Evolving Legacy System Features using Regression Test Cases and Components

WPI-CS-TR-01-14

Alok Mehta
Chief Technology Officer
American Financial Systems, Inc.
9 Riverside Office Park
Weston, MA 02493
amehta@afs-link.com

George T. Heineman
Associate Professor
Computer Science
Worcester Polytechnic Institute
Worcester, MA
heineman@cs.wpi.edu

ABSTRACT
There is a constant need for practical, efficient and cost-effective software evolution techniques. We propose a novel evolution methodology that integrates the concepts of features and component-based software engineering (CBSE). We collect information about a legacy system's features through interviews with key developers, users of the system and analyzing the existing regression test cases. We found that regression test cases are untapped resources, as far as information about system features is concerned. By exercising each feature with their associated test cases using code profilers and similar tools, we are able to locate code that we can refactor to create components. These components are then inserted back into the legacy system, ensuring a working system structure. Our methodology is divided into two parts. Part one deals with identification of source code associated with features which need evolution and part two deals with creating components. In this paper, we present preliminary results of the our methodology.

KEYWORDS
Software Evolution, Legacy Systems, Program Slicing Feature Engineering, Component Based Software Engineering (CBSE), Testing, Refactoring, Source Code Renovation.

1. INTRODUCTION

Increasingly, organizations are viewing their software assets as an investment that grows in value rather than a liability whose value depreciates over time [1]. At the same time, organizations are under tremendous pressure to evolve their existing systems to better respond to marketplace needs and rapidly changing technologies. This constant pressure to evolve is driven by escalating customer expectations and the need to respond to new enterprise standards, incorporate new products and system features, improve performance, cope with endless new software releases, and hardware and software obsolescence.

To effectively evolve legacy systems in this fast-paced environment, managers require answers to the following types of question [2]: How do we plan the evolution of a large and complex system, including the reengineering of the system? What are the critical success factors of system evolution? How do we evolve the system without adversely affecting operations?

1.1. EVOLUTION MODEL

The repeated modification of legacy system has a cumulative effect that increases system complexity. Eventually, existing information systems become too fragile to modify and too important to discard; organizations mu st consider modernizing these legacy systems to remain viable.

Legacy systems are now written in modern programming languages; reengineering offers an approach to transforming a legacy system into one that can evolve in a disciplined manner. To be successful, reengineering requires insights from different perspectives including the software, managerial, and economic perspective [6]. Many software maintenance initiatives do not sufficiently incorporate the user's point of reference [7].

Researchers [3,4,5,8] have identified the two domains around which the entire field of software engineering revolves: the *problem domain* and the *solution domain*. Users interact with the system by inputting their requirements in input files (or database) that the system uses. These users are directly concerned with systems functionality; their perspective is always in the problem domain. These input files are often part of regression test cases that are used to check the stability between one version to another. Developers are concerned with the creation and maintenance of software development life cycle artifacts such as components; their perspective is rooted in the solution domain. A major source of difficulty in developing, delivering, and evolving successful software is the *complexity gap* that exists between the two perspectives (as termed by Raccoon [4]). The risk to viewing evolution just within a single domain is missing the connection between the two domains.

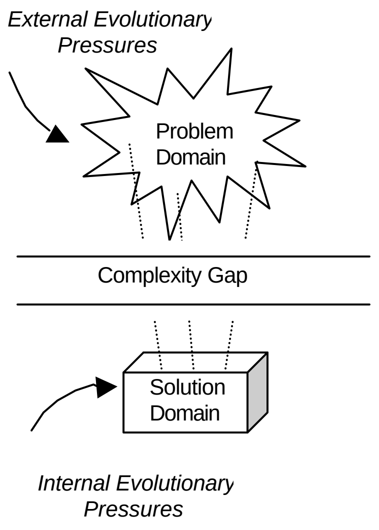

External Evolutionary Pressures
Problem Domain
Complexity Gap
Solution Domain
Internal Evolutionary Pressures

Figure 1: A unified evolution strategy is demanded.

Evolution focused solely on the problem domain may suggest changes that degrade the structure of the original code; similarly, evolution based solely on technical merits could propose changes unacceptable to end-users. The external evolutionary pressures shown in Figure 1 drive the implementation of new enhancements and functionality within the legacy system. During such implementations the developers are focused in implementing the business logic that is *directly* visible to the end users such as a menu item that spell checks the document in a word processing application. Often times the protocol and standards are not followed by the developers in order to meet the project dead lines, which can result in bad code such as adding a global variable when not needed. The internal evolutionary pressures force the developers to either restructure or refactor the code so the future enhancement or maintenance becomes manageable and cost-effective. During such evolutionary activities, the code is refactored, and protocols and standards are reestablished. The end-user may or may not see the changes made to the system but the goal of such refactoring is to reduce the future maintenance costs. Our research provides methodology for handling internal evolutionary pressures.

Years ago, researchers identified features as a natural organization of the problem domain [8,9]. Surprisingly, few approaches in the research literature concentrate on feature-based organization of a system's functionality. On the contrary, the solution domain is full of research that develops solutions revolving around software artifact management activities like design, component construction and testing. However, features are discussed in the problem domain and not mentioned in the solution domain.

