Evolving Legacy System Features into Fine-Grained Components

Alok Mehta
American Financial Systems, Inc.
9 Riverside Office Park
Weston, MA 02493
1 781 893 3393
amehta@afs-link.com

George T. Heineman
WPI Computer Science Department
100 Institute Road
Worcester, MA 01609
1 508 831 5502
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, regression tests, and component-based software engineering (CBSE). Regression test cases are untapped resources, full of information about system features. By exercising each feature with their associated test cases using code profilers and similar tools, code can be located and refactored to create components. These components are then inserted back into the legacy system, ensuring a working system structure. This methodology is divided into three parts. Part one identifies the source code associated with features that need evolution. Part two deals with creating components and part three measures results. By applying this methodology, AFS has successfully restructured its enterprise legacy system and reduced the costs of future maintenance. Additionally, the components that were refactored from the legacy system are currently being used within a web-enabled application.

1. INTRODUCTION

Increasingly, organizations view their software assets as investments that grow in value rather than liabilities whose value depreciates over time [30]. 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 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 and manage hardware and software obsolescence.

To effectively evolve legacy systems in this fast-paced environment, organizations must answer two questions [25]: What are the critical success factors of system evolution? How do we evolve the system without adversely affecting operations? American Financial Systems (AFS) developed their strategy by pursuing the following two 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'02, May 25-28, 2002, Buenos-Aires, Argentina.
Copyright 2002 ACM 1-58113-000-0/00/0000…$5.00.

Our results show an innovative use of existing regression test suites and give extra incentives for designing 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 within the enterprise.

2. EVOLUTION MODEL

The repeated modification of a 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 must consider modernizing these legacy systems so that they remain viable. 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 software, managerial, and economic perspectives [26][27].

Many software maintenance initiatives do not sufficiently incorporate the user's point of reference [4]; such lack of consideration can leave users unsatisfied and frustrated because users may not see the benefit of these initiatives. Researchers [28][22][11][5] 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 requirements in the form of input files (or the database) that the system uses. Because these users are directly concerned with system functionality, their perspective is always in the problem domain. Composed from input files, regression test cases are used to check the stability from one version of the system to another. In reviewing test cases, developers 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. 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 [22]). 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 the document in a word processing application. While

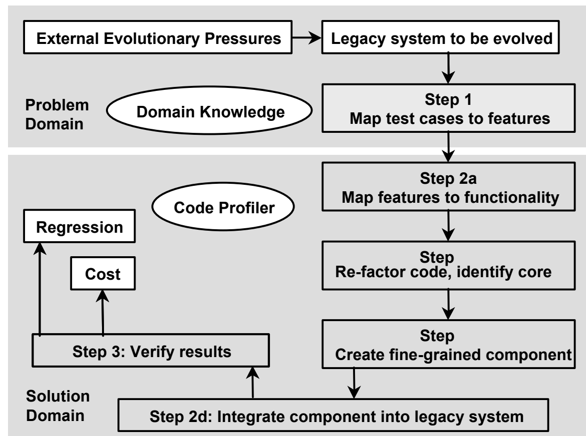

Figure 1: Evolution Methodology.

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. The 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. The end-user may or may not see the changes made to the system but the goal of such refactoring is to reduce future maintenance costs. Our research provides a methodology for handling both external and internal evolutionary pressures.

Researchers have long identified features as a natural organization of the problem domain [5][6]. Surprisingly, few approaches in the research literature concentrate on feature-based organization of a system's functionality. In contrast, the solution domain is full of research that incorporates software artifact management activities such as design, component construction, and testing. Regression test suites are an untapped resource for software evolution because they tell a legacy system's story in a way that can be used to identify features of interest to end-users. We show how to identify the code associated with features, extract that code, and create fine-grained components. These components are inserted back into the legacy system to validate results in two ways. First, we match the output of the regression tests after the insertion with original output. Second, we measure the cost of adding a new feature and compare that to the prior costs. The outline of our methodology as shown in Figure 1 is:

- Step 1: Select test cases by considering features.
- Step 2: Execute selected test cases using code profilers to locate source code that implements features. Analyze and refactor source code to create components.
- Step 3: Compare pre- and post-evolution maintenance costs.

