Easing Software Evolution: Change-Data and Domain-driven Approach

Hitesh Sajnani
School of Information and Computer Sciences
University of California, Irvine
Irvine, California 92617
Email: hsajnani@uci.edu

Ravindra Naik
Tata Research Development and Design Center
Pune, India
Email: rd.naik@tcs.com

Cristina Lopes
School of Information and Computer Sciences
University of California, Irvine
Irvine, California 92617
Email: lopes@ics.uci.edu

***Abstract*—Quick and quality changes to a software application to add new feature or change existing feature, depend largely on the code architecture and its atomically defined responsibilities that map to the features of the application. As the application evolves, it is known that the structure of the code undergoes unexpected modifications and drifts away from its original design, leading to a number of anomalies in the code structure.**

In this paper, we propose a defect and change-data driven approach to analyze the application, and determine the modules that need re-factoring. While improving the code structure by leveraging the domain knowledge is the key, we present the various code smells that hamper the code structure and re-factorings to resolve them. The defect and change-data gathered serves as an evaluation measure for the contribution of the re-factorings for existing applications. We validate the approach by applying to an existing financial system. The preliminary analysis for the case-study reveals that the approach creates meaningful structure from the code, which enables the developers to quickly identify the code that implements a given functionality.

I. Introduction

Every software that is in active use continues to evolve [1]. Requirements often change with time or new requirements are elicited. Code is enhanced to reflect new functionality or partly re-written to incorporate new designs. Code is changed to fix defects. Although these changes make the software evolve continually, if not incorporated carefully, the software is in danger of becoming non-evolvable and expensive to change. Sometimes, it may become difficult to incorporate a desired change without changing the architecture of the code. Many existing systems are in a state where they are indispensable to the business, but unable to cope with pending changes. Is an expensive transformation or even more expensive re-development the only options for such systems? Can something be done for such systems? The goal of our work is to enable the software in such conditon to gracefully adapt to the requested changes, while ensuring that the overall architecture of the system and the structure of the code is retained.

We use defect and change history of the software to determine which kinds of changes are hard to make. The changes and their associated data, when analyzed, help us to prioritize the improvement areas, identifying software life cycle phases that need improvement, and code units which need re-structuring. The analysis involves interpreting trends and percentages, and comparing them with known bench-marks. Next, we detect code smells and re-factor the code to prepare for re-structuring into functional components, followed by applying our domain driven algorithm to attain such functional components. The necessary analysis also detects a number of opportunities to re-factor the code that ensure quick location of features in the code and consistent usage of design patterns.

We describe this process on a Retail Banking Software System maintained by Tata Consultancy Services, which is in active use by a number of banks and is undergoing a similar evolution. It requires changes - changes to incorporate modified business processes, new capabilities and modified products for the bank's customers. Changes are also made to fix defects reported from the field. In general, changes may also be to enable simplicity for use by customers and improve performance, scalability to handle more number of customers, and more such non-functional requirements. The problem that the banking software encountered was that the changes to specific modules (that implement specific functional components) were taking increasingly more time due to the complex nature of the software. For every change requested by the customer in the specific functional components, a set of large programs always needed to be modified, resulting into complex version merging scenarios (due to parallel changes) or delayed releases due to sequential modification of the large programs. This way of developing is the result of many years of software evolution. Continuous changes to the software deteriorate the application structure and architecture, and make subsequent changes expensive, unless specific care and efforts are taken to prevent the deterioration [2]. By using our analytical model built on top of the collected data, we concluded that there is a need to have a closer look at the "large programs", and evaluate the possibility of splitting or re-structuring them into smaller, functionally coherent programs. The contribution of this paper is multifold. It proposes templates for software change data collection, builds an analytical model on top of such data helping to monitor and check the state of the software. The paper proposes a technique leveraging software change data and domain knowledge for re-structuring large-scale source code.

In the subsequent sections, we describe the relevant work, the data collection and analysis, followed by the case-study in short and propose criteria for re-structuring and splitting the programs into functional components. Next, we describe the experiment and the results, followed by the conclusion and the future work.