A successful software evolution methodology must be self-sustaining; that is, over time, it should ensure that evolution is possible. Towards this end, we have identified an approach that integrates reengineering, features, and components. The basic outline of our methodology is as following:

- Test cases are selected by considering features.
- Slicing is guided by exercising system on the selected test cases.
- Slicing results drives refactoring, to create components.
- Results are measured using maintenance cost.

Our methodology has three basic assumptions. First, we assume that the legacy system to be evolved is written in one of the modern programming languages such as Visual Basic, C++, Java, COBOL or FORTRAN. Our methodology depends on a code- profiling tool for tracing the source that implements a particular feature. Second, we assume that the legacy system have regression test suites. Third, we assume that some domain knowledge and expertise is available, although this is not a binding constraint.

In Section 2 of this paper, we present our feature model that provides the theoretical basis for the evolution. We present a novel way to use the code profiling tools in the context of evolution in Section 3, while sharing some results. Section 4 explores related work and describes the expected benefits of our methodology.

2. FEATURE MODEL

Users often think of systems in terms of the features provided by the system. They exercise the system features by some sort of user input (stored in files or databases) which often times is also used by system maintainers as a part of regression testing. Intuitively, a feature is an identifiable bundle of system functionality that helps characterize the system from the user's perspective. Examples of features include ability of a word processor to spell check or ability of an accounting system to generate balance sheet statement for a given fiscal year. Software developers are expected to translate such feature-oriented requests and reports into a system design. Feature Engineering is the area 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 [3]. We define the term *feature* by partly borrowing from Turner's definition [3]. We developed our definition by integrating and extending the definitions from [3,4]:

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.*

This definition considers the root of feature(s) in the problem domain. It gives hints regarding the way a feature is implemented, traced [10] and how it can be used for software evolution because we consider the point of view relative to software development cycle. For example, one of the features of a system that performs complex calculation could be an ability to run in a batch mode without user's interaction. To an end-user this feature is a tremendous time saver as they do not interact with the system but instead they input their requests in a file or a database. At the same time this feature is used as a regression-testing tool by testers to maintain stability between two versions of the system, and to a developer it means designing a solution that requires no user interaction. No matter what the perspective relative to the life cycle is, we show that features can be located inside a program using regression test cases and code-profiling tools. Evolution follows after the location of the code.

2.1 FEATURES AND FUNCTIONALITY

Features and functionality are often used interchangeably which is a regrettable mistake. While a *function* is inherently an encapsulated entity in programming languages, a unit of functionality may not be so easily contained. For examp le, when a *user spell checks a text document*, many functions might execute.

Users comprehend a system through its features and 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; again, we see a clear gap between the problem and solution domain. To understand this gap between features and functionality consider figure 2.

When a single Feature implementation is contained within many software functions then the point of reference is the solution domain. Such code is often highly coupled and embedded within the legacy system. When many related features are implemented by a single function then the point of reference is the problem domain. It is trivial when a feature is implemented by a single function and the domain distinction is not important. Features and Regression Tests Researchers from a theoretical point of view [26-29] have extensively studied regression testing. Over its lifetime, a legacy system accumulates test cases that exist to ensure its integrity as it evolves. Often companies develop proprietary regression testing tools to automate these tests or to reduce the total number of tests to execute. However, there has been little discussion on specifically applying regression testing for evolutionary reasons. We propose a novel use of dynamic slicing [11] during regression testing to identify the code artifacts that interact with a particular feature and to incrementally refactor the code base to enable its future evolution.

Feature	***Functionality***	***Domain***
1	Many	Solution
Many	1	Problem
1	1	Trivial Case
Many	Many	Solution and Problem

Figure 2: Relationships between features and functionality

Testers, engineers and users work together to develop test cases to exercise the system. The selection of test cases is often a manual, analytical, iterative and time-consuming process. The goal in this step is to obtain right test cases instead of minimizing the number of test cases. Many times the testers ensure that the test cases are valid with respect to the changes programmed into the system. Over an extended period of time, these test cases reflect the system functionality in an implicit way because these test cases are viewed as a tool to test the stability of the system rather that a database of user input that reflects system functionality.

2.1. FEATURES/FUNCTION INTERACTION

To complete our description of our feature model, we identify feature/function interaction as depicted logically in Figure 3. This analysis is important when two or more features share common data or functions, and if developers are trying to identify the functionality implemented by these features. There are 5 cases where shared functionality between two or more features either affect the data and/or functionality in other features:

SSF - Shared Stateless Function: A stateless function [13] can be shared between two features. To refactor this code, simply place the common function into a component to be invoked from both features' code.

SSFF - Shared State-Full Function: A state-full function [13] is shared between two features in question. Refactoring may be complex, involving global variables and require control structures to make a full analysis [14].

DF – Dependent Function: A feature is dependent on a function that is part of another feature.

DD – Dependent Data: A feature is dependent on the data that is part of another feature.

SFD – Strong Function Dependency: A common function is associated with more than one feature and there is strong dependency on that common function.

As each feature is executed, *the code profiling tools* identify the code slices associated with each feature. Once the code is identified we can refactor that code to enable evolution of key parts of the system.

