area of study that addresses the understanding of features in software systems and then defines a set of mechanisms for carrying a Feature from the problem domain into the solution domain (thereby reducing the complexity gap).

F**eature Implementation.** It is the set of statements (including data) within all functions that execute when that feature is invoked. The feature is invoked by one or more test cases.

Feature Interaction. Features must interact with each other to provide wider system functionality. When features interact with each other, they have an "effect" on the system. This effect can be positive or negative.

Feature Model. helps to identify where features are located within the legacy system, how features are related to other features, and how they interact with each other.

Feature Oriented Domain Analysis (FODA). A method of system analysis which provides a generic description of the requirements of a class of systems and a set of approaches for their implementation, based on the feature set of the system.

Feature Relationships. Feature relationships refine the concept of interaction by providing specification through calling sequence among features sharing data/functions.

Fine-grained components. Components whose interaction is clearly specified by the interfaces provided by each feature interface.

Fine-grained component model. It provides guidelines to extract feature specific code/data.

Formal Model. It provides the theoretical foundation for our evolution methodology. The formal model is supported by the data model.

Function Point Analysis. The basic notion of this discipline is that the functionality of a software project can be objectively estimated independent of the implementation.

Gray Box Technique: Source code of a component is not modified but the component provides its own extension language or Application Programming Interface (API).

Input Processing. It validates and prepares data from user inputs (also called items) so AMS can perform complex calculations to generate various reports.

Item. Field within the AMS system.

Law Of Two, The: If a feature can be used in another system, its implementation becomes a candidate for reuse.

Legacy Code. A system or application which continues to be used because of the cost of replacing or redesigning it, often despite its poor competitiveness and compatibility with modern equivalents.

Legacy System. Any software system that is currently in operation is considered legacy system.

Methodology: Used for evolving legacy system's features by exercising each feature with its associated test cases using code profilers and similar tools, feature implementations can be located and refactored to create reusable fine-grained components.

Neighboring Features (K). Number of features interacting within a function

Opportunity Cost.

Problem domain. User expectations and concerns, pertaining to software functionality.

Property Get: It is a way to retrieve the values of local or global variables from the component.

Property Set. It is a way to pass these variables to the refactored FI/component.

Regression Testing. Part of the test phase of software development where, as new **modules** are integrated into the system and the added functionality is tested, previously tested functionality is re-tested to assure that no new module has corrupted the system.

Required Relationship. When a feature is required to be present for other features to function is known as Required Relationship.

Requirement Engineering. Requirement Engineering is the discipline that is focused on providing a concise, consistent, unambiguous, and complete definition of the problem domain.

Shared Relationship. When a group of feature share resources (global data, objects or other implementation) with other feature(s) then a **shared relationship** among features exists.

Software Architecture. The way a system is designed; the way components fit together.

Software Evolution: See Methodology.

Software Reconnaissance: Implies "*preliminary survey of enemy terrain*" where software program is considered as an enemy whose secrets must be extracted.

Solution domain. Developer concerns regarding the creation and maintenance of software development life cycle artifacts such as components.

State-full Function. A state-full function can be shared between two features. It maintains state and is used as a means to communicate by features.

Stateless Function. A stateless function can be shared between two FIs and does not retain any state.

Suppression. Suppression is a sub-feature of AMS's Input Processing feature that either shows or hides an item in the user interface based upon the input for another item.

Threshold: Optimal number of neighboring features, the number of functions and the average coverage percentage within the function. It provides heuristics on whether to continue with the evolution methodology or not.

Traceability. Ability of a code-profiling tool to trace the source code implementing a specific feature.

White Box Technique. Access to source code allows a component to be significantly rewritten to operate with other components.

References

Appendices

Appendix A: AFS Master System

- The AFS Master System (AMS) is used daily by hundreds of top insurance producers successfully competing in the supplemental benefits market. Key features include:

- True 32-bit power of the latest Windows operating systems, featuring user-friendly interface, high-quality graphic output, drag-and-drop input from other applications, and unparalleled presentation capabilities

- Quick preparation of illustrations and proposals to support new sales

- Ability to re-project plans to adjust for changing plan assumptions

- Easy entry and control of all census data and convenient handling of an infinite number of lives with the Census Manager database, allowing sorts on any census item and the option to zoom into individual census records

- Power to tailor plans to clients' complex needs by combining all types of benefits and insurance products in a single, composite illustration

- Flexibility to make as many changes as needed in any design variable in any illustration, including tax brackets, cost of money, salary increase rates, term rider amounts, split dollar premium bonuses, etc.

- Easy navigation among appropriate sales concepts, advice on accounting issues and case design, and guidance on the impact of each selection with hypertext help and the package design wizard

- Handles and integrates all types of executive benefits sales, including SERPs, True Deferral Plans, 162 Bonus Plans, Split Dollar Plans, Death Benefit Only Plans, and all types of Group Carve-Out Plans

- Provides highly flexible Group Carve-Out modeling

- Solves for appropriate amounts of insurance financing, taking into account MEC status projections, multiple-target cash values, rollout solves, maximum withdrawal solves, year-by-year benefit tracking and various combinations of solves

- Demonstrates emerging shortfalls in benefit funding, and solves for additional insurance

- Saves historical values of a case so re-projections can utilize the accumulated historical data

- Accounts for benefits and insurance amounts according to FAS 87, FAS 107, FAS 109, and APB 12

- Provides TAMRA analysis, MEC compliance with sophisticated MEC avoidance options, and in-force re-projections, including illustration of multiple in-force policies for the same individual

- Anticipates financial impact of a program through the use of partial mortality

- Runs universal, variable, traditional, and all types of interest-sensitive products concurrently on one system

- Controls and tests new policy outlay options

- Assesses and projects material changes, flags MEC status, and adjusts policy taxation

- Controls and updates flexible term riders; tests for re-projected amounts and limits

- Handles all forms of loans, partial surrenders, and changes in dividend options

- Individual Executive Reports - A wide range of reports on any individual on a given run

- Composite Reports - An equally wide range of composite reports on a given run

- Assumption Page and Report - Lists all assumptions made in the illustration Census and Master File.

- Benefit Report - Provides selected census information and a summary of insurance and benefit information for each plan participant and calculates group totals when applicable

- New Business Reports - Automatically uploads census data and policy information to the home office, feeding new business and policy administration systems

- Output to Access and Excel - Sends output directly to database tables and worksheets for further processing and formatting without having to parse and reformat ASCII text files

Appendix B: AMS Architecture

There are three sub-systems that constitute the AFS Master System ® (AMS):, the Input Engine, the Calculation Engine and the Output Engine. The Input Engine is an ActiveX executable, the Calculation Engine is also an ActiveX executable and the Output Engine is a standalone executable. In addition, Microsoft Access ® is used as the data repository. MS Access ® is used both to manage the user's data and as a communication vehicle between the three engines (see Figure B.0.1). ActiveX is part of AMS's COM (Component Object Model) technology. Thus by creating ActiveX components via VB, COM components are actually created at the same time. The Input Engine performs input data validation (along with other functionality such as Import, Export, Menus, /Census Manager) and "prepares" data for the Calculation Engine. The Calculation Engine performs calculations (see Figure B.0.1) and dumps the data into an MS Access ® Table. Through MS Windows ® API and a "polling" mechanism the Output Engine is instructed to generate reports. User Data is stored in the Master and Census Tables of MS Access ®.

Figure B.2 provides an overview of the interactions between the three entities (Input, Calculation and Output). There are two main communication vehicles that help the communication between entities; the Status Run Table and the Run Form.

Status Run Table is created by the Input Engine; it contains a variety of status information about the progress of calculation and printing. When the Calculation Engine is done with calculating, a status of “6” is posted to the record(s). When the Output Engine reads (polls) that status of “6”, it performs reporting and finally updates the status to “14” when done. There are many records in the Status Run Table in a given session. Both the Calculation and Output Engines operate asynchronously. Run Form is part of the Input Engine. The Input and Calculation Engines communicate via a “callback mechanism”. The Run Form displays messages to the user that are sent from the Calculation and Output Engines. When the Calculation Engine sends messages to the Run Form, they are sent via the “callbacks”. However, when the Output Engine sends messages to the Run Form, they are done via the Windows API (since the Output Engine is a standalone executable).

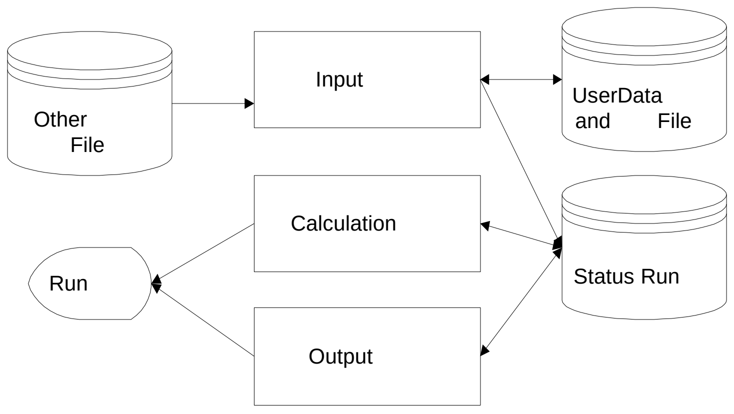


Figure B.0.1: Interactions Among the Input, Calculation, and Output Engines.