II. Background

The related work can be divided into two sections. One is the restrucuturing techniques, and the other is the use of change-data to address maintenance issues. Restructuring of software systems is an important problem and there is substantial literature on that topic. J. Neighbors [3] creates cross-references of functions, subroutines and data elements and identifies static and sometimes dynamic interconnections between them to infer the subsystems. The correspondence with the domain-specific components is determined by user confirmation. Anquetil and Lethbridge [4] also identify abstractions of the architecture using both formal and informal entities and relations between them, but cluster modules based on file names. Lung, Zaman and Nandi [5] apply the resemblance coefficients to the data-sets, which comprise of function calls, inheritance and shared features, and calculate resemblance matrix. Based on the degree of similarity, they group similar components, and repeat the process. However, the resulting partitions do not necessarily reflect domain coherence. Mancoridis et. al [10,11] developed Bunch, a clustering tool which creates abstractions using Artificial Intelligence techniques. Marcus et. al [6] use identifier names and internal comments in the code to create a corpus of soure artefacts. Using Latent Semantic Indexing, the user can identify concepts in the source code. In the context of COBOL, Van Deursen and Kuipers [8] identify potential objects by clustering highly dependent data record fields.They assume that record fields that are related in the implementation are also related in the application domain, which is practically a naive assumption. Hutchens and Basili [9] identify potential modules by clustering that is based on data-bindings between procedures. Glorie et. al [12] applied formal concept analysis technique to solve similar probem and found that it is not so much suited for an analysis that is expected to produce a precise, nonoverlapping partitioning of the object set. We found that the techniques that enable users to restructure [4,5,6] and refactor [7] programs are based on identifying the code elements like types, variables, subroutines, files, or informal elements like comments, identifier names, descriptions, and locating the relations, bindings or coupling between them, to infer the partitions. Execution traces are used by [3] to identify dynamic interconnections. The domain information is not captured explicitly in any of these techniques. [14] describes our initial work to split a large code unit based on domain information.

Work related with to mining software repositories to assist the maintenace problem is described in [16-24]. Although research based on mining software repositories is increasingly getting adopted, most of it is carried out with the intention of doing predictive analysis for future defects, changes, securities, and resource management. [17-20] use historical account of past change requests and/or source code changes, with the help of IR techniques to give developer recommendations. Kagdi's approach in [23] does not need mining of past change requests (e.g., history of similar bug reports to resolve the bug request in question), and requires source code change history of only selective entities. Zimmermann et. al [21] apply data mining to version histories in order to guide programmers along related changes: "Programmers who changed these functions also changed....". Breu and Zimmermann [22] apply concept analysis on additions of method calls. This helps developers to become aware of cross-cutting concerns in the code and to refactor them into aspects, which in the long term avoids serious maintenance challenges. To the best of our knowledge, almost all the work in this field uses the version control log for the actual code-change data analysis. Our work leverages the daily data collected for Failures/Change requests in maintenance projects. This gives the bigger picture of the state of the software development phases, improvement areas, and also helps to prioritize proactive refactoring of the software. This is important because you may not want to restructure the part of the software which is not frequently changed. The initial work in this respect is described in [16]. After using the inputs from change and defect data to identify the code units that need re-factoring, we propose an approach to segregate large programs based on the domain functions. This results in multiple programs, each of which performs distinct domain functions, or denotes common library (utility) functions. The resulting programs are smaller and functionally self-contained. We describe our experience of applying the technique in an industrial setting on a large scale application.

III. Maintenance data Schema and Analysis

The objective of measurements of data for software maintenance projects is multifold:

- To prioritize the productivity concerns, and apply solutions to make incremental gains.
- To estimate the number of Failures and Changes in the future, and plan for appropriate staffing and effort distribution, etc.
- To identify the long term concerns and maintenance problems, and evolve approaches to address them.
- To measure improvements on the basis of collected data

We classify the defect-change data into three dimensions. First is the Failure/Change rate and their characteristics; next is the actual resolution process data for fixing the Failure or executing the Change requests; the last dimension is the actual code changes made for the resolution process. By measuring the data and analyzing the patterns and trends, appropriate steps can be taken to improve the productivity of the maintenance activities. In this section we describe the schema of the necessary data to be collected for each of these dimension. We propose an analytical model using this data, which establishes relations with the maintenance problems.

The data to be collected from maintenance projects can be categorized distinctively to perform individual analyses. However, for the incidents that are reported by the client, we recommend collecting data as described in the sub-sections below. The following is the terminology that we used in our work.