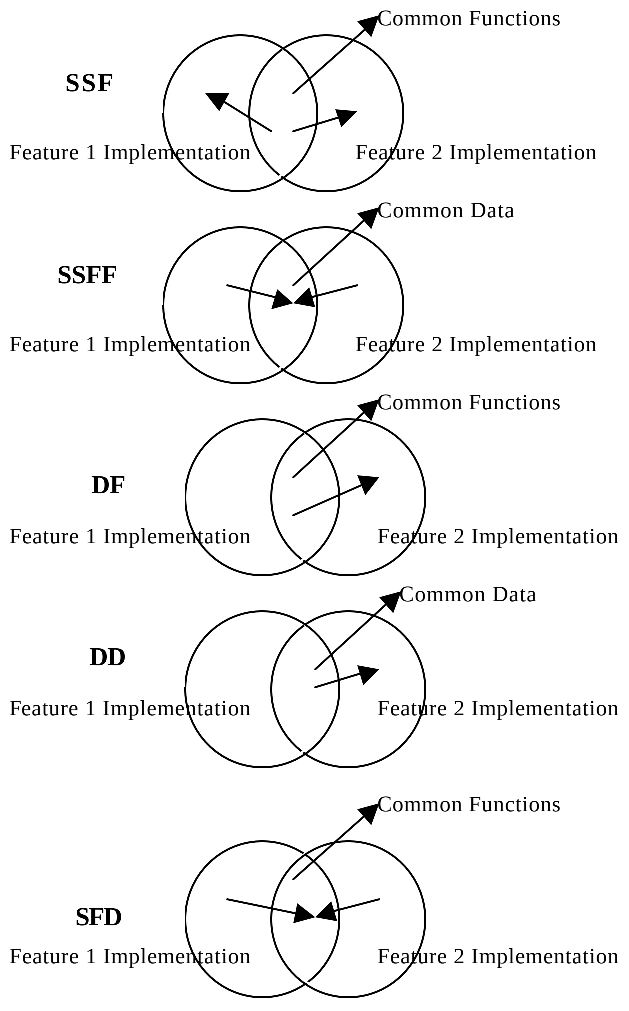


Figure 3: Feature/Function Interaction

3. METHODOLOGY

There are many reasons for evolving a legacy system [1,6]. When evolving the system, the planned work must be prioritized first, and then mapped to their associated features within the system. The system features are then identified and associated with the test cases, and a technique is developed to identify the code associated with each feature using the test cases (see Figure 4). The code is then extracted to create a component; finally, the component is inserted back into the legacy system to validate results. Our goal is to incrementally evolve it. The methodology we propose does not reduce the complexity of a legacy system, but it helps to clarify that comp lexity by explicitly defining component interfaces.

The legacy system that is used as a case study is American Financial Systems; Incr.'s (AFS) product called Master System (AMS). AFS is a small (60 employees) software

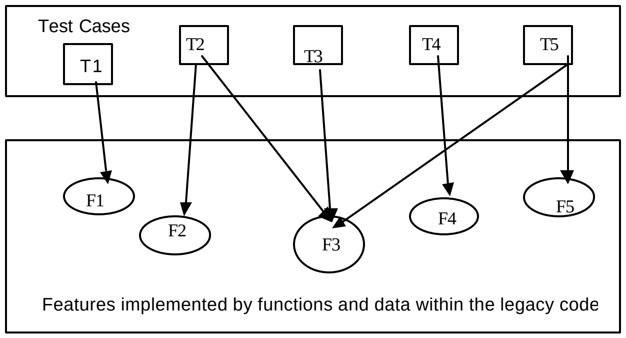
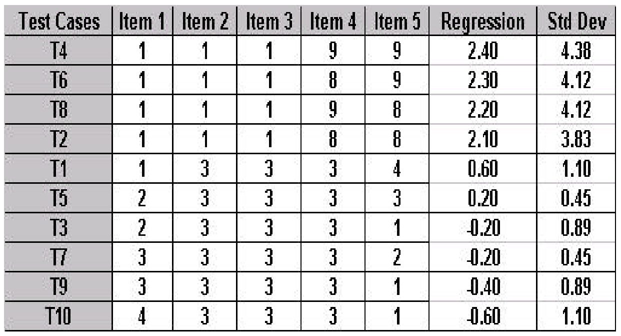

Test Cases
T1
T2
T3
T4
T5
F1
F2
F3
F4
F5
Features implemented by functions and data within the legacy code

Figure 4: Test cases exercising system features

firm that develops software for the COLI (Corporate Owned Life Insurance) market. AFS developed AMS to integrate Life Insurance and Executive Benefits using mathematical and financial modeling. AMS was developed nearly 14 years ago using BASIC. During this time, Microsoft® has evolved BASIC into the more modern programming language, Visual Basic®. Although, AMS is classified to be a legacy system, AFS has also evolved AMS from its original DOS version to use MS Windows. Currently, AMS uses Microsoft Visual Basic 6.0 ® and runs on Microsoft's Windows operating system. We applied the following eight-step methodology to AMS.