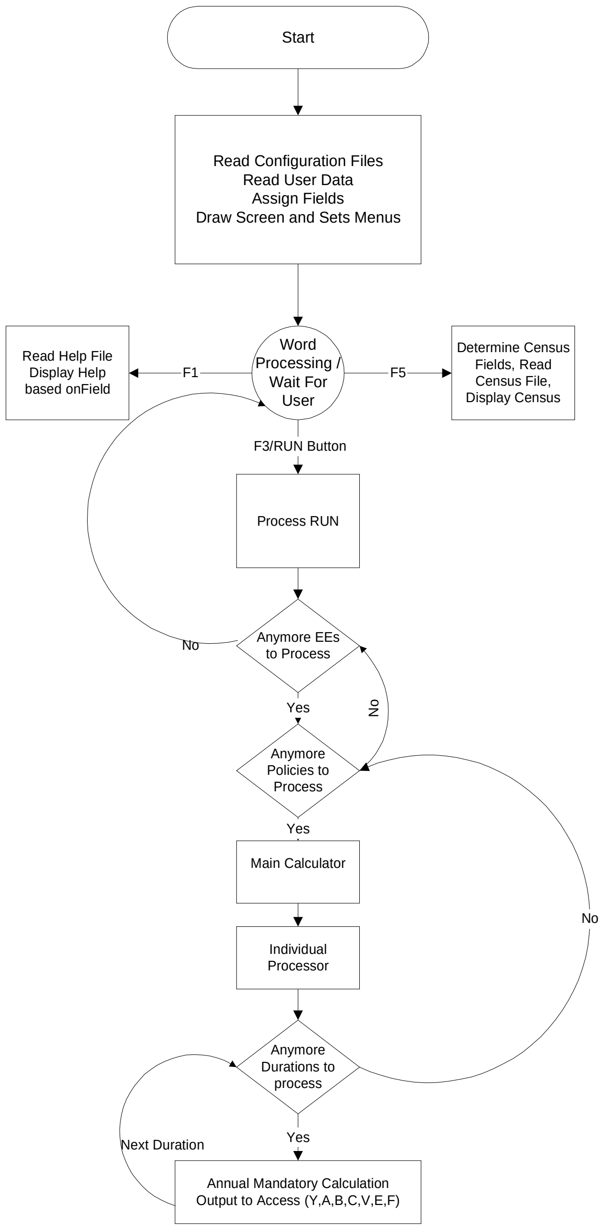

Start
Read Configuration Files
Read User Data
Assign Fields
Draw Screen and Sets Menus
Read Help File
Display Help
based onField
F1
Word Processing / Wait For User
F5
Determine Census Fields, Read Census File, Display Census
F3/RUN Button
Process RUN
No
Anymore EEs to Process
Yes
No
Anymore Policies to Process
Yes
Main Calculator
No
Individual Processor
Anymore Durations to process
Next Duration
Yes
Annual Mandatory Calculation
Output to Access (Y,A,B,C,V,E,F)

Figure B.0.2: AFS Master System – Calculation Processing.

Appendix C: AMS Regression Testing Utility

Overview

AFS Quality Control Analysis is the process of monitoring that interactions and interdependencies are maintained in proper working order throughout the implementation of controlled system changes. Simply put, this analysis verifies that changes between versions occur only where expected and do not adversely affect other areas of the system.

AMS provides a feature to help address this process through the Batch Processing Utility. The Batch Processing Utility offers an effective and timesaving method to generate a series of file-based calculations for a predetermined set of test cases. As part of your Quality Control effort, the Batch Processing Utility will provide the necessary information to help analyze the accuracy of the system through a representative sampling of control cases that illustrate the system's functionality. Figures C.1 and C.2 show how AMS regression tool is invoked via the AMS GUI and the regression tool's GUI respectively.

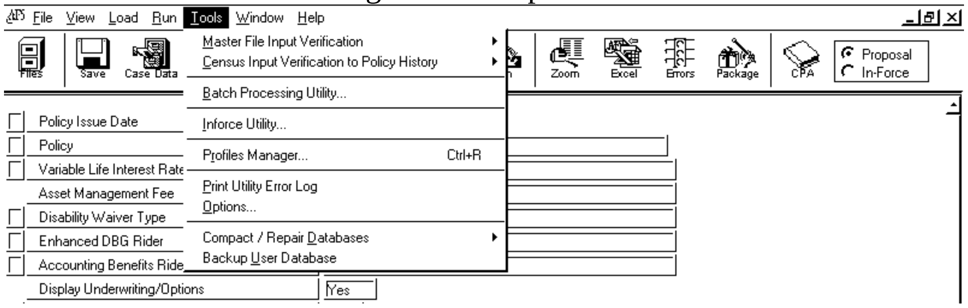
The Batch Utility contains the following features:

- Test cases are organized into a matrix.
- Output may be produced for all or some of the cases.
- Output can be sent to ASCII, Excel, Access, or to a Printer.
- Individual test files may be added or deleted.

- Groups of test files can be categorized and submitted by category to the Batch Processor for calculation.

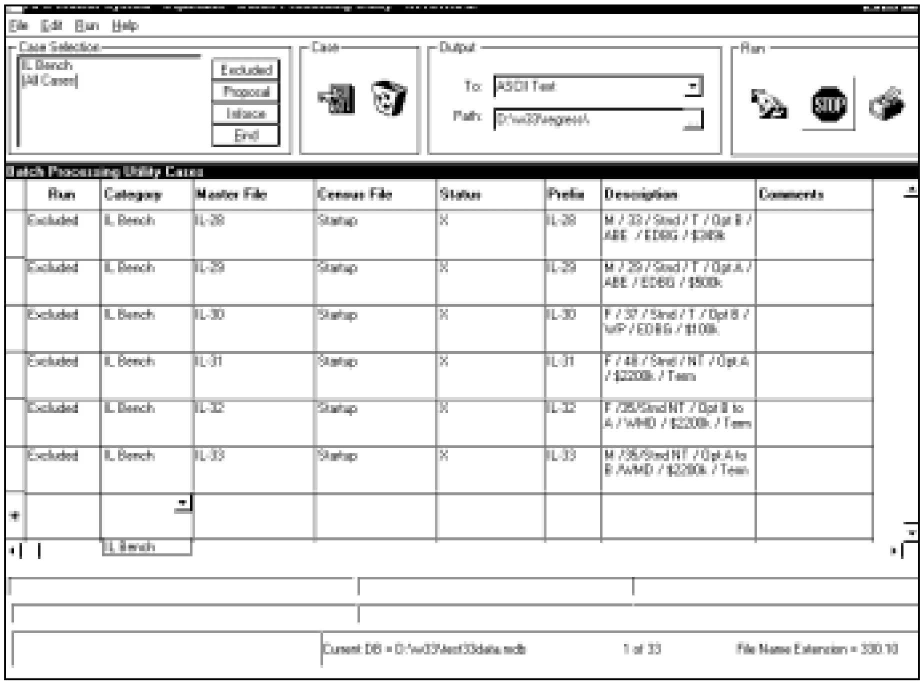
- The Matrix Report can be printed to document the results produced through the Batch processor.

- The Matrix Report includes the category, master file name, census file name, CVF file name, file prefix, file description as well as any user comments concerning the file output.


File View Load Run Tools Window Help
Files Save Case Data
Master File Input Verification
Census Input Verification to Policy History
Batch Processing Utility...
Inforce Utility...
Profiles Manager... Ctrl+R
Print Utility Error Log
Options...
Compact / Repair Databases
Backup User Database
Zoom Excel Errors Package CPA
Proposal
In-Force

Policy Issue Date	
Policy	
Variable Life Interest Rate	
Asset Management Fee	
Disability Waiver Type	
Enhanced DBG Rider	
Accounting Benefits Ride	
Display Underwriting/Options	Yes

Figure C.1: Invoking regresssion testing utility via AMS GUI.



To: ASCII Text
Path: D:\w33\regress\

Batch Processing Utility Cases

Run	Category	Master File	Census File	Status	Prefix	Description	Comments
Excluded	IL Bench	IL-28	Startup	X	IL-28	M / 33 / Stnd / T / Opt B / ABE / EDBG / $285k	
Excluded	IL Bench	IL-29	Startup	X	IL-29	M / 29 / Stnd / T / Opt A / ABE / EDBG / $500k	
Excluded	IL Bench	IL-30	Startup	X	IL-30	F / 37 / Stnd / T / Opt B / WP / EDBG / $100k	
Excluded	IL Bench	IL-31	Startup	X	IL-31	F / 48 / Stnd / NT / Opt A / $2200k / Term	
Excluded	IL Bench	IL-32	Startup	X	IL-32	F /35/Stnd NT / Opt B to A / WMD / $2200k / Term	
Excluded	IL Bench	IL-33	Startup	X	IL-33	M /35/Stnd NT / Opt A to B /WMD / $2200k / Term	
	IL Bench						

Current DB = D:\w33\test33data.mdb 1 of 33 File Name Extension = 330.10

Figure C.2: AMS regresssion testing utility user interface.

Build a Batch Processing Utility Matrix

Test File Creation:

Create and view each representative master, census and case file in your ‘working’ test database.

This ‘working’ test database will be the basis for your ‘Batch’ database. Ultimately, the ‘Batch’ database will not be used for any testing apart from Batch Processing. This strategy will help maintain the integrity of your ‘Batch’ database and will also bypass any confusion that may arise when the Batch Utility is used to create output for your file comparisons. In addition, as you create your test files, be aware of the report columns for these files. It is important to include appropriate and illustrative column sets.

File Import:

After each file is reviewed, import the representative master, census or case files into your ‘Batch’ database.

Only files that are in the active database may be added to the Batch Processing Utility Matrix. The 'Add' function will not create new master, census or case files, but rather include existing files in the matrix.

Add Files to a Batch Processing Utility Matrix:

Click on the Category drop down and type in a category name. Use the Category entry as an identifier for groups of cases. This categorization will become helpful if you decide to either include or exclude groups of cases from the batch run.
From the Master File Drop Down list, select the master file. If applicable, from the Census Master File Drop Down list, select the census file. If the case does not have a census, select ‘STARTUP’ from the drop down list.

Enter a five characters prefix for the test case. This prefix will be used if the test case is sent to Access, Excel, or ASCII. The Batch Processor takes the five characters prefix and appends information from the release and version information that is contained in the executable. For example; if the file prefix were ‘IL-01’ and the executable number was 33.0.11, the system would produce a file with the following name --- IL-01330.11 It is good practice to verify the initial file output.

<u>Note:</u>

An alternative to the method described above involves the ‘Add’ icon. This 'Add' function will not create new master, census or case files, but rather includes existing files in the matrix.

Only files that are in the active database may be added to the Batch Processing Utility Matrix. To add files that are not in the current test database, first import them into the active database.

The STARTUP census file is assumed as the default census for master files that are added.