Change: *Small* enhancements, usually originating from client
Reported Failure/Defect (RF): Indicates that the execution of the system deviates from its requirements. Development team analyzes the reported Failure, accepts it, or rejects it, or classifies it as a Change.
Failure/Defect: Failure accepted by the development team, for which the development team provides a fix. The reason for a Failure is a Fault.
Severity Class: Represents the per-failure impact on the users. This is usually classified by the user as severity 1,2,3,4 (1: highest, 4: lowest).

Recovery: This is a temporary fix or work around done to bypass the Failure/Fault.
Fault: Represents a bug or an error that usually leads to some Failure. One Fault may lead to many Failures
Resolution: This is the permanent fix which is done by analyzing the Failure, identifying the Fault and rectifying it.

A. Failure rate, Change rate, and classification of data

Table I describes a template to capture the rate of Failures, Faults and Changes per unit time reported in production, and a classification based on the severity, and an approximate measure of size.

Reporting Month	Severity (1-4)	Actual time to resolve the Failures, or develop the Change (S: 1day, M: 1 to 5 days, L: 5 days)	#RF + #Changes	#Failures + #Faults	#Changes

TABLE I: Failure, Fault, and Change rate and classification based on severity and size

Table II describes template to capture rate of Failures, Faults and Changes per unit time reported during user acceptance testing and system testing, and a classification based on the Severity, an approximate measure of the size.

# Failures identified		# Faults identified		# Changes	
User acceptance testing	System testing	User acceptance testing	System testing	User acceptance testing	System testing

TABLE II: Failure, Fault, and Change rate and classification during UAT and ST based on severity and size

Table III describes the data to be collected for classifying the Faults, based on developer's perspective. We recommend that projects collect data for releases already made, since few of the classification attributes depend on the code-changes made.

Release #	Fault#	#RRF	#NRF			
			NRF-P-ICC	NRF-P-IC	NRF-P-SE	NRF-D

TABLE III: Data based on developer's perspective

Note that the data is for Faults that are resolved in the release for which data is collected. The Faults are due to Failures reported in a previous release

Release #: The release in which the Faults are fixed
RRF: Faults that the developer concludes are due to changes in the initially communicated requirements or due to improper communication of the requirements
NRF-P-ICC: Faults that are due to incomplete code-changes made in some previous release. The incomplete code-change may be because the developer could not estimate and identify other code-changes necessary due to the initial code-change
NRF-P-IC: Faults that are due to incorrect code-changes made in some previous release. The incorrect code-change was because the developer did not have a complete knowledge of the code or could not map the requirement correctly into the code
NRF-P-SE: Faults that are due to side-effect causing code-changes made in some previous release. Such code-changes are correct and complete with respect to the given requirements. However, the developer was not able to estimate that his code-change has broken down another functionality of the software. This can be detected only during regression testing
NRF-D: Dormant Faults which cannot be traced back to any previous code-changes. Either the Faults were introduced during the initial development of the software and not detected till date, or due to code-changes made very long ago for which no history is available

We model correlation between the parameters computed using this data and the quality related maintenance issues. Table IV describes the inferences based on the analyses of the Failure and Change rate and classification data:

Failure-Change rate and classification data	**Inferred problem (Quality related)**
Large number of Failures due to code-changes to overcome Faults and Changes # Failures per unit time, # Changes per unit time	Design, Development and Testing of each code change is a problem
Percentage of production Failures as compared to UAT Failures is large # Production Failures * 100 / (# Production Failures + # UAT Failures)	Quality of UAT (regression) testing is a problem UAT tests are not able to detect many Failures
Percentage of UAT Failures as compared to System Testing Failure is large - # UAT Failures * 100 / (# UATFailures + # System Test Failures)	Quality of system testing is a problem systems tests are unable to detect many Failures. Inadequate use of code coverage analysis
Number of system test Failures is large # System Test Failures / (# Changes + # Failures) for which code changes were made	Problems in Design and Development process Inadequate design-level impact analysis, Inadequate knowledge of implementation
Larger percentage of requirements related Faults - # RF * 100 / # Faults. Percentage of Changes coming from UAT is large - # Changes identified during (UAT + System Test) / # Changes	Completeness of requirements or Communication of requirements is a problem area. Inadequate domain knowledge
Percentage of Faults due to incomplete code-changes (to address an earlier Fault or Change) is large - # NRF-P-ICC * 100 / # Faults	Incomplete design-level impact analysis and / or Inadequate knowledge of implementation or Complex code structure
Percentage of Faults due to incorrect code-changes (to address an earlier Fault or Change) is large - # NRF-P-IC * 100 / # Faults	Inadequate knowledge of implementation or Complex code structure
Larger percentage of Faults due to side-effect causing code-changes (to address an earlier Fault or Change) code-changes that breakdown other features - # NRF-P-SE * 100 / # Faults	Regression testing is inadequate or test verification is improper. Improper verification of code coverage during testing
Percentage of dormant Faults is large - # NRF-D * 100 / # Faults	Regression testing is inadequate or test verification is improper