Test Cases	Item 1	Item 2	Item 3	Item 4	Item 5	Regression	Std Dev
T4	1	1	1	9	9	2.40	4.38
T6	1	1	1	8	9	2.30	4.12
T8	1	1	1	9	8	2.20	4.12
T2	1	1	1	8	8	2.10	3.83
T1	1	3	3	3	4	0.60	1.10
T5	2	3	3	3	3	0.20	0.45
T3	2	3	3	3	1	-0.20	0.89
T7	3	3	3	3	2	-0.20	0.45
T9	3	3	3	3	1	-0.40	0.89
T10	4	3	3	3	1	-0.60	1.10

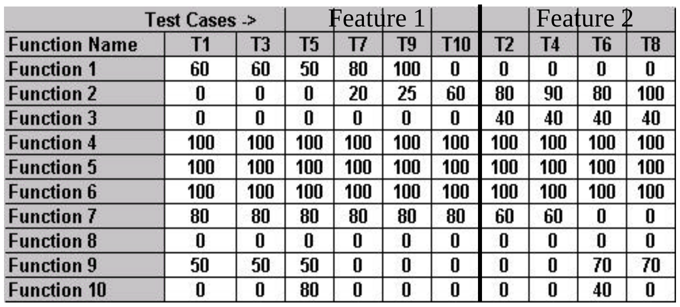
Figure 5: Test cases and Items relationship

3.1 Prioritize evolution reasons: While it is theoretically possible to determine an optimal evolutionary path, we suggest instead that the engineers prioritize their reasons for evolution, including technical as well as marketing. In the same way that requirements are prioritized [16]], we suggest that a clear and concise list be developed that can dictate the evolution efforts. In this paper, we apply the methodology in evolving the *error processing* and *assignment* section of the AMS code. It was brought to the developer's notice that this area was problematic because a simple fix or a new enhancement in that area was taking unreasonable amount of time and often times the fix needed to be resent to the developer. *Error processing* is a feature within AMS that validates user input. Since there are about 400 user inputs (or *Items*) and many of them dependent upon each other, this area was a natural candidate for evolution. Upon closer examination we found that in addition to the *error processing*, the source code also made several *assignments (user data is stored as strings and is then assigned to integer, float or array for later use).* Thus, the evolution of *error processing* in AMS involved two features, *error processing and assignments. Error Processing* is visible to the user but the *assignments* are made within the system so they are invisible. In addition, *error processing* for a single item may be dependent upon other items. For example within the AMS system, the retirement age item value of an individual cannot be less than policy issue age item, when a user enters the policy issue age then retirement age must be checked for the above mentioned error condition. Conversely, if retirement age is the input item then it must be checked against the policy issue age for the error condition. There are more complicated dependencies within the AMS items, for space reasons we are unable to list them all. Finally, a proper message must be displayed indicating the problem with the input item in question. User is also given a GUI to correct the problem. Clearly the interdependencies between the items is so called the feature interaction problem mentioned in Section 2.1.

3.2 Logically arrange features to be evolved: Once the features are associated with their test cases, we order the features to be evolved to minimize the interference between them. The test cases used in this step can be viewed as the representation of the AMS data model. The AMS data model is a simple hierarchy of plan, employee and policy level information where a plan can have many employees and an employee can have many life insurance policies. A group of employees are part of a plan. Information regarding the plan is stored in the *Master File Table*. The *Master File Table* contains the default input for the entire plan. These input fields are called *Items*. The employee information is stored in the *Census File Table*. This information (*Items*) can be varied for each employee in the plan by indicating that the *Master File Item* belongs to the *Census File Table*. This association allows a set of *Items* be varied for a group of employees. For example, if a given plan has 3 employees who have everything in the plan the same except of their ages. Then the *Master File Items* in this case will contain the same information for all the *Items* except that the ages will be stored in the *Census File Table*. There are about 400 Items in the AMS and about 75% of them can be varied from employee to employee. A test case is a combination of *Master File* and *Census File data*. There are about 250 test cases in the AMS with an average size of 10 employees per test case. This step in the methodology provides heuristics on how to logically arrange features (using test cases) that needs evolution. We have identified the following three areas that can help detect interfering features:

3.2.1 Domain Knowledge: Using domain knowledge it is possible to identify test cases that represent a particular feature or a group of features. We found that in many cases the testers knew exactly which test cases would execute specific functionality in the code.

3.3.2 Documentation: Legacy systems also have rich regression test suites that consist of hundreds of test cases. Often these test suites are well documented and they are already grouped by the functionality that needs to be tested.



Test Cases ->		Feature 1					Feature 2			
Function Name	T1	T3	T5	T7	T9	T10	T2	T4	T6	T8
Function 1	60	60	50	80	100	0	0	0	0	0
Function 2	0	0	0	20	25	60	80	90	80	100
Function 3	0	0	0	0	0	0	40	40	40	40
Function 4	100	100	100	100	100	100	100	100	100	100
Function 5	100	100	100	100	100	100	100	100	100	100
Function 6	100	100	100	100	100	100	100	100	100	100
Function 7	80	80	80	80	80	80	60	60	0	0
Function 8	0	0	0	0	0	0	0	0	0	0
Function 9	50	50	50	0	0	0	0	0	70	70
Function 10	0	0	80	0	0	0	0	0	40	0

Figure 6: Test cases, Functions and Feature relationship

3.2.3 Clustering and textual pattern analysis: Related test cases that exercise a feature of closely related feature can be clustered. There are several clustering techniques described in the literature. According to [32]:

C*lustering 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*.

The purpose of our research is not to explore the clustering techniques but to use them creatively. Jain and Flynn survey existing clustering techniques that can be used to group related test cases [32]. 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 we selected 10 test cases and identified 5 sets of items that are considered the most important user inputs in AMS. We analyzed the user input and assigned an ordinal value to each of the valid user input for a given *Item*. For example, if item number 1 had ten valid user input then the user input was given a numeric value of 1 through 10 respectively. We created a matrix of test cases and *Items* as shown in Figure 5. We then used Microsoft Excel™ to calculate the statistical measures that provide insight on potential clusters of related test cases. For example, if we consider two test cases T4 and T6 (assuming that all the other items are exactly the same and only items 4 and 5 vary) we calculate the regression and standard deviation values to find the best fit lines. It is easy to see that test cases T4, T6, T8, and T2 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 group the test cases into two broad groups: group 1 that exercises 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 (Figure 5). We found that grouping these test cases into broad categories simplifies the evolution process by reducing the feature interaction problem.