Use the Category entry as an identifier for groups of cases. This is also helpful in excluding groups of cases from a batch run.

A file may be added to the matrix once per category.

Delete Files from a Batch Processing Utility Matrix:

Click the Delete icon.

Using the Record Selector, highlight the file(s) to be deleted.

For multiple file selection, hold down the CTRL key and click on each file that you want. To select a group of files that are next to each other, click on the first or last file of the group, and then hold down the SHIFT key while clicking on the file at the end of the group that you want to select.

Click OK.

In the Confirm Deletion window, click Yes to delete the cases.
You may click No to exit the delete option entirely or click Cancel to return to the Select one or more box and redefine the files to be deleted.

The record(s) corresponding to the deleted file(s) will be deleted from the Batch Processing Utility Matrix.

Deleting a file from the matrix does not delete if from the database.

You can add a file that has been deleted from the matrix back into the matrix at later time.

Find Entries in a Batch Processing Utility Matrix:

On the Tools menu, point to Batch Processing Utility.

In the Case Selection box, click the category that you want to find.

Click Find.

The Record Selector will move to the first entry in the matrix that has the specified category.

Print a Batch Processing Utility Matrix Report

The Matrix Report includes a detailed list of the cases in the Batch Processing Utility Matrix which has been sorted by category. The report includes any file descriptions and comments that have been entered into the grid.

On the Tools menu, point to the Batch Processing Utility.
On the File menu, click Print or Click on the 'Printer' icon.
The Matrix Report will be printed to the default printer specified in the AFS Print Utility Configuration window.

Direct Batch Run Output to Printer or File

On the Tools menu, point to Batch Processing Utility.

In the Output To box:

Click Printer to print the output to the default printer specified in the AFS Print

Utility Configuration window

Click Excel File to direct the output to an Excel spreadsheet

Click Access to direct the output to an Access database

Click ASCII Text to direct the output to a text file

If you choose to direct the output to an Excel file or to an ASCII text file, then in the Path box, specify the location of the output file. You may type in the full path name or use the Browse feature.

Run Batch Output:

Prior to a run, make sure that the Run ‘types’ are set correctly for your cases.
All cases with run types of 'In-force' and 'Proposal' will be included in the batch run. Cases with run type of 'Excluded' are not included in the run.

To select cases for a Batch Run:

On the Tools menu, point to Batch Processing Utility.

Individually or by category, set the Run Types for the Batch Run.

Direct the Batch Output

Click Submit Run.

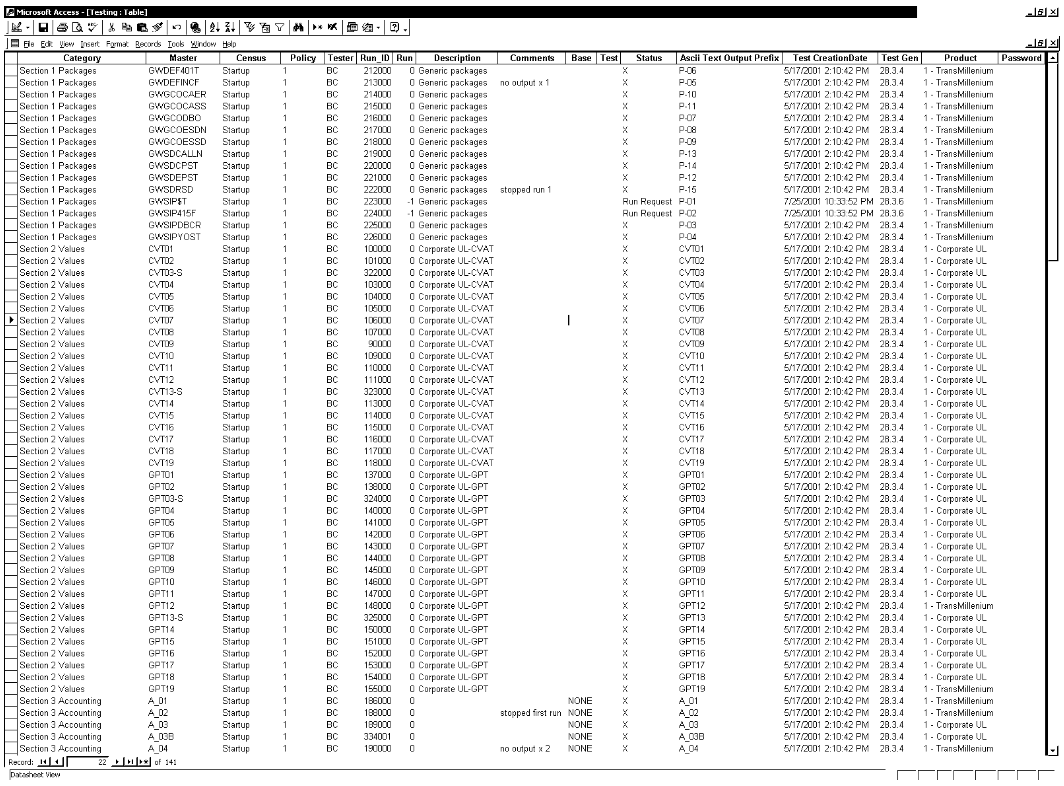
<u>Note:</u>

The Status box for each case will reflect where the case is in the run process.

An 'X' in the status box signifies that the case has not been selected for the run.

'Run request' signifies the case is currently being processed, while 'Scheduled' means that the case has been selected and has not yet been processed.

In a batch run, a case with run type of Proposal is run with a normal run (without a CVF file), while a case with run type of In-force is run with a CVF file. This is equivalent to the Normal Run (F3) and In-force Run (Shift-F3) options in the Run menu in the Master Control Panel. Figure C.3 shows partial AMS regression test suite. The AMS system can also run via command-line executing each of the records shown in Figure C.3.


Microsoft Access - [Testing : Table]
File Edit View Insert Format Records Tools Window Help
Record: 22 of 141
Datasheet View

Category	Master	Census	Policy	Tester	Run_ID	Run	Description	Comments	Base	Test	Status	Ascii Text Output Prefix	Test CreationDate	Test Gen	Product	Password
Section 1 Packages	GWDEF401T	Startup	1	BC	212000	0	Generic packages				X	P-06	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 1 Packages	GWDEFINCF	Startup	1	BC	213000	0	Generic packages	no output x 1			X	P-05	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 1 Packages	GWGCOCAER	Startup	1	BC	214000	0	Generic packages				X	P-10	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 1 Packages	GWGCOCASS	Startup	1	BC	215000	0	Generic packages				X	P-11	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 1 Packages	GWGCODBO	Startup	1	BC	216000	0	Generic packages				X	P-07	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 1 Packages	GWGCOESDN	Startup	1	BC	217000	0	Generic packages				X	P-08	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 1 Packages	GWGCOESSD	Startup	1	BC	218000	0	Generic packages				X	P-09	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 1 Packages	GWSDCALLN	Startup	1	BC	219000	0	Generic packages				X	P-13	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 1 Packages	GWSDCPST	Startup	1	BC	220000	0	Generic packages				X	P-14	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 1 Packages	GWSDEPST	Startup	1	BC	221000	0	Generic packages				X	P-12	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 1 Packages	GWSDRSD	Startup	1	BC	222000	0	Generic packages	stopped run 1			X	P-15	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 1 Packages	GWSIP$T	Startup	1	BC	223000	-1	Generic packages				Run Request	P-01	7/25/2001 10:33:52 PM	28.3.6	1 - TransMillenium	
Section 1 Packages	GWSIP415F	Startup	1	BC	224000	-1	Generic packages				Run Request	P-02	7/25/2001 10:33:52 PM	28.3.6	1 - TransMillenium	
Section 1 Packages	GWSIPDBCR	Startup	1	BC	225000	0	Generic packages				X	P-03	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 1 Packages	GWSIPYOST	Startup	1	BC	226000	0	Generic packages				X	P-04	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 2 Values	CVT01	Startup	1	BC	100000	0	Corporate UL-CVAT				X	CVT01	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT02	Startup	1	BC	101000	0	Corporate UL-CVAT				X	CVT02	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT03-S	Startup	1	BC	322000	0	Corporate UL-CVAT				X	CVT03	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT04	Startup	1	BC	103000	0	Corporate UL-CVAT				X	CVT04	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT05	Startup	1	BC	104000	0	Corporate UL-CVAT				X	CVT05	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT06	Startup	1	BC	105000	0	Corporate UL-CVAT				X	CVT06	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT07	Startup	1	BC	106000	0	Corporate UL-CVAT				X	CVT07	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT08	Startup	1	BC	107000	0	Corporate UL-CVAT				X	CVT08	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT09	Startup	1	BC	90000	0	Corporate UL-CVAT				X	CVT09	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT10	Startup	1	BC	109000	0	Corporate UL-CVAT				X	CVT10	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT11	Startup	1	BC	110000	0	Corporate UL-CVAT				X	CVT11	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT12	Startup	1	BC	111000	0	Corporate UL-CVAT				X	CVT12	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT13-S	Startup	1	BC	323000	0	Corporate UL-CVAT				X	CVT13	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT14	Startup	1	BC	113000	0	Corporate UL-CVAT				X	CVT14	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT15	Startup	1	BC	114000	0	Corporate UL-CVAT				X	CVT15	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT16	Startup	1	BC	115000	0	Corporate UL-CVAT				X	CVT16	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT17	Startup	1	BC	116000	0	Corporate UL-CVAT				X	CVT17	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT18	Startup	1	BC	117000	0	Corporate UL-CVAT				X	CVT18	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	CVT19	Startup	1	BC	118000	0	Corporate UL-CVAT				X	CVT19	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT01	Startup	1	BC	137000	0	Corporate UL-GPT				X	GPT01	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT02	Startup	1	BC	138000	0	Corporate UL-GPT				X	GPT02	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT03-S	Startup	1	BC	324000	0	Corporate UL-GPT				X	GPT03	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT04	Startup	1	BC	140000	0	Corporate UL-GPT				X	GPT04	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT05	Startup	1	BC	141000	0	Corporate UL-GPT				X	GPT05	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT06	Startup	1	BC	142000	0	Corporate UL-GPT				X	GPT06	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT07	Startup	1	BC	143000	0	Corporate UL-GPT				X	GPT07	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT08	Startup	1	BC	144000	0	Corporate UL-GPT				X	GPT08	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT09	Startup	1	BC	145000	0	Corporate UL-GPT				X	GPT09	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT10	Startup	1	BC	146000	0	Corporate UL-GPT				X	GPT10	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT11	Startup	1	BC	147000	0	Corporate UL-GPT				X	GPT11	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT12	Startup	1	BC	148000	0	Corporate UL-GPT				X	GPT12	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 2 Values	GPT13-S	Startup	1	BC	325000	0	Corporate UL-GPT				X	GPT13	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT14	Startup	1	BC	150000	0	Corporate UL-GPT				X	GPT14	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT15	Startup	1	BC	151000	0	Corporate UL-GPT				X	GPT15	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT16	Startup	1	BC	152000	0	Corporate UL-GPT				X	GPT16	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT17	Startup	1	BC	153000	0	Corporate UL-GPT				X	GPT17	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT18	Startup	1	BC	154000	0	Corporate UL-GPT				X	GPT18	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 2 Values	GPT19	Startup	1	BC	155000	0	Corporate UL-GPT				X	GPT19	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 3 Accounting	A_01	Startup	1	BC	186000	0			NONE		X	A_01	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 3 Accounting	A_02	Startup	1	BC	188000	0		stopped first run	NONE		X	A_02	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	
Section 3 Accounting	A_03	Startup	1	BC	189000	0			NONE		X	A_03	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 3 Accounting	A_03B	Startup	1	BC	334001	0			NONE		X	A_03B	5/17/2001 2:10:42 PM	28.3.4	1 - Corporate UL	
Section 3 Accounting	A_04	Startup	1	BC	190000	0		no output x 2	NONE		X	A_04	5/17/2001 2:10:42 PM	28.3.4	1 - TransMillenium	