Our methodology has three basic assumptions. First, we assume that the legacy system to be evolved is written using a modern programming language such as Visual Basic, C++, Java, or COBOL; this allows us to employ existing code-profiling tools to trace the source code implementing a specific feature. Second, we assume that the legacy system has regression test suites. Third, we assume domain knowledge and expertise are available, although this is not a binding constraint as discussed in Section 6.

3. FEATURE MODEL

End-users often view a system in terms of its provided features. They exercise the system features through user input (stored in files or databases) that is often used for system maintenance as part of regression testing. Intuitively, a feature is an identifiable unit of system functionality from the user's perspective. Examples of features include the ability of a word processor to spell check or 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 [29]. We developed the following definition by integrating and extending existing definitions [22][29]:

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 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.

Table 1: Feature/Functions Relationship

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

3.1 Features and Functions

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 1 outlines how a feature might be implemented within function(s). In this paper we are concerned only with the first two relationships. When a single feature implementation is contained within many functions then the critical viewpoint regarding evolution is the solution domain because the feature "cross-cuts" the software [9]. 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.

Many researchers have studied regression testing from a theoretical point of view [3][14][19][23][24]. A testing organization accumulates regression test cases for a legacy system

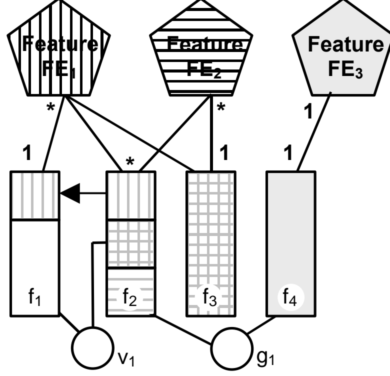

Figure 2: Feature Implementation Interactions.

to ensure the stability of the system over multiple evolutions. Often companies develop proprietary regression testing tools to automate these tests or reduce the total number of tests to execute. We propose a novel use of dynamic slicing [13] during regression testing 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.

3.2 Feature/Function Interaction

In Figure 2 functions are represented as rectangles, variables (both local and global) as circles, and features as pentagons. A feature implementation (FI) is the set of statements within all functions that execute when that feature is exercised. FIs are shaded using the same pattern as their corresponding feature. When two or more feature implementations share common data or functions, there are four key interactions.

SS - Shared Stateless Function: A stateless function [7] 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.

SSF - Shared State-Full Function: A state-full function [7] 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 FI_1 and FI_2 there is an implicit interaction.

DD – Dependent Data: An FI may be dependent on the data accessed 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.

DF – Dependent Function: 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 2) 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 2). 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. Code can then be refactored during evolution.

4. FINE-GRAINED COMPONENT MODEL

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 3. Once we identify FI using regression tests cases, code profilers, and similar tools such as χSuds [1] and NuMega's *TrueCoverage*™[34], we refactor FI into a fine-grained component. As defined in [21]:

*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.*

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

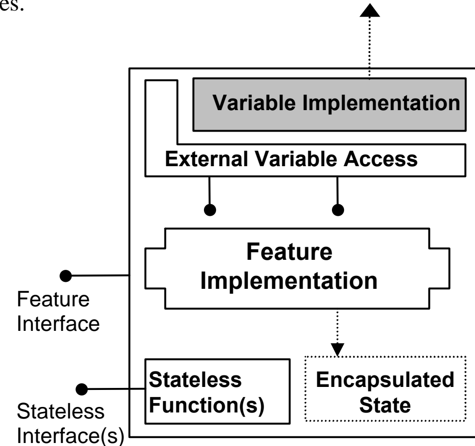

Figure 3: Fine-Grained Component Model.