TABLE IV: Failure and change rate classification analysis

B. Resolution process data

Resolution is the process of fixing the Failure, resolving a Fault or executing a Change request. For Failures and Faults the process captures the activities before making the actual changes in the source code. This usually includes reproducing the Failure and locating the Fault. Table V describe the activities for Failures and Faults along with the data to be collected for each activity. In case of Change requests the process includes design, development and testing a change for a feature. Table VI describes the data to be collected for Change requests.

Failure #	Failure Analysis: Reproduce the Failure, Locate the Fault and estimate effort and time to fix the fault	
	Effort	Time

TABLE V: Failure Resolution data

Failure #	Design, develop and unit Test		Code review		System testing		User acceptance testing	
	Effort	Time	Effort	Time	Effort	Time	Effort	Time

TABLE VI: Change Resolution data

For certain projects, few steps outlined above in the resolution process may be merged with other steps, while some projects may have additional steps. We recommend that the data columns be changed in such projects.

Table VII correlates the resolution data with the maintenance issues related to productivity.

Resolution process data	**Inferred problem (Productivity related)**
Large percentage of elapsed time to reproduce the Failures Locating the Fault consumes a large percentage of elapsed time Time required to reproduce and locate faults * 100 / Total time required to complete the fault fixing Efforts required to reproduce and locate fault * 100 / Total efforts required to complete the fault fixing	Improper test environment and/or lack of proper (Failure) test data Inadequate knowledge of implementation Inability to locate fault quickly Complex code structure with heavy interdependency of code modules
Large percentage of time to design and develop the code-changes - Time required to design and develop (unit test and review included) the code changes * 100 / Total time required to complete the fault fixing or change request, Efforts required to design and develop the code changes * 100 / Total efforts required to complete the fault fixing or change request	Inadequate knowledge of the mapping of domain functions to design Inadequate knowledge of the mapping of design to implementation Inadequate knowledge of implementation Complex code structure with heavy interdependency of code modules
System testing consumes a large percentage of the total time and efforts Time required for system testing * 100 / Total time required to complete the fault fixing or change request	Too many test cases, possibly locating the same Faults/Failures Too many false negatives in the regression test results leading to large manual analysis
Large percentage of time and efforts in UAT - Time required for UAT * 100 / Total time required to complete the fault fixing or change request	Lack of quick set-up of integrated test environment

TABLE VII: Failure and change rate classification analysis

C. Code-change data

Code-change data is the data or each code-change done to overcome either a Fault or a Change. Table VIII represents the schema for collecting such data. Note that # LOC changed means number of lines of code modified, new lines of code added, and lines of code deleted effectively. Table IX describes the inferences based on the analysis of code-change data

Failure #	Estimated		Actual		# Code-changes in each program	
	Effort	Time	Effort	Time	Program Name	#LOC changed

TABLE VIII: Resolution data for changes

Code-change data	**Inferred problem**
Large gap between estimated efforts, estimated time and actual efforts, actual time (Actual time Estimated time) * 100 / Actual time, (Actual efforts Estimated efforts) * 100 / Actual efforts	Inadequate functional understanding of the enhancements Inadequate knowledge of the mapping of domain functions to design Inadequate knowledge of the mapping of design to implementation Inadequate knowledge of implementation
Frequent code-changes to the same programs Per program # code-changes in the program to fix all Failures * 100 / # Failures # code-changes in the program to implement all changes * 100 / # Changes	Large functionality or a common set of functionality is implemented in a small set of programs (programs may need to be split or parameterized)
Large number of code-changes to programs Per program # code-changes in the program to fix all Failures * 100 / # total code-changes in all the programs to fix all the Failures	Code structure of programs is too complex or Program is too large