Figure C.3: AMS Regression Test Suite.

Appendix D: Mathematical Preliminaries

The mathematical concept underlying the relational model is the set-theoretic *relation* that is a subset of the Cartesian product of a list of domains. This set-theoretic relation gives the model its name. Formally a *domain* is simply a set of values. For example the set of integers is a *domain*. Also the set of character strings of length 20 and the real numbers are examples of domains.

The *Cartesian product* of domains D_1, D_2, ... D_k, written $D_1 \times D_2 \times ... \times D_k$ is the set of all k-tuples v_1, v_2, ... v_k, such that $v_1 \in D_1$, $v_2 \in D_2$, ... $v_k \in D_k$.
For example, with k=2, D_1={0,1} and D_2={a, b, c} then $D_1 \times D_2$ is {(0,a), (0,b), (0,c), (1,a), (1,b), (1,c)}.

A Relation is any subset of the *Cartesian product* of one or more domains: $R \subseteq D_1 \times D_2 \times ... \times D_k$.

For example {(0,a), (0,b), (1,a)} is a relation; it is in fact a subset of $D_1 \times D_2$.

The members of a relation are called tuples. Each relation of some *Cartesian product* $D_1 \times D_2 \times ... \times D_k$ is said to have arity k and is therefore a set of k-tuples.
A relation can be viewed as a table where every tuple is represented by a row and every column corresponds to one component of a tuple. Giving names (called attributes) to the columns leads to the definition of a *relation scheme*.

A *relation scheme* R is a finite set of attributes A_1, A_2, ... A_k. There is a domain D_i, for each attribute A_i, $1 \leq i \leq k$, where the values of the attributes are taken from. We often write a relation scheme as R(A_1, A_2, ... A_k).

A *relation scheme* is just a kind of template whereas a *relation* is an instance of a *relation scheme*. The relation consists of tuples (and can therefore be viewed as a table); not so the relation scheme.

Operations: Relational Algebra consists of a set of operations on relations:

SELECT (Ω): extracts *tuples* from a relation that satisfy a given restriction. Let *R* be a table that contains an attribute *A*. $\Omega_{A=a}(R)$ = {t ∈ R &mid; t(A) = a} where t denotes a tuple of *R* and t(A) denotes the value of attribute *A* of tuple t.

PROJECT (∏): extracts specified *attributes* (columns) from a relation. Let R be a relation that contains an attribute X. $\prod_X(R) = \{t(X)$ &mid; $t \in R\}$, where t(X) denotes the value of attribute X of tuple t.

PRODUCT (×): builds the Cartesian product of two relations. Let R be a table with arity k_1 and let S be a table with arity k_2. $R \times S$ is the set of all $k_1 + k_2$-tuples whose first k_1 components form a tuple in R and whose last k_2 components form a tuple in S.

UNION (∪): builds the set-theoretic union of two tables. Given the tables R and S (both must have the same arity), the union $R \cup S$ is the set of tuples that are in R or S or both.

INTERSECT (∩): builds the set-theoretic intersection of two tables. Given the tables R and S, $R \cap S$ is the set of tuples that are in R and in S. We again require that R and S have the same arity.

DIFFERENCE (/ or &setmn;): builds the set difference of two tables. Let R and S again be two tables with the same arity. R / S is the set of tuples in R but not in S.

JOIN (∏): connects two tables by their common attributes. Let R be a table with the attributes A, B and C and let S be a table with the attributes C, D and E. There is one attribute common to both relations, the attribute C. $R \prod S = \prod_{R.A,R.B,R.C,S.D,S.E}(\prod_{R.C=S.C}(R \times S))$. What are we doing here? We first calculate the Cartesian product $R \times S$. Then we select those tuples whose values for the common attribute C are equal ($\prod_{R.C = S.C}$). Now we have a table that contains the attribute C two times and we correct this by projecting out the duplicate column.

Appendix E: List of contemporary coverage and profile tools

The information on coverage tools includes the results of a comparative feature analysis by Paterson Technology [57].

- C-Cover is a coverage tool made by Bullseye [52] Platforms: Win32, Unix; languages: C/C++. It is highly customizable and flexible. Among its features are support for multiple threads, processes, users; support for DLLs, shared libraries, device drivers, ActiveX, DirectX, COM, and time-critical applications; and full support for both C++ and C including templates, exception handling, inline functions, namespace.

- TrueCoverage is a coverage tool made by NuMega [56]. Platform: Win32; languages: C/C++, Java, VB. TrueCoverage analyzes and reports how much of an application's code was, or was not executed. This analysis and reporting can cover an individual testing session or a combination of “n” number of testing sessions. TrueCoverage reports this data down to the individual line of code and function levels.

- PureCoverage is a coverage tool produced by Rational [60]. Platforms: Win32, Unix; languages: C/C++, Java, VB. It automatically pinpoints areas of code that code that have and have not been exercised during testing. PureCoverage exposes untested code in the target application, including components with or without source code such as third party controls or system DLLs.

- TCAT is a coverage tool made by Software Research [61]. Platforms: Win32, Unix; languages: C/C++, Java. It features both static source code analysis and coverage analysis. It can be either GUI or command-line driven.

- LiveCoverage is a coverage tool from PatersonTechnology [58]. Platform: Win32; languages: C/C++, VB. The tool is capable of monitoring multi-threaded and multi-process scenarios, as well as out-of-process COM servers. Both interactive and automated modes are available.

- Visual FoxPro Coverage Profiler from Microsoft [50] contains both a coverage analyzer and a profiler application. The tool consists of a customizable coverage engine and a multiwindow analysis application. The coverage analyzer can be automated to run without user interaction.

- ActiveOptimizer pdProfiler from Hallogram Publishing [55] is a VB profiler with remote tracing and code coverage. It has minimal effect on application performance. It gives a complete execution trace of the application run.

- VB-Miner from CAST [53] is a source code analyzer for VB on Win32 platforms. Provides graphic representation of elements internal to the target module, external elements, and all interactions between these elements.

Appendix F: List of Common Refactoring Techniques

The following technique synopses are taken from Fowler [86]. We found these useful in our refactoring efforts.

Add Parameter

- Motivation: Method needs more information from caller.
- Technique: Add parameter for object that can pass on this information.

Change Bidirectional Association to Unidirectional

- Motivation: Two-way association where one class no longer needs features from other
- Technique: Drop unneeded end of association.

Consolidate Conditional Expression

- Motivation: Sequence of conditional tests with same result
- Technique: Combine into single conditional expression and extract.

Consolidate Duplicate Conditional Fragments

- Motivation: Same code fragment in all branches of conditional expression
- Technique: Move it outside of expression.

Convert Procedural Design to Objects

- Motivation: Code written in procedural style
- Technique: Turn data records into objects, break up behavior, and move the behavior to the objects.

Decompose Conditional

- Motivation: Complicated conditional statement

- Technique: Extract methods from the condition, the *then* part, and the *else* parts.

Duplicate Observed Data

- Motivation: Domain data available only to GUI, domain methods need access.
- Technique: Copy data to domain object. Create observer to synchronize the duplicated data.

Extract Interface

- Motivation: Multiple clients use same subset of class interface, or two classes have partial common interface.
- Technique: Extract the subset into an interface.

Extract Method

- Motivation: Code fragment can be grouped together.
- Technique: Turn fragment into method with self-explanatory name.

Extract Subclass

- Motivation: Class has features used only in some instances.
- Technique: Create subclass for that feature subset.

Introduce Explaining Variable

- Motivation: Complicated expression
- Technique: Put expression result, or expression parts, in temporary variable with self-explanatory name.

Parameterize Method

- Motivation: Several methods do similar things with different values inside the method body.
- Technique: Create one method with parameter for the different values.

Remove *Assignments* to Parameters

- Motivation: Code assigns to a parameter.
- Technique: Use temporary variable instead.

Remove Control Flag

- Motivation: Variable acts as control flag for series of Boolean expressions.
- Technique: Use break or return instead.

Remove Parameter

- Motivation: Parameter no longer used by method body
- Technique: Remove it.

Rename Method

- Motivation: Method name not indicative of purpose
- Technique: Rename it.

Replace Array with Object

- Motivation: Certain array elements mean different things.
- Technique: Replace array with object with field for each element.