The first step is to isolate each function that contains code belonging to the target FI. The analysis is often complex because local variables, global variables, and dependent functions can be shared between FIs. 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 shows a single function f_x whose code is shared between implementations FI_1 and FI_2. 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; it is clear that v_4 is set by FE_1. FI_2 also accesses global variable g_2, SS function f_2, and SSF f_3.

$Comp_2$ in Figure 4 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 for accessing data, and contains

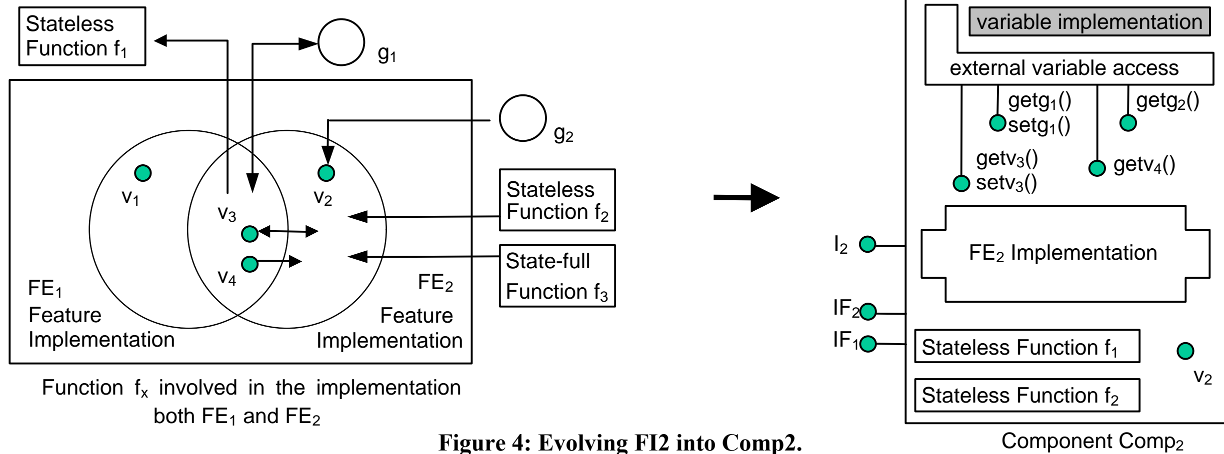

Figure 4: Evolving FI2 into Comp2.

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. Local and global variables used by FI_2 can be accessed via `GetValue/SetValue` methods. Additionally, the `get` method 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 (as shown in Figure 12).

5. CASE STUDY

We applied the three-step methodology outlined in Section 2 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.

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* is a feature that either shows or hides an item in the user interface based upon the input for another item. *Error Processing* is a feature that validates item values. There are 400+ items 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, Float, 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 400+ *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 of nearly 250 test cases with an average size of 10 employees per test case. Running all regression tests executes AMS nearly 2,500 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 45 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 invalidates a constraint, AMS must display a message indicating the specific problem (note that suppressed items are not involved in error processing).

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

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

Figure 5: Fragment for Validating Values for Item 9.

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 5, 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.

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.

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.

5.1 Step 1: Map Test Cases to Features

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. Step 1 of our methodology provides heuristics on how to logically arrange features (using test cases) that need evolution. Once the features are associated with their test cases, we group the features to be evolved with the related test cases for code coverage. The test cases used in this step can be viewed as the representation of the AMS data model. We have identified three means of grouping related test cases to identify feature implementation.

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.

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

Clustering and textual pattern analysis: We find that related test cases (based on input data) exercise closely related features. A simple technique can be used to cluster these related test cases, and there are several clustering techniques described in the the software engineering literature. According to Jain and Flynn [10]:

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.

The purpose of our research is not to explore the clustering techniques but to use them creatively. Jain and Flynn [10] provide a survey of existing clustering techniques that can be used to group related test cases. We created a matrix of test cases and *Items* as shown in Figure 6 and calculated statistical measures (regression and standard deviation) to identify clusters of related test cases. We assigned an ordinal value to each valid plan item value. For example, **item 5** had ten valid user inputs, so its column contains values ranging from 1 to 10. Test cases T4, T6, T8 and T2 can be grouped together; these exercise feature FE_1. Similarly, test cases T1, T3, T5, T7, T9 and T10 can be grouped together because they vary by **item 1** and **item 5**; these exercise feature FE_2. Pattern analysis of item values could also be used to group related test cases by textual input. We found that grouping test cases into broad categories simplified the evolution process.