TABLE IX: Code-change data analysis

IV. Case Study Description

The business application is a retail banking software. Primarily transaction-driven, the system is a collection of applications providing various retail banking functions like General Ledger (Credit, Debit, Transfer, Foreign Exchange, Remittances, etc.), Advances, Deposits, General Banking, miscellaneous and shared functions like Fees & Charges, Collaterals, Limits and Exposures, utility functions like date formatting, check-digit validation, and many others.

The core of the system is programmed in MF-COBOL language, running on IBM AIX platform and uses a message-driven architecture.

We performed the data collection and analysis of the data for few other projects too. However, in this paper we use only the retail banking software case-study for illustration.

A. Parameterized Design

The interesting aspect, as outlined by the product manager, is that the system originally had a *parameterized* design, with the parameters providing handles to the developers to reuse existing code to develop feature variants. The parameters primarily represent the *transaction codes* (examples are *Cash Posting, Cheque Deposit, Cash Distribution*) and *transaction attributes*

(examples are *Correction, Backdating, Deposit, Withdrawl*). Unfortunately, as commonly observed with most evolving systems, these parameters are not completely documented, nor uniformly designed. The latest implementation, therefore, only partially complies with the paramterized design.

B. The Problem

The overall system is extremely large, running into few million lines of code that span across few thousand COBOL programs. Figure 1 is the result of data analysis of the efforts spent during the maintenance of the system. The observation is that besides development, the team spends 30% of the time in system testing and only 7% of time on design of the new feature or change request. Clearly, system testing draws attention and should consume less efforts. This could be done by spending more time in designing a change so it is complete and correct. At the same time, based on the failure-change rate analysis in Table IV there may be too many test cases, possibly lead to the same Faults/Failures or/and too many false negatives in the regression test results leading to large manual analysis. Figure 2 shows the distrubution of 454 change requests across various components of the software system. It is observed that for 276 change requests out of 454 change requests, changes were made to a program called ML0000 owned by General Ledger component. This is surprising as only 11% (49 our of 454) changes were tagged for General Ledger component. The above data analysis quickly helped us to focus on functionality of General Ledger and the changes that were required by the customer in General Ledger. By itself, the General Ledger subsystem is about *210K* lines of code, with the largest monolithic program of about *62K* lines of code, having more than *1000 Paragraphs*.

Some of the critical questions posed by the Product Manager were:

- What is exactly the large program (ML0000) doing?
- Can the ML0000 program be split into multiple programs?
- Can the functionality contained in the ML0000 program (and other large programs) be spread into appropriate functional buckets?
- Assuming that the original *parameterized* code structure is violated while making changes, can this be put back into places where such violations have occurred?
- Are the large programs and the potentially complex application architecture of these programs responsible for the fairly high frequency (on an average 200 per month) of reported *transaction outages*?

V. Methodology

Our method consists of following broad steps:

1) Enlisting the high-level and low-level functions/features by studying the functional overview of the software system.
2) Identifyiing the subset of features and sub-features applicable to the program(s) under analysis.
3) Identifying the parameters (if any) used in the code to implement the features and sub-features. The conjecture is that any product implementation makes use of parameters to design and implement features and their variations.