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

Figure 7: Test Clusters

To identify interrelated and dependent items we used a combination of the three techniques mentioned above. We collected test cases considered by the testers to be relevant for the error processing feature within AMS. We also looked at existing documentation to see what problems were encountered and the solutions sought. Finally, although the first two techniques gave us good intuition regarding the feature interactions, we verified our intuition by clustering the test cases (see Figure 7) that were used to test a particular set of item(s) and their dependencies.

3.3 Locating System Features using Regression Test Cases: Besides validating marginal changes in regression testing, the test cases for a legacy system can be viewed as one of the primary source of information about the features that are most important to the end users. This is particularly true for AMS because end-users input their requirements in these test cases. Test cases are a repository of inputs that exercise the system features. In this step we provide techniques to data-mine this repository and develop heuristics for evolutionary purposes. As the regression test suite increases in size, more and more test cases are used to exercise the stability of system features from one version to another. The goal of this step is to identify the test cases that are correlated to the features we want to evolve. Figure 4 shows, for example, how test cases T1-T5 exercise features F1-F5. A single test case may exercise many features and vice versa.

We instrument the source code with code-coverage software. We run the regression test. We then analyze the path covered. Finally, we develop heuristics to group related test cases together that exercise a particular feature for evolutionary purposes.

The code coverage tool that we used is called *TrueCoverage*™ from *NuMega*®. *TrueCoverage*™ works with many programming languages such as Microsoft Visual Basic, Java, C++ and some scripting languages such as Jscript and VBScript. To instrument the source code we compiled the source code image with *TrueCoverage*™. Since the regression testing is already being done using batch mode it was easy to get the instrumented output against the entire 246 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 more standard file formats (comma-separated values) so that we can then import them in a spreadsheet tool for further analysis. The *TrueCoverage*™ tool has a *merge utility* that aggregates all the 246 test cases that were instrumented. This *merge utility* revealed that 95% of the code was covered using the 246 test cases. We are in the process of identifying whether the rest of the code is either unused or there are hidden features within the system that are not currently being exercised. The *TrueCoverage*™ tool provides the following information on each of the regression test cases:

- Function name – Name of the function that got executed.
- % lines covered – Percentage of lines in the function that were executed
- Called – Number of times the function was called
- # of lines not executed – Number of lines that were not executed
- Total # of lines – Number of lines in the function
- Image – Name of executable, DLL or OCX that contains the function
- Source – Name of source file that contains the function
- Address – Relative virtual address of the function

For our analysis, we selected two columns: *Function name* and *% lines* covered for each of the test cases that represent features to be evolved. We sorted the data based upon the function name column for each of the 246 test cases by developing a simple utility that combined all 246 test cases. We then calculated standard deviation on the entire matrix. Figure 6 shows partial results due to space reasons. The matrix is sorted based on the standard deviation column. The function column is the function that got executed and it is preceded by the module name. Each of columns after the function column represents the % covered for that particular test case. Consider the evolution of two features, Feature 1 and Feature 2, each represented by test cases {T1, T3, T5, T7, T9, and T10} and {T2, T4, T6, and T8} respectively, we deduce the following results from the data in Figure 6:

- For example, a standard deviation of 0 means either that all the functions in all test cases were executed or none of them were. This analysis helps identify unused code within the system and possible hidden features.
- Function 1 totally belongs to Feature 1 and likewise function 3 belongs to Feature 2.
- Functions 4, 5, and 6 appear to be 100% common to the two features that we consider for evolution. These are potentially part of the system core. The concept of core is defined in the next section.
- Functions 2 and 7 are a potential for the feature interaction problem (see Section 2.4) because parts of function 2 are exercised by Feature 1 (test cases, 7 and 9). Likewise, all of Feature 1 test cases and some of Feature 2's test cases exercise function 7.
- Function 8 is not used by any of the test cases while function 3 is used by Feature 2.

Applying the aforementioned technique we identified following problems in the error processing part of AMS:

1. Circular dependencies: As Figure 7 illustrates that item 9 is dependent on 119 and 119 is dependent on 13 that in fact is dependent on 9. We found about 8 circular dependencies. The circular dependencies were the cause of system hangs as we verified this in the AMS' bug tracking system.
2. Readiness of dependent items: To solve the circular dependencies and determine what state an item is during assignment we found that original architects used an array called UNREADY(), which meant that if an item is dependent on another item and the other item still needs to be evaluated then the original item was identified in UNREADY state. Each item had a ready state (1) and an unready state (2). The following code illustrates the issue at hand. We show a partial listing due to space reasons, in the code below item number 5 is assumed to be ready by setting the UNREADY array index. The item's value is then evaluated and a global error flag is set to 1 in case of invalid input. The UNREADY state is set to the error flag's value indicating that the item is in fact not ready. Since items are processed sequentially, if another item that is dependent upon 5 needs its value then the calling item will use the UNREADY array with an index of 5. The implicit setting of item state resulted in many problems such as bad patches to solve circular dependencies.

```
nUnready(5) = 1
Call Fix_Date(nItem)
If nError_F > 0 Then
  nUnready(5) = nError_F
  Exit Sub
End If
```

3. Assignment intermingled with error processing: As items were evaluated for dependencies and error conditions, the original program also set the values to the internal program variables. Due to the nature of AMS data, time series is often used. An example of a time series is 100,1,200,5 which means that starting from year 1 through 5 use 100 and from year 5 and onwards use 200 as an input for certain items. Time series presents some complicated problems because the data needs to be evaluated over a period of time and thus errors can be present in any of the years. Coupled with circular dependencies we found that internal assignments were inconsistently used with error processing.