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 6: Test Case vs. Items.

5.2 Step 2: Refactor and Create Components

Besides validating marginal changes in regression testing, the test cases for a legacy system can be viewed as one of the primary sources 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 using the same format as these test cases. These test cases are a repository of inputs that exercise the system's features. Step 2 of our methodology mines the data in this repository and develops the heuristics for evolution. 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 test cases that are correlated to the features we want to evolve. A single test case may exercise multiple features, so we must take care to identify appropriate test cases.

5.2.1 Map Features to Functions

To locate a feature implementation, we instrumented the source code of AMS (only need to do this once) using code-coverage software and ran all regression tests. We then analyzed the

coverage results and grouped related test cases together that exercised specific features.

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. We calculated the standard deviation on the entire matrix for all 250 test cases. Figure 7 partially shows the matrix sorted by function and standard deviation. Each numeric column represents the percentage of coverage for a function in that particular test case. A standard deviation of zero (not shown for space reasons) means that either a function was executed for all test cases or the function was not executed at all. This analysis helped to identify unused code within the system and possible hidden features.

Test Cases →	T1	T3	T5	T7	T9	T10	T2	T4	T6	T8
Function Name	Feature 1						Feature 2			
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 7: Function vs. Test Case Matrix.

We use these numbers to develop heuristics. For example, if we consider evolving 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.

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 identify for evolution. These functions are potentially part of the core. Functions 2 and 7 have a potential feature interaction problem (see Section 3.2) because parts of function 2 are exercised by Feature 1 (test cases 7 and 9). Likewise, all of Feature 1's test cases and some of Feature 2's test cases exercise function 7.

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

Circular dependencies: As Table 2 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 bug tracking system for AMS.

Table 2: Example of Circular Dependencies

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

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 8 illustrates that: **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 8: Dependent Items.

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*.

5.2.2 Refactor Code and Identify Core

Once we identify feature implementations, we refactor the code as outlined in Section 4. 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.

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

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.

Replaced recursive calls with sequential calls to evaluate items: In the original system, *Error Processing*, *Suppression* and

Assignments were largely recursive. Essentially, a single large routine 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

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.

5.2.3 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 3. (EP/S means variables involved both in EP and S).

Table 3: Variable Analysis (Pre/Post Evolution)

Var.® Comp. –	G	L	SS	SSF	*Get Value*	*Set Value*	*L*	*AFS Core*
EP	35	5	4	2	25	10	5	6
A	14	8	6	4	10	12	6	4
S	50	5	8	5	55	5	4	4
EP/A	11	3	3	3	8	6	2	4
EP/S	20	5	4	3	17	8	3	4
A/S	25	6	3	2	18	12	4	2
EP/A/S	8	9	2	2	6	7	4	4

When creating fine-grained components, these variables and functions become properties of a component. The first two columns in Table 3 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.

Feature Composition and Relationships: Turner identified several relationships among interacting features [29]. We expand this concept into direct and indirect relationships among interacting features, and also add a dependent relationship as a part of a direct relationship. Within the indirect relationship a feature may be a *composed, generalized or specialized* part of another feature. This is typically an end-user's view. For example, *Input Processing* is composed of *Error Processing*, *Suppression* and *Assignments* sub-features. Within the direct feature relationships, a feature relationship with another feature may be that of *dependent, altered, required, conflicting, and competitive.* 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 9 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 9: Dependent Feature Example.

Required: The function in Figure 10 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 declare this interaction.

```
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
```

Figure 10: 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 11 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 11: Altered Feature Example.

Once feature relationships and properties are determined we can create the component's interface. This is shown in Table 4.

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 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.