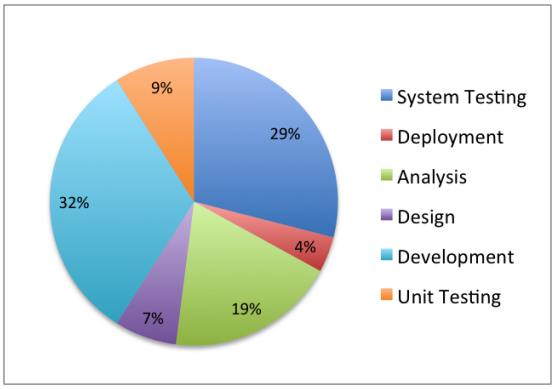

Fig. 1: Distribution of efforts spent

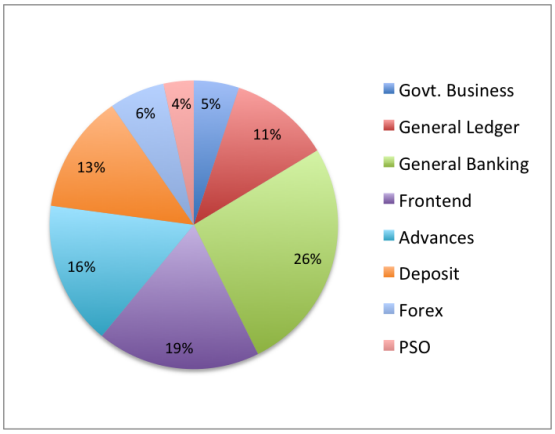

Fig. 2: Distribution of change requests

4) Identifying the *code smells* that make the program analysis imprecise, and eliminating them.
5) Identifying and marking the *paragraphs* / code for each feature / sub-feature, using program analysis and the knowledge of the parameters used in the code.
6) Analyzing the marked *paragraphs* with respect to the features / sub-features to determine which *paragraphs* can be split into separate programs.
7) Analyzing the code for detecting inconsistency of design patterns and code smells that are domain information dependent, and recommending to eliminate them.
8) Physically splitting the program into multiple programs that may belong to the domain modules or re-usable utility modules

A. Initial Study

We first examined the high-level functional overview of the software system. This helped us understand the overall functionality of the system, the transactions that it enabled, and the high-level organization of the functionality into modules. Next, to comprehend the General Ledger component, which was giving trouble to the development team, we needed information about its multiple functionality, how were the functionalities organized as modules and programs, and the dependency structure across the modules and within the programs of the General Ledger module. As a side-effect, the comprehension also helped to identify program(s) from other modules which were crucial from the dependency perspective.

We collated this information in two ways - one was by studying the functional documents and talking to the development team, second was by running program analysis tools on the module code. The latter was to validate the programs and their dependencies given by the development team to the extent possible, and to close the gaps, if any. Program analysis tools were used to generate -

- Call-graph of the General Ledger subsystem
- Interfaces with external programs / libraries / subsystems
- Cross-references of critical records and programs of General Ledger subsystem
- Inventory information of each program in General Ledger subsystem

A critical analysis of the reports and the information provided by developers enabled us to identify the code organization of the General Ledger subsystem. This information is depicted in Figure 3, and was validated by the development team.

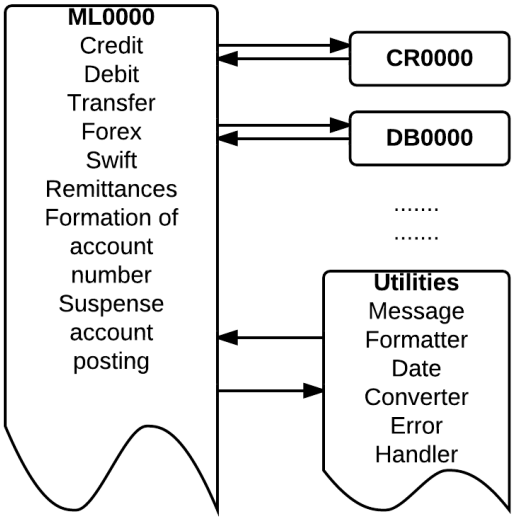

Fig. 3: Current Code Organization of the General Ledger Program

It was evident that the program ML0000 was the starting program for all the functionalites encoded in the General Ledger subsystem. This program would, in turn, pass the control to other programs in the subsystem. However, its size of about 62K lines, with more than 1000 Paragraphs was a surprise, since all other programs in the subsytem were 9K lines or smaller; indeed the average size of all the remaining programs in the subsystem was 1.8K lines. It thus became important to study the program ML0000 in detail, and identify what functionalities were encoded in the program. Though the original intent of the program ML0000, as confirmed by the development team, was to perform all the *initializations* for General Ledger subsystem, the large size of the program implied that it contained some functional logic as well. If it indeed contained functional logic of multiple functions, then the ideal criteria to split the large program would be based on the functions implemented by ML0000. Based on this criteria, we describe our proposed algorithm to split the program.