Replace Parameter with Explicit Methods

- Motivation: Method runs different code depending on values of enumerated parameter.
- Technique: Create separate method for each parameter value.

Replace Parameter with Method

- Motivation: Object invokes method 1, passes result as method 2 parameter. Receiver can also invoke method 1.
- Technique: Remove parameter; let receiver invoke method

Split Temporary Variable

- Motivation: Temporary variable assigned more than once, but is not loop variable or collecting temporary variable.
- Technique: Make separate temporary variable for each assignment.

Appendix G: Evolution Manager Utility

Figure G.1 shows evolution manager utility list of features:

- Feature function relationship based upon test case and features, and test case and functions (Figure G.2)
- Feature function relationship in terms of coverage percentage (Figure G.3)
- Exclusive coverage of a feature within a function (Figure G.4)
- Threshold T(FI,K,C) (Figure G.5)
- Variable usage (set or use) by a feature within a function (Figure G.6)
- Feature implementation in terms of which lines of code and variables implement the feature (Figure G.7)
- Several tracking reports such as feature lists, function lists, features within a release etc

Main Functions with input parameters, return values and SQL statements used:

Purpose:	This function is used to retrieve and compile data for feature-function relationships.
Parameters:	None
Return Value:	A recordset with the following structure: • **Feature_ID:** Unique identifier for each feature • **Function_ID:** Unique identifier for each function • **Feature_Name** : The name of the feature • **Function_Name** : The name of the function • **Total_Lines** : Total number of lines of function • **Common_Lines** : Comma separated list of lines common to all test cases for the current feature - function pair • **Test_Cases** : Comma separated list of test cases for the current feature - function pair • **All_Lines** : Comma separated list of all lines in all test cases for the current feature - function pair • **Exclusive_Lines** : Comma separated list of exclusive lines in all test cases for the current feature - function pair • **Common_Lines_Count** : Number of lines in common lines list • **All_Lines_Count** : Number of lines in ALL lines list • **Exclusive_Lines_Count** : Number of lines in exclusive lines list • **Common_Coverage** : Common_Lines_Count / Total_Lines * 100 • **All_Coverage** : All_Lines_Count / Total_Lines * 100 • **Exclusive_Coverage** : Exclusive_Lines_Count / Total_Lines * 100
Expected Initial Status:	Any
Expected Final Status:	*rstRet* contains feature-function information for future information requests.
SQL Statements:	*SELECT DISTINCT Test_Cases_TO_Function.Used_Lines, Test_Cases_TO_Function.Test_Case_ID, Function_List.Function_ID, Feature.Feature_ID, Feature.Feature_Name, Function_List.Function_Name FROM Function_List RIGHT JOIN (Feature RIGHT JOIN (Test_Cases_TO_Function LEFT JOIN Test_Case_Feature_Map ON Test_Cases_TO_Function.Test_Case_ID = Test_Case_Feature_Map.Test_Case_ID) ON Feature.Feature_ID = Test_Case_Feature_Map.Feature_ID) ON Function_List.Function_ID = Test_Cases_TO_Function.Function_ID;* This statement is used to get the properties of the relation feature - function. It can be anywhere from NO-LINES to ALL-LINES relation. The selection takes all the test cases related with a feature (Test_Case_Feature_Map), and then selects all the functions related with each of the test cases (Test_Cases_TO_Function).
	SELECT Function_Name, Feature_Name, Feature_ID, Function_ID, Total_Lines FROM Function_List, Feature ORDER BY Feature_Name This statement is used to get the function-feature pairs and the information about them. The data is used later to make sure all GRID information is included. A JOIN is not used between the tables because we want all possible function-feature combinations.

Table G.0.1: Implementation details on Evolution Manager Utility.

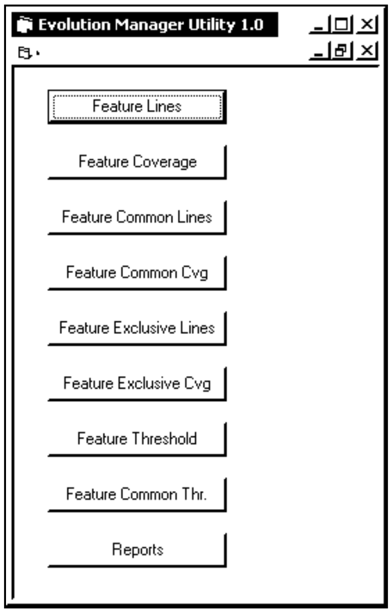
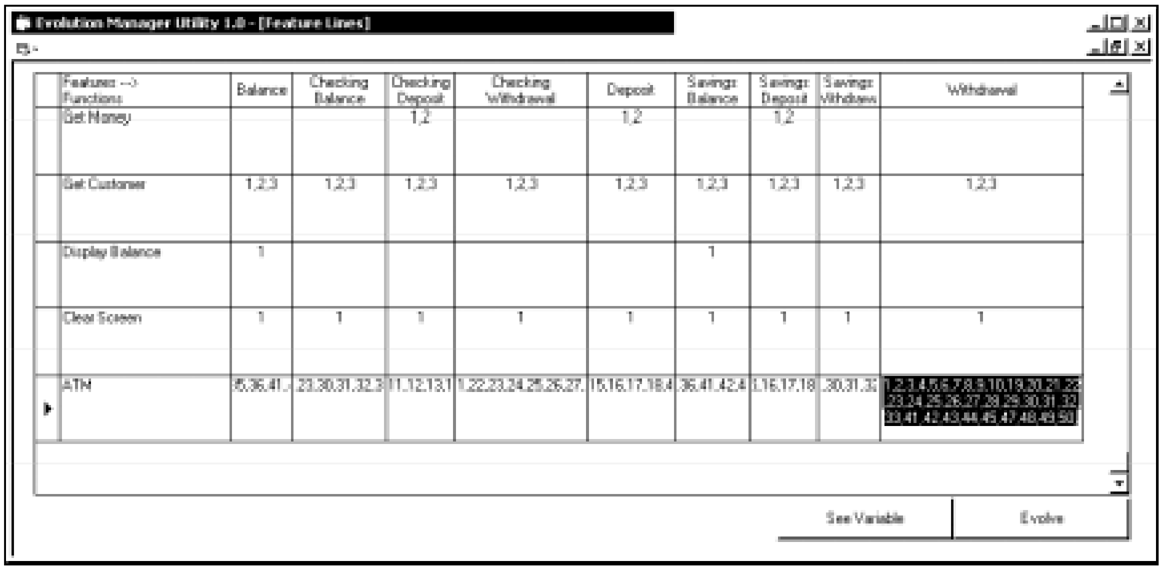
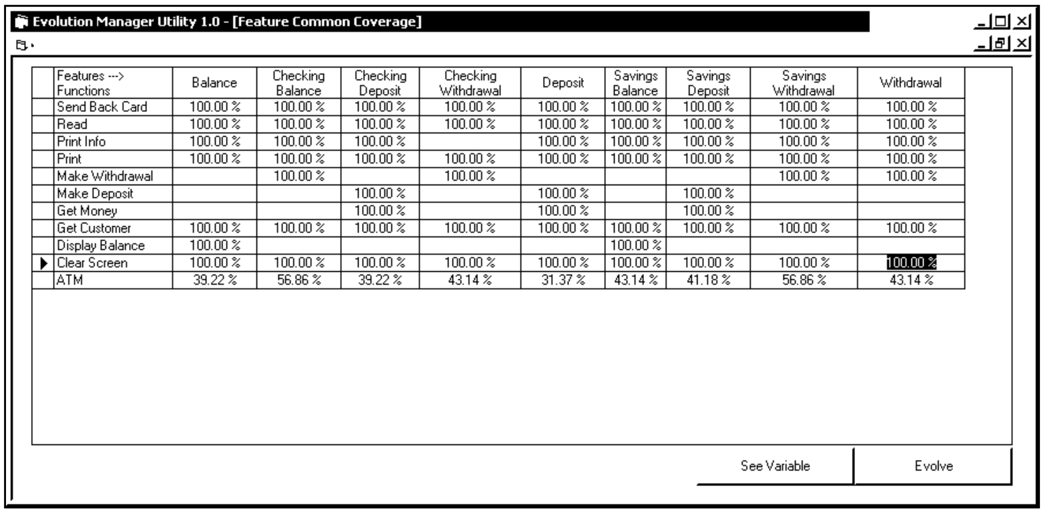


Figure G.1: Evolution manager list of features.



Features --> Functions	Balance	Checking Balance	Checking Deposit	Checking Withdrawal	Deposit	Savings Balance	Savings Deposit	Savings Withdraw	Withdrawal
Get Money			1,2		1,2		1,2		
Get Customer	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3
Display Balance	1					1			
Clear Screen	1	1	1	1	1	1	1	1	1
ATM	35,36,41,.	23,30,31,32,3	11,12,13,1	1,22,23,24,25,26,27,	15,16,17,18,4	36,41,42,4	3,16,17,18	,30,31,32	1,2,3,4,5,6,7,8,9,10,19,20,21,22 ,23,24,25,26,27,28,29,30,31,32 33,41,42,43,44,45,47,48,49,50

Figure G.2: Feature function relationships for ATM example.