Table 4: Component Interface (Partial Listing)

Component	*Interface*	*Methods*
Assignments	clsAssignment	Assignments
Error Processing (EP)	clsErrorProcessing	ErrorChecking
EP Core	clsEProcessingCore	AddError ClearError RemoveError RemoveErrorForSuppressedItem ClearAllErrors
Suppression (S)	clsSuppression	Suppression
S Core	clsSuppressionCore	SuppressTheItem UnSuppressTheItem SetTheSuppressCodes
AFS Core	clsAFSCore	Too many to list (42 in all)

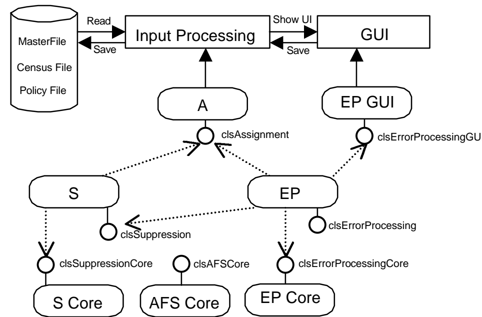

Figure 12: Input Processing Component Infrastructure.

5.2.4 Integrate Fine-Grained Components into AMS

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 integrated component is shown in the Figure 12.

5.3 Step 3: Measure Results

The changes we made to the system were validated in two ways: First, a regression test of *Input Processing* was performed to compare data after the evolution of these three features. Using the batch facility of AMS, we verified that the text file was identical to the one generated before evolution. Second, our initial evolution reasons, as listed earlier, were validated.

1. **The system-locking problem**: The component-based implementation is a linear solution. In all three features, core items are evaluated first and then each item is individually evaluated. Previous communication through global variables was replaced with interactions between component interfaces.

2. **Cost of adding a new item**: The average time to add a new item and code all the relevant *Assignments*, *Error Processing* and *Suppression* logic took 3 days prior to applying the evolution methodology. After the system was evolved, we collected data on adding 4 new items and the average time spent was about 1.25 days.

3. **Reusability between AMS and the web version of AMS**: There were six resulting components from this evolution exercise: *Assignments*, *Error Processing*, *Suppression*, *Error Processing Core*, *Suppression Core* and *AFS Core*. While *AFS Core* is being used in all AFS product lines (a total of 4 different projects), the other five components are used in both the desktop and Internet platform of AMS.

6. Lessons Learned

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

Selecting Evolvable Features: 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. If an evolvable feature is spread out across many functions, and if the code execution is below 50% 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. Typically, life insurance acts as an asset to fund the executive benefits. There are many legal-, accounting-, insurance- and benefits-related constraints that play a very 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 and heuristics tell 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, or depend on or share global variables as a means of communication.

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.

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 very important from a business perspective. An informal survey of 7 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.

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.

Features and Code 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*).

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 4 AFS projects, 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`.

Performance and Security: In refactoring the recursion into linear functions, the performance of AMS was unaffected. We observed a 4% decrease in execution time once *AFS Core* was introduced. We attribute this improvement to the removal of global variables and in-line code. Because the global variables were reduced by 6% (AMS has over 150 global variables), the system is more secure in terms of memory consumption since COM+ offers better security and performance when components use fewer global variables. Although COM+ is not being used in the desktop version of AMS, there is significant benefit for using components under COM+ for the web-based version.

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.

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 5, 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 4 of its product lines (an unexpected side effect).

Table 5: Budget Analysis

Effort	*Cost (+)/ Savings (-) in Months*
Map Features and Test-Cases	+1
Identify code and Refactor	+3
Component Creation (EP, S, A, Cores)	+4
Testing, Training, Documentation, Configuration and Project Management	+3
Savings from solving specific problems	Data being gathered
Reuse (AFS Core in 4 projects and other components in dual-platform)	-10
Net (Cost (+)/Savings (-))	+1

7. RELATED WORK AND CONCLUSION

Our work is closely related to the following areas of software engineering: CBSE, Feature Engineering, Separation of Concerns and AOP.