B. Code smells

While programming in COBOL, the developers are observed to make use of un-structured statements (like GO TO) that sometimes, inadvertently, introduce jumps across *perform ranges*. Such jumps usually lead to transfer of control using *fall-throughs*. The jumps not only lead to potential failures during execution [25], but also increase the number of valid but practically infeasible paths in the code, by a very large number, leading to high level of imprecision in the results when static program analysis is applied to the code. It is this latter effect that makes it important to remove such un-desirable GOTOs from the code. We apply the principles described in [25] to detect and eliminate the un-desirable GOTOs from the COBOL code, before applying static program analysis to the code.

C. Algorithm for splitting the program

1 For the program to be restructured, list down all the possible features for which program is required to be modified in order to execute a change request
2 Map the features to one or more values of the parameters that are responsible to control the features
3 Mark the code elements at desired level (e.g. statement, methods, class, etc) with the feature name(s), corresponding to the features they (code elements) refer to
4 Analyze the marked code elements with respect to the features and form programs, domain and reusable components or utilities leading to logical partitioning of the code

In the steps 1 and 2, it is essential to involve the development team working on corresponding modules and incorporate their feedback to know the features and controlling parameters. Likewise, it is important to engage the system architect while analyzing the marked code (step 4), leading to better partitions driven by the way the subsystem is anticipated to change in the future. The sections below describe the steps in detail.

D. Leveraging the domain knowledge and the parameterized design of the software

To begin with, we use a combination of static and dynamic analysis to identify the parameters in the General Ledger subsytem. Next, by analyzing the the change-request log, we identify the various domain functionalities for which the subject program was modified. Finally, using the versioning system diff mechanism, we determine the parameters that provided the handle over the domain functionality, i.e., identify the parameters that are relevant to the domain function(s) identiifed in the previous step. We interacted with the development team to validate and verify our findings.

For steps 3 and 4, the input consists of source program and list of application functions (LAF) along with the controlling parameters (for the case-study, the parameters are transaction code and attribute bits). The output consists of multiple programs, such that each program represents a function or a domain utility or a technical utility, or a collection of related functions and utilities that are used only by one collection of functions. The description given below considers COBOL programming language statements.

E. Detecting elements and relations between them

1 Determine list of Paragraphs or Paragraph-clusters (LPC) from the COBOL program. The Paragraph or Paragraph-cluster is defined as the largest unit of code that is "performed" explicitly, multiple times. Frequency of execution / invocation can be parameterized in the implementation.

2 For each application function (AF) in LAF
–For each Paragraph or Paragraph-cluster (PC) in LPC
–If PC implements AF, Then PC-AF = PC ∪ PC-AF
–If PC is invoked multiple-times from the same AF, Then PCM-AF = PC ∪ PCM-AF
PC-AF represents the list of Paragraph-clusters/Paragraphs (PC) which is invoked once from AF, and PCM-AF represents the list of PC which are invoked multiple times from AF

3 Determine the Paragraph-clusters or Paragraphs that participate across multiple (two or more) application functions (PC-LAF)

4 For each application function (AF)
–If ALL the PC that implement AF are used by another application function (AFC) implying AF ⊂ AFC
–Then, mark AF as a domain utility
–Else, CONTINUE

F. Analyzing relations

5 *Foreach* PC in LPC
–If PC belongs to PCM-AF but does not belong to PC-LAF
–Then, PC can be marked as a Technical Utility
–ElseIf PC belongs to PCM-AF and also belongs to PC-LAF
–Then, PC be marked as a Technical Utility or Domain Utility
–ElseIf PC does not belong to PCM-AF but belongs to PC-LAF
–Then, PC be marked as a Technical Utility or Domain Utility
– Else, process Next PC (CONTINUE)

6 *Foreach* AF ∈ LAF
–If AF does not belong to any PG-LAF
–Then, Create a new program (PG-LAF) that implements AF
–Else, set PG-LAF to the one containing AF

Foreach PC
–If PC belongs to some PG-LAF-PC
–Then process Next PC (CONTINUE)
–Else, Next step
–If PC belongs to PC-AF
–Then add PC to program PG-LAF thus creating relation PG-LAF-PC
– Else, Next step
–If PC belongs to another application function (PC-LAF)
–Then, add each application function from PC-LAF to PG-LAF
–Else, Next Step
–If AF contains another application function completely (has entry in the LAFC part of AF-LAFC) and AF is marked as a domain utility
–Then, replace PERFORM of AF with CALL to AF