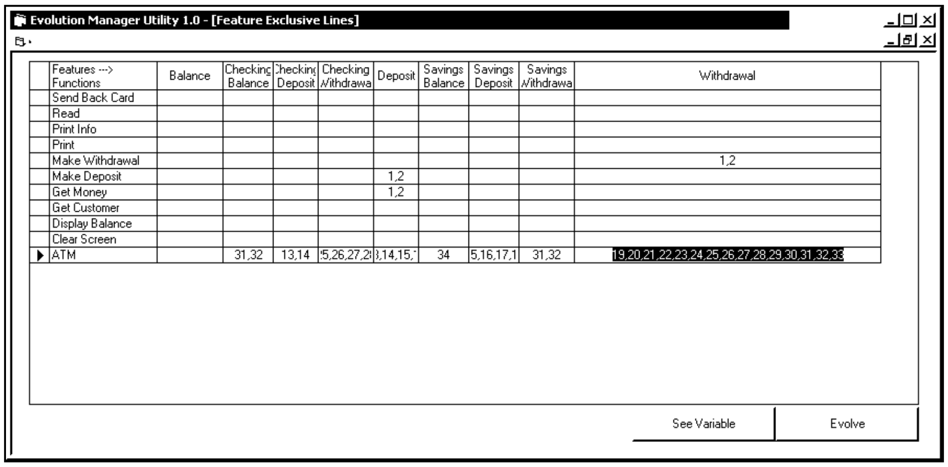


Evolution Manager Utility 1.0 - [Feature Common Coverage]

Features ---> Functions	Balance	Checking Balance	Checking Deposit	Checking Withdrawal	Deposit	Savings Balance	Savings Deposit	Savings Withdrawal	Withdrawal
Send Back Card	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %
Read	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %
Print Info	100.00 %	100.00 %	100.00 %		100.00 %	100.00 %	100.00 %	100.00 %	100.00 %
Print	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %
Make Withdrawal		100.00 %		100.00 %				100.00 %	100.00 %
Make Deposit			100.00 %		100.00 %		100.00 %		
Get Money			100.00 %		100.00 %		100.00 %		
Get Customer	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %
Display Balance	100.00 %					100.00 %			
Clear Screen	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %
ATM	39.22 %	56.86 %	39.22 %	43.14 %	31.37 %	43.14 %	41.18 %	56.86 %	43.14 %

See Variable | Evolve

Figure G.3: Feature function coverage (ATM example).

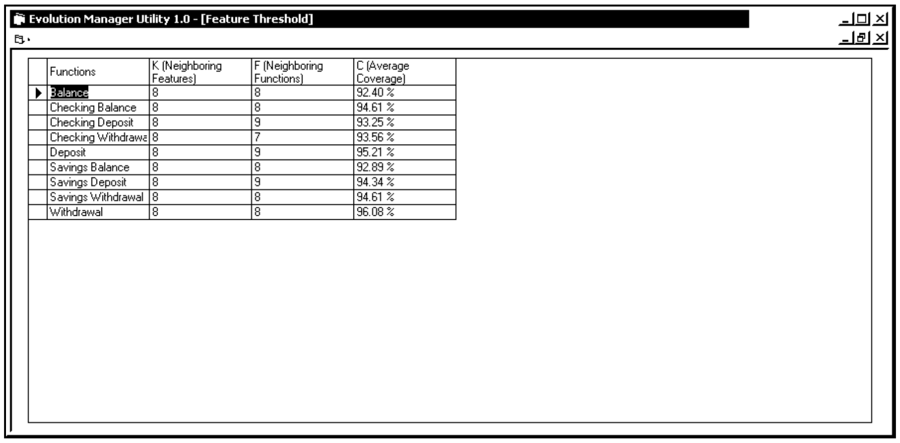


Evolution Manager Utility 1.0 - [Feature Exclusive Lines]

Features ---> Functions	Balance	Checking Balance	Checking Deposit	Checking Withdrawa	Deposit	Savings Balance	Savings Deposit	Savings Withdrawa	Withdrawal
Send Back Card									
Read									
Print Info									
Print									
Make Withdrawal									1,2
Make Deposit					1,2				
Get Money					1,2				
Get Customer									
Display Balance									
Clear Screen									
ATM		31,32	13,14	5,26,27,2	3,14,15,	34	5,16,17,1	31,32	19,20,21,22,23,24,25,26,27,28,29,30,31,32,33

See Variable | Evolve

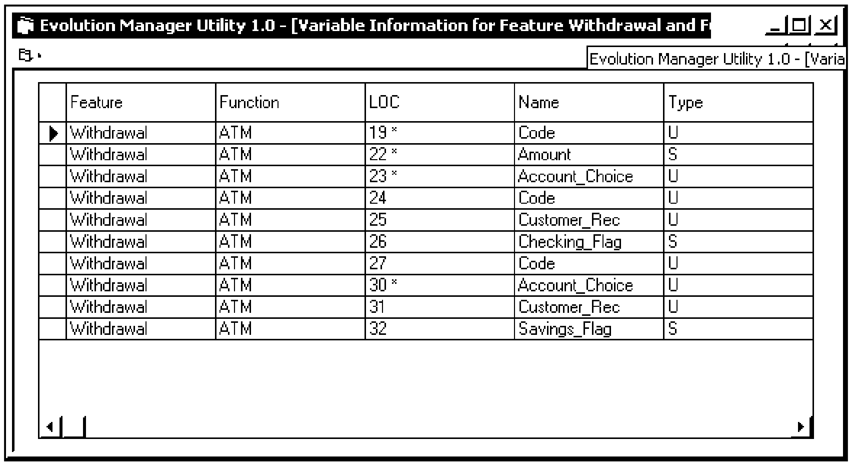
Figure G.4: Exclusive coverage by withdrawal feature in ATM function.



Evolution Manager Utility 1.0 - [Feature Threshold]

	Functions	K (Neighboring Features)	F (Neighboring Functions)	C (Average Coverage)
▶	Balance	8	8	92.40 %
	Checking Balance	8	8	94.61 %
	Checking Deposit	8	9	93.25 %
	Checking Withdrawa	8	7	93.56 %
	Deposit	8	9	95.21 %
	Savings Balance	8	8	92.89 %
	Savings Deposit	8	9	94.34 %
	Savings Withdrawal	8	8	94.61 %
	Withdrawal	8	8	96.08 %

Figure G.5: Threshold in ATM example.

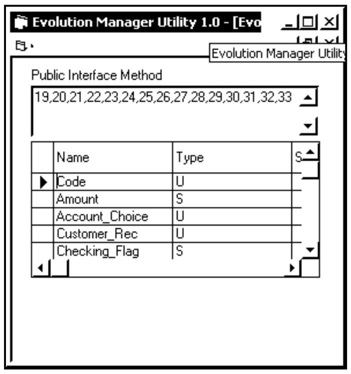


Evolution Manager Utility 1.0 - [Variable Information for Feature Withdrawal and F

Evolution Manager Utility 1.0 - [Varia

	Feature	Function	LOC	Name	Type
▶	Withdrawal	ATM	19 *	Code	U
	Withdrawal	ATM	22 *	Amount	S
	Withdrawal	ATM	23 *	Account_Choice	U
	Withdrawal	ATM	24	Code	U
	Withdrawal	ATM	25	Customer_Rec	U
	Withdrawal	ATM	26	Checking_Flag	S
	Withdrawal	ATM	27	Code	U
	Withdrawal	ATM	30 *	Account_Choice	U
	Withdrawal	ATM	31	Customer_Rec	U
	Withdrawal	ATM	32	Savings_Flag	S

Figure G.6: Variable usage for withdrawal feature in ATM function.



Evolution Manager Utility 1.0 - [Evo

Evolution Manager Utility

Public Interface Method

19,20,21,22,23,24,25,26,27,28,29,30,31,32,33

	Name	Type	S
▶	Code	U	
	Amount	S	
	Account_Choice	U	
	Customer_Rec	U	
	Checking_Flag	S	

Figure G.7: Withdrwal FI (feature lines of code and variables).

Bibliography

[1] χSuds User's Manual. Telecordia Technologies, 1998.

[2] A. Brown, "*Component Based Software Engineering*" IEEE Computer Society, 1996, pp. 175-186.

[3] A. Davis and R. Rauscher, "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications", *Proceedings, 1st Conference on Specifications of Reliable Software*, IEEE Computer Society, Cambridge, MA, April 1979, pp. 15-35.

[4] A. Davis, "The Design of a Family of Application-Oriented Requirements Languages", *IEEE Computer*, Vol. 15, No. 5, May 1982, pp. 21-28.

[5] A. Lakhotia and J. C. Deprez, "Restructuring Functions with Low Cohesion", *Proceedings, 6th Working Conference on Reverse Engineering*, London, England, May 1996, pp. 36-46.

[6] A. Malony, D. Hammerslag, and D. Jabalonski, "Traceview: A Trace Visualization Tool", *IEEE Software*, September 1991, pp. 19-28.

[7] A. Mehta and G. Heineman, "Architectural Evolution of Legacy System", *Proceedings, 23rd Annual International Computer Software and Applications Conference*, Phoenix, Arizona, August 1999, pp. 110-119.

[8] A. Mehta and G. Heineman, "COTS Integration and Extension Workshop", *Continuing Collaborations for Successful COTS Development*, International Conference on Software Engineering, Limerick, Ireland, May 2000, pp. 67-72.

[9] A. Mehta and G. Heineman, "Evolving Legacy System Features into Fine-Grained Components", *ICSE 2002*, Orlando, FL, May 2002, pp. 417-427.

[10] A. Onoma, W. Tsai, M. Poonawala, and H. Suganuma, "Regression Testing in an Industrial Environment", *ACM Communications*, Vol. 41, May 1998, pp. 81-86.

[11] A. Onoma, W. Tsai, M. Poonawala, and H. Suganuma, "Regression Testing in an Industrial Environment", *Communications of the ACM*, Vol. 41. No.5, May 1998, pp. 81-86.

[12] A. Sloane and J. Holdsworth, "Beyond Traditional Program Slicing", *Proceedings, 1996 International Symposium on Software Testing and Analysis (ISSTA '96)*, ACM SIGSOFT, San Diego, CA, January 1996, pp. 180-186.

[13] B Sanden, "Designing control systems with Entity-life Modeling", *Journal of Systems and Software*, Vol. 28, No. 4, pp. 225-237.

[14] B. Beizer, *Software Testing Techniques*, Van Norstrand Reinhold, New York, 1990.

[15] B. Boehm, *Software Engineering Economics*, Prentice Hall, 1981, pp. 182-194.

[16] B. Calloni, M. DelPrincipe and K. Littlejohn, INSERT: A COTS-based Solution for Building High-Assurance Applications, "*Proceedings, Gateway to the New Millennium; 18th Digital Avionics Systems Conference*", St Louis, MO, May 1999, pp. 101-112.

[17] B. Kitchenham and N. Taylor, "Software Project Development Cost Estimation", *The Journal of Systems and Software*, May 1985, pp. 267-278.