3.4 Refactor code: Once we have identified the functions that implement the features that need evolution we begin refactoring the code. Typically, refactoring will result in low coupling and high cohesion. Refactoring will result in the removal of global variables and explicit communication rather than implicit communication across system functions. The refactoring may require extensive analysis, especially if two or more features interact or interfere within a given source function.

For the error processing and assignment problem we refactoring by taking following steps:

1. Removing the UNREADY array: The UNREADY array was used implicitly and was tightly coupled with assignments. Instead, we used a component that accepted a collection of errors. Then we developed routines to access (add, display and delete) the collection for one individual or the entire census data. This collection was then passed to the GUI that displayed errors to the users.
2. Replace recursive call with sequential calls to evaluate each item: In the original system, items were checked for error condition and assignments were made using recursion. In one routine the items were listed using the "Select Case" statements, so in case one item needed to check dependencies for another item a recursive call was made.
3. When working with a given item in both assignment and error processing we established a protocol that no other items will be processed.

These design decisions forced us to think about the core.

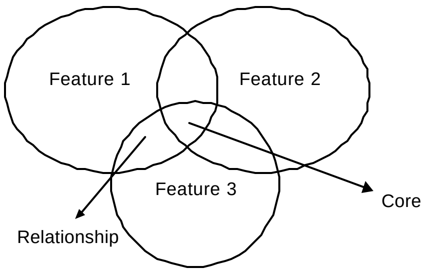


Figure 8: Example of System Core

3.4.1 Identify core by identifying relationships between features to be evolved: If more than one features is to be evolved, it is important to evaluate the relationship between them. The possible relationships were discussed earlier in Section 2. Indirect relationships are typically found in the problem domain. Direct relationships are found in the solution domain. These relationships can arise at various points in the software development cycle. The generalization, specialization, and composition are part of the problem domain and they are also more abstract in nature. The other relationships can arise in either the problem domain or in the solution domain, but for refactoring purposes they are part of the solution domain.

It seems natural to ask the question: "What else is a system comprised of besides features?" Software systems include underlying infrastructure to support and implement their features. Turner identifies this infrastructure as "the core" [3]. This infrastructure exists solely within the solution domain. Users are generally not concerned with the core, and therefore it is not directly reflected in the requirements. The core is often composed of control structures, protocols and communication mechanisms that cannot be traced back to any feature at the requirements level. Chen, Rosenblum, and Vo [17] make an observation about the existence of feature components and core components; core components are exercised by all test cases, whereas feature components are those exercised by only a subset of the test cases. We will use this definition of core.

The concept of core is also mentioned in feature-oriented domain models, although in this context it relates more to the properties of some features [18]. The FODA model defines the core to be what remains of the system in the absence of features. We identified earlier this to be the underlying infrastructure. Our methodology is not about re-architecting the legacy system to impose a radically new vision of the software. Our primary goal in this step is to identify features that are not part of core by factoring out code that is common to all test cases.

For example, Figure 8 shows three features to be evolved. Each of the features is implemented in the code represented as a circle. The intersection shown in the figure is the core. Running the code profiler tool with the test cases that implement these features can identify this intersection. Features tend to be cross cutting in implementation. Refactoring will bring together code related by features into well defined, cohesive units with clear interfaces.

For the error processing and assignment evolution we found that the core consisted of following routines:

1. Routine to determine age based on date of birth or simply a digit
2. Routine for evaluating time series
3. Add Error Routine
4. Routine to evaluate a particular entry in the string (other than time series)

The old code for a given item looked like following:

1. *Set the UNREADY Flag*
2. *Do Assignment*
3. *Go to dependent items and reset the UNREADY state.*
4. *Evaluate error flag from dependent item then do recursion*

The new code for assignment looked like following:

1. *Set core items*
2. *Do Assignment for Item 1*
3. *Do Assignment for Item 2*

The error processing looked like following

1. *Set core items*
2. *Check Errors on Item 1 by calling core functions*
3. *If error is returned then add Item to collection including error*
4. *Do next item (sequentially)*

Figure 9 illustrates our findings regarding the feature-function interaction among the items.

Items	Feature-Function Interaction
9,5,13,119	SSF
9,5	DD
19,158	SFD

Figure 9. Feature-Function Interactions

3.5 Create components & Disable old code: Once the code is factored, we create components from that code. We expect that features encapsulated in components will be easy to maintain and evolve. We will initially use Microsoft's Component Object Model (COM). Once extracted, the old code is disabled, for example, using compiler directives.

3.6 Plug the component back in and verify behavior: Once the old code is disabled, we plug the component back into the legacy system. In essence we are evolving the legacy system into a component-based system. With our approach, the same test cases used in Section 3.2 can be run to compare the results before and after the evolution.