Although CBSE provides viable techniques to develop modularized software systems, the components are often designed and implemented from scratch rather than reengineering them from within a legacy system. Recent approaches to evolution within CBSE, such as ArchStudio [20], 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 [8]. 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[16]. However, Software Architecture does not provide an efficient solution for legacy system evolution. In addition, we are encouraged by results from our prior work [17][18], where we converted a standalone AMS executable into a component that evolved overall system architecture resulting in a better maintenance platform for AMS, the feature-rich legacy system that we used for our case study.

While there are techniques[2][13][15][31][32] to locate program features using execution slices, they are predominantly used for system debugging rather than evolution. A contribution of this paper is to provide a practical model for features that can be used in conjunction with slicing. Our methodology suggests using any available code-profiling tool. The most closely related technology is the χSuds [1] tool that can identify program feature in a C program. Our innovative contribution is showing how to construct a reusable fine-grained component from the feature.

The SEI FODA feature model ties business models together by structuring and relating feature sets [9]. FODA framework explores how this structured information can be leveraged across the software development effort. Griss [9] extended the FODA methodology to create an explicit feature model of functionality to facilitate reuse-driven software engineering. We agree with Griss that a feature model integrates the viewpoint of both the user and the developer; in this paper we show the practical application of this integrated perspective.

The feature interaction literature is primarily focused on telecommunications networks [28]. Telecommunications networks are massive, complex, distributed systems that incorporate 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. Telecommunication 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 is intuitive and easily applicable for evolution purposes.

Two theories related to our work are the separation of concerns and Aspect-Oriented Programming (AOP). There are a number of dimensions of concern that might be of importance for different purposes (such as comprehension, traceability, reusability, and evolution potential), for different systems, and at different phases of the life cycle. There is an increasing focus on ways to encapsulate multiple overlapping and interacting concerns. Tarr *et al.* admit that a large part of their theory is unproven with an industrial size example [26] 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[12][33]. 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 like 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.

In conclusion, there are several benefits to our methodology. First, it addresses the important issue of legacy system evolution in an incremental manner. Over time, an increasing collection of fine-grained components are extracted from the legacy system. Second, we bridge the complexity gap by mapping problem-domain features using regression test cases and the solution-domain functions in the source code. Third, we use existing code-profiling and similar tools to refactor code related to features. Fourth, by clearly defining a fine-grained component model, we are able to develop software assets with clearly defined interfaces that can be used throughout the enterprise.

7.1 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 their 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. We will also investigate the challenges in applying our methodology when the underlying system is programmed in an object-oriented language such as C++ or Java.

8. ACKNOWLEDGMENTS

We would like to thank Eric Wong of Telecordia Technologies and Lisa Amaya Price of AFS for providing feedback on an early draft of this paper.

9. REFERENCES

[1] χSuds User's Manual, Telecordia Technologies, 1998.

[2] T. Ball, "Software visualization in the large", IEEE Computer, VOL 29 NO 4, Apr. 1996, pp. 33-43.

[3] Y. Chen, D. Rosenblum, and K. Vo, "Test Tube: A System for Selective Regression Testing", Proceedings, 16th International Conference on Software Engineering, IEEE Computer Society, May 1994, pp. 211-220.

[4] S. Comella-Dorda, K Wallnau, R. Seacord, and J. Robert, "A Survey of Legacy System Modernization Approaches", Technical Note CMU/SEI-00-TN-003, SEI, Carnegie Mellon University, Apr. 2000.

[5] A. Davis and R. Rauscher, "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications", Proceedings, Conference on Specifications of Reliable Software, IEEE Computer Society, 1979, pp. 15-35.

[6] A. Davis, "The Design of a Family of Application-Oriented Requirements Languages", IEEE Computer, Vol. 15, No. 5, May 1982, pp. 21-28.

[7] J. Field, G. Ramalingam, and F. Tip, "Parametric Program 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1995, pp. 379 – 392.