[18] B. Korel and J. Laski, "Dynamic Program Slicing", *Information Processing Letters*, Vol. 29, No. 3, 1998, pp. 155-163.

[19] C. Dorda, L. Grace, P. Place, D. Plakosh, and R. Seacord, "Technical Report - Incremental Modernization for Legacy Systems", CMU/SEI-2001-TN-006, July 2001.

[20] C. Jones, *Applied Software Measurement: Assuring Productivity and Quality*, McGraw-Hill, 2nd Edition, 1996.

[21] C. Kemerer and S. Slaughter, "An empirical approach to studying software evolution", *IEEE Transaction on Software Engineering*, Vol. 25, No. 6, pp. 493-509.

[22] C. Kop and H. Mayr, "Conceptual Pre-design: Bridging the Gap between Requirements and Conceptual Design", *Proceedings, 3rd International Conference on Requirements Engineering*, IEEE Computer Society, Tucson, AZ, April 1998, pp. 90-98.

[23] C. Lindig, “Concept-Based Component Retrieval”, *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, Montreal, Canada, January 1995, pp. 99-110.

[24] C. Prehofer, "*Feature-Oriented Programming: A Fresh Look at Objects*", European Conference on Object-Oriented Programming (ECOOP), 1997, pp. 419-443.

[25] C. Turner, “*Feature Engineering of Software Systems*”, Ph.D. Thesis, May 1999.

[26] C. Turner, A. Fuggetta, L. Lavazza and, A. Wolf, “A Conceptual Basis for Feature Engineering”, *Journal of Systems and Software*, Volume 49, Issue 1, December 1999, pp. 3-15.

[27] D. Carney, “Assembling Large Systems from COTS Components: Opportunities, Cautions and Complexities”, *SEI Monograph Series*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, June 1997, pp. 71-82.

[28] D. Coleman, B. Lowther, P. Oman, and D. Ash, “Using metrics to evaluate software system maintainability”, *IEEE Computer*, Vol. 27. No. 8, pp. 44-49.

[29] D. Garlan and M. Shaw, “An Introduction to Software Architecture”, *Advances in Software Engineering and Knowledge Engineering*, Vol. I, World Scientific Publishing, January 1993, pp. 1-39.

[30] D. Kafura, “The use of software complexity metrics in software maintenance”, *IEEE Transaction on Software Engineering*, Vol. 12, No. 4, pp. 335-343.

[31] D. Smith, H. Muller, and S. Tilley, “The Year 2000 Problem: Issues and Implications”, *Technical Report CMU/SEI-97-TR-002*, Software Engineering Institute, Pittsburgh, Pennsylvania, 1997.

[32] *DevPartner Studio User Manual*, Compuware Corporation, 2000.

[33] E. Buss, “Investigating Reverse Engineering Technologies for the CAS Program Understanding Project”, *IBM Systems Journal*, Vol. 33, No. 3, 1994, pp. 477-500.

[34] E. Codd, *Relational Completeness of Data Base Sublanguages*, Prentice Hall, 1972, pp. 65-98.

[35] F. Tip, "A Survey of Program Slicing Techniques", *Technical Report CS-R9428*, Centrum voor Wiskunde Informatica, Amsterdam, The Netherlands, 1994, pp. 35-42.

[36] G. Booch, J. Rumbaugh, and I. Jacobson, "*The Unified Modeling Language User Guide*", 2nd Edition, Addison-Wesley, 1998, pp. 160-165.

[37] G. Bruno and R. Agarwal, "Modeling the Enterprise Engineering Environment", *IEEE Transactions on Engineering Management*, Vol. 44, No. 1, February 1997, pp. 2-30.

[38] G. Heineman and W. Councill, *Component-Based Software Engineering: Putting The Pieces Together*, Addison-Wesley, 2001.

[39] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M Loingtier, and J. Irwin, "Aspect-Oriented Programming", *Proceedings, 11th European Conference on Object-Oriented Programming*, Berlin, Germany, June 1997, pp. 220-242.

[40] G. Rothermel and M. Harrold, "A Comparison of Regression Test Selection Techniques", *Technical Report, Department of Computer Science*, Clemson University, October 1994, pp. 65-72.

[41] G. Rothermel and M. Harrold, "A Safe, Efficient Algorithm for Regression Test Selection", *Proceedings, IEEE Software Maintenance Conference*, Montreal, Canada, September 1993, pp. 358–367.

[42] G. Rothermel and M. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Transactions Software Engineering*, Vol. 22, No. 8, August 1996, pp. 529-551.

[43] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Test Case Prioritization: An Empirical Study", *Proceedings, International Conference on Software Maintenance*, Oxford, UK, August 1999, pp. 179–188.

[44] G. Succi and F. Baruchelli, "The Cost of Standardizing Components for Software Reuse", *Standard View*, Vol. 5, No. 2, pp. 61-75.

[45] G. Valetto and G. Kaiser, "Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments", *Proceedings, 7th IEEE International Workshop on CASE*, Toronto, Canada, July 1995, pp. 40-48.

[46] H. Agrawal, J. Horgan, E. Krauser, and S.A. London, “Incremental Regression Testing”, *Proceedings, IEEE Software Maintenance Conference,* Montreal, Quebec, September 1993, pp. 348–357.

[47] H. Kaindl, S. Kramer, and R. Kacsich, “A Case Study of Decomposing Functional Requirements Using Scenarios” *Proceedings, 3rd International Conference on Requirements Engineering*, IEEE Computer Society, Vienna, Austria, April 1998, pp. 82-89.

[48] H. Leung and L. White, “Insights into Regression Testing”, *Proceedings, IEEE Software Maintenance Conference*, Los Alamitos, CA, October 1989, pp. 60–69.

[49] H. Sneed, “Architecture and Functions of a Commercial Software Reengineering Workbench”, *Proceedings, 2nd Euromicro Conference on Maintenance and Reengineering,* Florence, Italy, March 1998, pp. 2-10.

[50] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fox7help/html/newcoverage_profiler_application.asp

[51] http://www.anubex.com/solutions!main.asp

[52] http://www.bullseye.com/ccoverFeature.html

[53] http://www.castsoftware.com/products/Miners/VB-Minerdetails.html

[54] http://www.cise.nsf.gov/new/evnt/wksp/presentations/software/sld005.htm

[55] http://www.hallogram.com/pdprofiler/index.html

[56] http://www.numega.com, Numega Corporation.

[57] http://www.patersontech.com/TestCoverage/Compare.htm

[58] http://www.patersontech.com/TestCoverage/LiveCoverage

[59] http://www.rational.com/

[60] http://www.rational.com/products/purecoverage_nt/prodinfo.jsp

[61] http://www.soft.com/TestWorks/

[62] http://www.sun.com

[63] IEEE Standard Glossary of Software Engineering Terminology, *IEEE Standards Collection, Software Engineering,* IEEE, New York, NY, 1994, pp. 54-55.

[64] J. Bergey, L. Northrop, and D. Smith, "Enterprise Framework for the Disciplined Evolution of Legacy Systems", *CMU/SEI-97-TR-007, ADA 330880*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, July 1997, pp. 12-14.

[65] J. Bisbal, "Legacy Information Systems: Issues and Directions", *IEEE Software*, September 1999, pp. 103-111.

[66] J. Borstler, "Feature-oriented Classification for Software Reuse", *Proceedings, 7th International Conference on Software Engineering and Knowledge Engineering*, Rockville, MD, September 1995, pp. 204-211.

[67] J. Borstler, "FOCS: A Classification System for Software Reuse", *Proceedings, 11th Pacific Northwest Software Quality Conference*, Portland, OR, October 1993, pp. 201-211.

[68] J. Bosch, "Organizing for Software Product Lines", *Proceedings, 3rd International Workshop on Software Architectures for Product Families,* Las Palmas de Gran Canaria, Spain, March 2000, pp. 60-71.

[69] J. Connell, *Coding Techniques for Microsoft® Visual Basic® .NET*, Microsoft Press, December 2001, pp. 77.

[70] J. DeBaud and K. Schmid, "A Systematic Approach to Derive the Scope of Software Product Lines", *Proceedings, 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999, pp. 34-43.

[71] J. Deprez and A. Lakhotia, "A Formalism to Automate Mapping from Program Features to Code", *Proceedings, 8th International Workshop on Program Comprehension,* International Conference on Software Engineering 2000, Limerick, Ireland, May 2000, pp. 72-83.

[72] J. Field, G. Ramalingam, and F. Tip, “Parametric Program Slicing”, *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, CA, January 1995, pp. 379 – 392.

[73] J. Karlsson and K. Ryan, “A Cost-Value Approach for Prioritizing Requirements”, *IEEE Software*, Vol. 14, No. 5, September 1997, pp. 67-74.

[74] J. Kuusela and J. Savolainen, “Requirements Engineering for Product Families”, *Proceedings, 22nd of the International Conference on Software Engineering*, Limerick, Ireland, June, 2000, pp. 61-69.

[75] J. Penix and P. Alexander, “Using Formal Specifications for Component Retrieval and Reuse”, *Proceedings, 31st Hawaii International Conference on System Sciences,* Hawaii, November 1995, pp.356-65.

[76] J. Ransom, I. Sommerville and I. Warren, “A Method for Assessing Legacy Systems for Evolution”, *Proceedings, 2nd Euromicro Conference on Software Maintenance and Reengineering,* Palazzo degli Affari, Italy, March 1998, pp. 46-53.

[77] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, “Feature-Oriented Domain Analysis (FODA) Feasibility Study”, *Technical Report CMU/SEI 90 TR 21*, Software Engineering Institute, Pittsburgh, Pennsylvania, 1990.

[78] K. Lukoit, N. Wilde, S. Stowell, and T. Hennessey, “TraceGraph: Immediate Visual Location of Software Features”, *Proceedings, International Conference on Software Maintenance*, San Jose, CA, May 2000, pp. 33 –39.

[79] L. Belady and M. Lehman, “A Model of Large Program Development”, *IBM Systems Journal*, Vol. 15, No. 3, pp. 225-252.