3.7 Verify evolutionary reasons: This is a longer-term data gathering and validating step. Once the legacy system has evolved using this methodology, we propose that the evolutionary result be measured against the expectations. This step usually will result in formal and informal data gathering regarding performance of the evolved system. This step also validates the reasons of why the evolution process was started in the first place.

Our proposed methodology is programming language and platform independent. It makes some basic assumptions about availability of code profiling tools, requirement management tools and domain expertise needed. Since the results of the evolution process can be verified easily, we believe that this methodology has a good chance of being successful within the practitioners.

4. SOFTWARE EVOLUTION – CURRENT TECHNIQUES

Software evolution is a broad term that covers a continuum from adding a field in a database to completely re-implementing a system. These evolution activities can be divided into three categories: maintenance, evolution, and replacement [1,21]. Repeated system maintenance supports the business needs sufficiently for a time, but as the system becomes increasingly outdated, maintenance falls behind the business needs. The evolution effort required represents a greater effort, both in time and functionality, than the maintenance activity. When a system can no longer be evolved, it must be replaced.

Determining the category of evolutionary activity that is most appropriate at different points in the life cycle is a daunting challenge. Should maintenance continue or should the system be modernized? Should the system be replaced? To make the correct decision, the legacy system should be assessed and analyzed to consider the implications of each action. Ransom describes an assessment technique for determining if a legacy system should be replaced, modernized or maintained [23]. Organizations can simply use Ransom's technique to determine whether they need to replace, modernize or maintain their legacy systems. For the purpose of this research we will assume that the legacy system in question (AMS) needs evolutionary efforts.

This research focuses on one aspect in the life of a system: software evolution. The primary focus will be on the white-box evolution technique because this technique makes it possible to trace features to particular function(s) in the code and then carve the source code to create components.

5. CONTRIBUTION AND RELATED WORK

Although CBSE provides viable techniques to develop modularized software systems, these solutions focus primarily on the solution domain and therefore do not help to bridge the complexity gap because CBSE techniques often focus on constructing components from scratch rather than reengineering them from within the legacy code. Recent approaches to evolution within CBSE, such as ArchStudio [24], 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 component [19]. 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 [20]. However, Software Architecture does not provide an efficient solution for legacy system evolution.

In addition, we are encouraged by results from our prior work [3,4] where we converted a standalone executable into a component to evolve overall system architecture that resulted in a better maintenance platform for AMS [7], the feature rich legacy system that we are considering for our case study

While there are some techniques [22,33-37] to locate program's features using execution slices exist, they all assume that valid sets of input data (or test cases) are available at hand and are predominantly used for system debugging rather than evolution. An opposing argument is often times the regression test cases are undocumented but are still part of the regression testing because testers are afraid they might miss testing a feature. Not to mention it is not always possible to know what group of test cases will exercise a given feature(s). It is also unclear as to how the existing techniques define the features and what feature model is used. Our methodology suggests using any code-profiling tool that is available to the developers. χSuds [25] tool can be used to identify the program features, however it is limited to C. We have developed a rich feature model that considers the issue of feature/function interaction (see Section 2). In addition, the existing techniques certainly do not consider evolution in mind as the primary goal.

Similarly, object oriented methodologies attempt to bridge the complexity gap by use cases. Since use cases are not represented in the requirements in a cohesive manner, they do not represent the end user's perspective clearly. In the end, the use cases are simply used as a tool for the developer, which remains in the solution domain thereby making no change to the complexity gap.

We believe that there are several benefits of our methodology. First, it addresses the important issue of legacy system evolution in an incremental manner. Second, it bridges the gap between the problem and the solution domain by mapping the features that the end user sees using regression test cases, to the functions in the source code that a developer sees. Third, it recommends using existing tools to carve out the code related to feature(s). Fourth, it recommends using the existing CBSE techniques to construct the components thereby saving resources. Fifth, it has provisions for validating and verifying the changes made so one can measure success.

5.1 FUTURE WORK

We are currently applying the second part of our methodology to AMS, a legacy system with rich sets of test cases, historical data and features. We are also developing a cost model to measure results.