[8] D. Garlan and M. Shaw, "An Introduction to Software Architecture", Advances in Software Engineering and Knowledge Engineering, Volume I. World Scientific Publishing, 1993.

[9] M. Griss, "Implementing Product-Line Features with Component Reuse", Proceedings, 6th International Conference on Software Reuse, Springer-Verlag, Vienna, Austria, June 2000.

[10] M. Jain, M. Murty and P. J. Flynn, ACM Computing Surveys, Vol. 31, No. 3, Sep. 1999, pp. 264-323.

[11] 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, Apr. 1998, pp. 82-89.

[12] 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 (ECOOP), June 1997, pp. 220-242.

[13] B. Korel and J. Laski, "Dynamic program slicing", Information Processing Letters, Vol. 29, No. 3, 1998, pp. 155-163.

[14] H. Leung and L. White, "Insights into Regression Testing", Proceedings, IEEE Software Maintenance Conference, 1989, pp. 60–69.

[15] A. Malony, D. Hammerslag, and D. Jabalonski, "Traceview: A Trace visualization tool", *IEEE Software*, Sept. 1991, pp. 19-28.

[16] N. Medvidovic and R. Taylor, "Separating Fact from Fiction in Software Architecture", Proceedings, 3rd International Workshop on Software Architecture, J. Magee and D. Perry, Eds., Orlando, Florida, Nov. 1998, pp. 105-108.

[17] A. Mehta and G. Heineman, "Architectural Evolution of Legacy System", Proceedings, 23rd Annual International Computer Software and Applications Conference, Phoenix, Arizona, Aug. 1999, pp. 110-119.

[18] A. Mehta and G. Heineman, "COTS Integration and Extension", Continuing Collaborations for Successful COTS Development, Workshop held in conjunction with ICSE 2000 Limerick, Ireland, May 2000, pp. 67-72.

[19] A. Onoma, W. Tsai, M. Poonawala, and H. Suganuma, "Regression Testing in an Industrial Environment", ACM Communications, Vol. 41, May 1998, pp. 81-86.

[20] P. Oreizy, N. Medvidovic, and R. Taylor, "Architecture-based runtime software evolution", Proceedings, 20th International Conference on Software Engineering, Kyoto, Japan, Apr. 1998, pp. 62-70.

[21] G. Heineman and W. Councill, *Component-Based Software Engineering: Putting The Pieces Together*, Addison-Wesley, Boston, MA, 2001.

[22] L. Raccoon, "The Complexity Gap", SIGSOFT Software Engineering Notes, Vol. 20, No. 3, July 1995, pp. 37-44.

[23] G. Rothermel and M. Harrold, "A safe, efficient algorithm for regression test selection", Proceedings, IEEE Software Maintenance Conference, 1993, pp. 358–367.

[24] G. Rothermel and M. Harrold. "A Comparison of Regression Test Selection Techniques", Technical Report, Dept. of Computer Science, Clemson University, Oct. 1994.

[25] D. Smith, H. Muller, and S. Tilley, "The Year 2000 Problem: Issues and Implications", Technical Report CMU/SEI-97-TR-002, SEI, 1997.

[26] P. Tarr, H. Ossher, W. Harrison and S. M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns", Proceedings, International Conference on Software Engineering, 1999, pp. 107-119.

[27] S. Tilley and D Smith, "Legacy System Reengineering", Presented at the International Conference on Software Maintenance, SEI, Carnegie Mellon University, Nov. 1996.

[28] 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, Oct. 1998, pp. 818-830.

[29] C. Turner C., 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, Feb. 1997.

[30] 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, 1997.

[31] M. Weiser, "Program Slicing", IEEE Transactions on Software Engineering, Vol. 10, No. 4, July 1984, pp. 352-357.

[32] N. Wilde N. and M. Scully, "Software Reconnaissance: Mapping Program Features to Code", Journal of Software Maintenance: Research & Practice, Vol. 7, 1995, pp. 49-62.

[33] 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 European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), Vienna, Austria, 2001, p. 88-98.

[34] www.numega.com