[80] L. Brownsword and P. Clements, “A Case Study in Successful Product Line Development”, *CMU/SEI-96-TR-016, ADA 315802,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.

[81] L. O'Brien and D. Smith, “MAP and OAR Methods: Techniques for Developing Core Assets for Software Product Lines from Existing Assets”, *CMU/SEI-2002-TN-007,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, April 2002.

[82] L. Raccoon, "The Complexity Gap", *SIGSOFT Software Engineering Notes*, Vol. 20, No. 3, July 1995, pp. 37-44.

[83] M. Brodie and M. Stonebraker, *"Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach"*, Morgan Kaufmann Publishers, 1995, pp. 171-185.

[84] M. Cusumano and R. Selby, *Microsoft Secrets*, The Free Press, New York, 1995, pp. 37-45.

[85] M. Fowler and K. Scott, "*UML Distilled - Applying the Standard Object Modeling Language*", Object Technology Series, Addison-Wesley, 1997, pp. 150-172.

[86] M. Fowler, *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999.

[87] M. Griss, "Implementing Product-Line Features with Component Reuse", *Proceedings, 6th International Conference on Software Reuse*, Springer-Verlag, Vienna, Austria, June 2000, pp. 134-152.

[88] M. Harrold, R. Gupta, and M. Soffa, "A Methodology for Controlling the Size of a Test Suite," *ACM Transactions Software Engineering and Methodology*, Vol. 2, No. 3, July 1993, pp. 270–285.

[89] M. Jain, M. Murty and P. J. Flynn, "Data Clustering: A Review", *ACM Computing Surveys*, Vol. 31, No. 3, September 1999, pp. 264-323.

[90] M. L. Petrie, K. R. Nair, and G. K. Raghavan, "A Domain Analysis of Web Browser Architectures, Languages and Features", *Southcon 1996 Conference Record*, 1996, pp. 168-174.

[91] M. Svahnberg and J. Bosch, "Issues Concerning Variability in Software Product Lines", *Proceedings, 3rd International Workshop on Software Architectures for Product Families,* Las Palmas de Gran Canaria, Spain, March 2000, pp. 50-60.

[92] M. Weiser, "Program Slicing", *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, July 1984, pp. 352-357.

[93] M. Weiser, "Program Slicing", *Proceedings, 5th International Conference on Software Engineering,* IEEE Computer Society, New York, NY, March 1981, pp. 439-449.

[94] N. Medvidovic and R. Taylor, “Separating Fact from Fiction in Software Architecture”, *Proceedings, 3rd International Workshop on Software Architecture*, Orlando, FL, November 1998, pp. 105-108.

[95] N. Medvidovic, P. Oreizy, and R. Taylor, “Reuse of Off-the-shelf Components in C2-style Architectures”, *Proceedings, 1997 International Conference on Software Engineering*, Boston, MA, June 1997, pp. 692-700.

[96] N. Weiderman, J. Bergey, D. Smith, and S. Tilley, “Approaches to Legacy System Evolution”, *CMU/SEI-97-TR-014*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, August 1997.

[97] N. Weiderman, J. Bergey, D. Smith, B. Dennis, and S. Tilley, “Approaches to Legacy System Evolution”, *Technical Report CMU/SEI-97-TR-014*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1997.

[98] N. Weiderman, L. Northrop, D. Smith, S. Tilley, and K. Wallnau, “Implications of Distributed Object Technology for Reengineering”, *CMU/SEI-97-TR-005 ADA326945*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 1997.

[99] N. Wilde and M. Scully, “Software Reconnaissance: Mapping Program Features to Code”, *Journal of Software Maintenance: Research & Practice*, Vol. 7, No. 5, May 1995, pp. 49-62.

[100] P. Fingar and J. Stikeleather, “Distributed Objects for Business: Getting Started With the Next Generation of Computing”, *Sunworld Online*, Vol. 10, No. 4, March 1999, pp. 6-17.

[101] P. Hsia and A. Gupta, “Incremental Delivery Using Abstract Data Types and Requirements Clustering”, *Proceedings, 2nd International Conference on Systems Integration*, Los Alamitos, CA, June 1992, pp. 137-150.

[102] P. Oreizy, N. Medvidovic, and R. Taylor, “Architecture-based Runtime Software Evolution”, *Proceedings, 20th International Conference on Software Engineering*, Kyoto, Japan, April 1998, pp. 62-70.

[103] P. Tarr, H. Ossher, W. Harrison, and S. Sutton, “N Degrees of Separation: Multi-Dimensional Separation of Concerns”, *Proceedings, International Conference on Software Engineering*, May 1999, pp. 107-119.

[104] R. Agarwal and N. Mishra, "Renaissance Project – Methods and Tools for the Evolution and Reengineering of Legacy Systems", *Esprit Project*, Lancaster University, Lancaster, UK, June 1997, pp. 14-20.

[105] R. Bracho, "*Integrating the Corporate Computing Environment with ActiveWeb*", Active Software, Inc., Santa Clara, CA, 1997. http://www.activesw.com.

[106] R. Kazman and S. Carriere, "Playing Detective: Reconstructing Software Architecture from Available Evidence", *Journal of Automated Software Engineering,* Vol. 6, No. 2, April 1999, pp. 107-138.

[107] R. Knuth and O. Patashnik, *"Concrete Mathematics"*, Addison-Wesley, 1989, pp. 65-72.

[108] R. Krikhaar, *Software Architecture Reconstruction,* Ph.D. Thesis. University of Amsterdam, Amsterdam, The Netherlands, 1999.

[109] R. Marwane and A. Mili, "Building Tailor-Made Software Cost Model: Intermediate TUCOMO*", Information and Software Technology,* March 1991, pp. 232-238.

[110] R. W. Krut, "Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology", *Technical Report CMU/SEI 93 TR 01*, Software Engineering Institute, Pittsburgh, Pennsylvania, July 1993, pp. 12-20.

[111] S. Dorda, K. Wallnau, R. Seacord, and J. Robert, "A Survey of Legacy System Modernization Approaches", *Technical Note CMU/SEI-00-TN-003*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, April 2000, pp. 37-46.

[112] S. Hissam, "Experience Report: Correcting System Failure in a COTS Information System", *Proceedings, International Conference on Software Maintenance,* IEEE Computer Society Press, Los Alamitos, CA, 1998, pp. 68-79.

[113] S. Letovsky and E. Soloway, "Delocalized Plans and Program Comprehension", *IEEE Software*, Vol. 19, No. 3, pp. 41-48.

[114] S. Tilley and D. Smith, "Legacy System Reengineering", *Presented at the International Conference on Software Maintenance,* Software Engineering

Institute, Carnegie Mellon University, Pittsburgh, PA, November 1996, pp. 70-81.

[115] S. Tilley and D. Smith, “Perspectives on Legacy System Reengineering”, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 1996, pp. 6-13.

[116] S. Tsang and E. Magill, “Learning to Detect and Avoid Run-Time Feature Interactions in Intelligent Networks”, *IEEE Transactions on Software Engineering,* Vol. 24, No. 10, October 1998, pp. 818-830.

[117] S. Woods, S. Carriere, and R. Kazman, “A Semantic Foundation for Architectural Reengineering”, *Proceedings, International Conference on Software Maintenance (ICSM) 1999*, Oxford, UK, September 1999, pp. 391-398.

[118] T. Ball and S. Eick, “Software Visualization in the Large. *IEEE Computer,* Vol. 29, No. 4, April 1996, pp. 33-43.

[119] T. Ball, “Software Visualization in the Large”, *Institute of Electrical and Electronics Engineers* (*IEEE) Computer*, Vol. 29, No. 4, April 1996, pp. 33-43.

[120] T. Chen and M. Lau, “Dividing Strategies for the Optimization of a Test Suite,” *Information Processing Letters,* Vol. 60, No. 3, March 1996, pp. 135–141.

[121] T. Reps, T. Ball, T. M. Das, and J. Larus, “The Use of Program Profiling for Software Maintenance with Application to the Year 2000 Problem”. *Proceedings, European Software Engineering Conference (ESEC)/Foundation of Software Engineering (FSE) 1997: 6th ESEC and 5th American Computing Machinery (ACM) SIGSOFT Symposium on the FSE,* Zurich, Switzerland, September 1997, pp. 432-449.

[122] V. Basili and K. Freburger, “Programming measurement and estimation in the Software Engineering Laboratory”, *The Journal of Systems and Software,* Vol. 2 No. 3, pp. 47-57.

[123] V. Brand, M. Sellink, and C. Verhoef, “Generation of Components for Software Renovation Factories from Context-Free Grammars”, *Proceedings, 4th Working Conference on Reverse Engineerin*g, Amsterdam, The Netherlands, October 1997, pp. 144-153.

[124] W. Wong, J. Horgan, S. London, and A. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Software–Practice and Experience*, Vol. 28, No. 4, April 1998, pp. 347–369.

[125] W. Wong, S Gokahle, J. Horgan, and K Trivedi, "Locating Program Features Using Execution Slices", *Proceedings, 2nd IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, Richardson, TX, March 1999, pp 68-79.

[126] Y. Chen and B. Cheng, "Formalizing and Automating Component Reuse", *Proceedings, 9th IEEE International Conference on Tools with Artificial Intelligence*, Rockville, MD, August 1998, pp. 94-101.

[127] Y. Chen, D. Rosenblum, and K. Vo, "TestTube: A System for Selective Regression Testing", *Proceedings, 16th International Conference on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, May 1994, pp. 211-220.

[128] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn, "Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code", *Joint 8th ESEC and 9th ACM SIGSOFT Symposium on the FSE*, Vienna, Austria, May 2001, pp. 88-98.

[129] Y. Kamigaki, T. Nara, S. Machida, A. Hakata, and K. Yamaguchi, "160 Gbits ATM Switching System for Public Network", *Global Telecommunications Conference*, November 1996, pp. 1380-1387.

[130] Y. Miyazaki and K. Mori, "Constructive Cost Model (COCOMO) Evaluation and Tailoring", *Proceedings, 8th Conference on Software Engineering*, London, England, August 1985, pp. 292-299.