REFERENCES

1. N Weiderman, J Bergey, D. Smith, B. Dennis and S Tilley. “Approaches to Legacy System Evolution” (CMU/SEI-97-TR-014). Software Engineering Institute, Carnegie Mellon University, 1997.
2. D. Smith, H. Muller, and S. Tilley. “The Year 2000 Problem: Issues and Implications” (CMU/SEI-97-TR-002, ADA325361). Software Engineering Institute, Carnegie Mellon University, 1997.
3. C. Turner, A. Fuggetta, and A. Wolf. “Toward Feature Engineering of Software Systems”. Technical Report CU-CS-830-97, Department of Computer Science, University of Colorado, Boulder, Colorado, February 1997.
4. L. Raccoon. “The Complexity Gap”. *SIGSOFT Software Engineering Notes*, 20(3), Jul. 1995, pp. 37-44.
5. H. Kaindl, S. Kramer, and R. Kacsich. “A Case Study of Decomposing Functional Requirements Using Scenarios”. In *Third International Conference on Requirements Engineering*, pp. 82-89. IEEE Computer Society, Apr. 1998.
6. S. Tilley, and D Smith, “Legacy System Reengineering, Software Engineering Institute”, Carnegie Mellon University, Presented at the International Conference on Software Maintenance, Nov. 4-8, 1996.
7. S Comella-Dorda. K Wallnau, R. Seacord, and J Robert. “A Survey of Legacy System Modernization Approaches”. SEI Technical Note CMU/SEI-00-TN-003. Software Engineering Institute, Carnegie Mellon University, Apr. 2000.
8. A. Davis and R. Rauscher. “Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications”. In *Proceedings of the 1979 Conference on Specifications of Reliable Software*, IEEE Computer Society, 1979, pp. 15-35.
9. A. Davis. “The Design of a Family of Application-Oriented Requirements Languages”. IEEE Computer, 15(5), May 1982, pp. 21-28.
10. *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standards Collection, Software Engineering, IEEE, New York, NY. 1994.
11. M. Weiser. “Program Slicing”, In Proceedings of the *5th International Conference on Software Engineering*, IEEE Computer Society, Mar. 1981, pp. 439-449.
12. D. Parnas. “On the criteria to be used in decomposition systems into modules”, *Communications of the ACM*, 15(12): 1053-1058, Dec. 1972.
13. J. Field, G. Ramalingam and F. Tip, Parametric program slicing, *Papers of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995, pp. 379-392.
14. R. Pressman. *Software Engineering: A Practitioner's Approach, 4th Edition*. New York, NY: McGraw-Hill, 1997
15. Martin Griss, Implementing Product-Line Features with Component Reuse, Proceedings, *6th International Conference on Software Reuse*, Springer-Verlag, Vienna, Austria, Jun. 2000.
16. D. D'Souza and A. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, Reading, MA, 1999.
17. Y Chen, D. Rosenblum, and K. Vo. “Test Tube: A System for Selective Regression Testing”. In *Proceedings of the 16th International Conference on Software Engineering*, IEEE Computer Society, 1994, pp. 211-220.
18. C. Kop and H. Mayr. “Conceptual Predesign Bridging the Gap between Requirements and Conceptual Design”. *In 3rd International Conference on Requirements Engineering*, IEEE Computer Society, Apr. 1998, pp. 90-98.
19. D. Garlan and M. Shaw, “An Introduction to Software Architecture”, *Advances in Software Engineering and Knowledge Engineering, Volume I.* World Scientific Publishing, 1993.
20. N. Medvidovic and R. Taylor, "Separating Fact from Fiction in Software Architecture", *3rd International Workshop on Software Architecture*, Edited by Jeff N. Magee and Dewayne E. Perry, Orlando, Florida, Nov. 1998, pp. 105-108.
21. N. Weiderman, D. Smith, S. Tilley, and K. Wallnau. “Implications of Distributed Object Technology for Reengineering” (CMU/SEI-97-TR-005 ADA326945). Pittsburgh, PA, SEI, CMU.
22. N. Wilde and M. Scully. “Software reconnaissance: Mapping features to code,” *Software Maintenance: Research and Practice*.

23. J. Ransom and I. Warren. “A Method for Assessing Legacy Systems for Evolution,” Proceedings, *Second Euromicro Conference on Software Maintenance and Reengineering* (CSMR98), 1998.

24. P Oreizy, N Medvidovic, and R. Taylor, “Architecture-based runtime software evolution”, Proceedings, *International Conference on Software Engineering*, Kyoto, Japan, Apr. 1998.

25. χSuds User’s Manual, Telecordia Technologies, 1998.

26. H. Agrawal, Horgan, J Krauser, E.W., and London, S.A. “Incremental regression testing”. In *Proceedings of the IEEE Software Maintenance Conference* (1993), pp. 348–357.

27. H. Leung, and L. White. “Insights into regression testing”. Proceedings, *IEEE Conference on Software Maintenance*, 1989, pp. 60–69.

28. G. Rothermel and M. Harrold. “A safe, efficient algorithm for regression test selection”. Proceedings, *IEEE Software Maintenance Conference*, 1993, pp. 358–367.

29. G. Rothermel. And M. Harrold. “A Comparison of Regression Test Selection Techniques”. Tech. Rep., Department of Computer Science, Clemson University, Clemson, SC, Oct. 1994.

30. A. Onoma, W Tsai, M Poonawala, H Suganuma. “Regression Testing in an Industrial Environment”, *Communications of the ACM*. Vol. 41, May 1998.

31. B. Beizer. *Software Testing Techniques*, 2d. ed. R. Norstrand , New York, 1990.

32. M Jain and Flynn. “Data Clustering – A Review”. *ACM Computing Surveys*, Vol. 31, No. 3, Sept. 1999.

33. T. Ball, “Software visualization in the large,” IEEE Computer, Apr. 1996, pp 33-43.

34. B. Korel and J. W. Laski, “Dynamic program slicing,” *Information Processing Letters*, 29(3):155-163,1998.

35. B. Korel and J Rilling, “Dynamic program slicing in understanding of program execution,” *Fifth International Workshops on Program Comprehension*, Dearborn, MI, May, 1997, pp 80-89.

36. A. D. Malony, D. H. Hammerslag, and D. J. Jabalonski, “Traceview: A Trace visualization tool,” *IEEE Software*, Sept. 1991, pp. 19-28.

37. M Weiser, “Program slicing”, *IEEE Trans. On Software Engineering*, SE-10 (4): 352-357, Jul. 1984.