G. Detecting Design Patterns and Domain-dependent Code Smells

With the knowledge that the source code has a parameterized design, the analysis exploited this knowledge to determine the code that implements application functions. In that process, it was observed that the conditional checks involving *parameters* were multifold:

- Conditions consisting of only transaction-code or only transaction-attribute checks
- Conditions consisting of transaction-code and transaction-attribute checks
- Conditions consisting of transaction attribute check with a check for local variables

```
114370   IF AB-INPUT-VALUE-TABLE (24) = 1   ← transaction-attribute check
114380     OR TRN-NO = 27000 OR 27010 OR 22010  ← transaction-code check
114390              OR 22011 OR 27011
114400       CALL "CL0001" USING
114420                    STARTAREA
114430       GO TO C099-EXIT.
```

Fig. 4: Transaction-code check and Transaction-attribute check

```
B00008 IF NO -TRAN-CHECK AND   ← local variable check
B00008    NO-CL-TRAN-SWEEP AND
B00008    AB-INPUT-VALUE-TABLE (20) = 0  ← transaction-attribute check
B00008    AND WA-ACCT-CHK-DONE = 0
B00008    PERFORM VARYING AB-GOCL-INDEX FROM 1 BY 1
B00008          UNTIL AB-GOCL-INDEX = 17
```

Fig. 5: Local variable check and Transaction-attribute check

Figure 4 and 5 show code samples to illustate the conditions. After detailed discussions with the development team, it was concluded that the valid use of parameters consists of conditional check for transaction attribute followed by an optional conditional check for transaction code. The use of local variables in the condiitonal check is invalid, and needs to be removed post analysis.

Another critical observation was that each and every check for transaction code was using hard-coded numeric literals. Code samples shown in Figure 6 illustrate this observation. Use of hard-coded numbers makes it very difficult for human reading and analysis of the code, though automated program analysis has less issues with this code smell.

It was recommended to the development team to replace all hard-coded constants that represent transaction code with symbols. In case of COBOL, this is possible by using variables with constant initial values.

It was also observed that the conditional checks and processing for obsolete and abondoned transaction codes is present in the source code. The source code also has invocation of empty *paragraphs*. The presence of such code is an aberration for a

```
Declaration:
026960      88  AB-CL-TRAN-SWEEP          VALUE 21047 21057.
Usage 1:
B00008 IF NOT AB-TRAN-CHECK AND
B00008     NOT AB-CL-TRAN-SWEEP AND
---------------------------------
Usage 2:
086240     IF TRN-NO NOT = 21047 AND 21057
CFX143     IF (AOSD-SYST-GEN NOT = "GEN"
CRW145       AND TRN-NO NOT = 20042
C143U7      AND AB-20042-TYPE-I NOT = '3')
C143U7       AND AB-TYPE-20042 NOT = '3')
086250     PERFORM M000-END-PARA THRU M099-EXIT.
```

Fig. 6: Hard-coded literals as transaction values

human reader, and also affects the efficiency of automated program analysis and is, therefore, recommended to be removed.

VI. Results

Figure 3 depicts the state of the program ML0000, the large General Ledger program, at a high level, before applying the splitting algorithm. The program ML0000 has sphagetti source code implementing parts of various features including Credit, Debit, Transfer, Forex, Swift, Remittances, Formation of general ledger account number, Suspense account posting, and others. This is inspite of the subsystem having a design with groups of programs, each group being responsible for a set of related features. Program DB0000 implements all the Debit related features, while program CR0000 implements all Credit related features. In addition, there are sets of programs which are re-usable utilities. Example functions are Message formatting, Date conversion,Error handling, and others. These programs are used by various features throughout the system. We denote such functions also as features, though they are not externally visible to the users of the system.

The organization of the code in program ML0000 explains why the program needs to be modified for every change request related to the General Ledger functionality. The development team potentially added new code for new features in ML0000 rather than adding the new code into the respective module, eventually converting the generic, initialization program into a monolithic piece of code implementing multiple domain functions.

Figure 7 describes the proposed code organization to enable easier and quicker evolution.

Due to space constraints, we discuss the analysis of only the *credit* feature, which is one of the significant features of General Ledger module. Credit has various sub-features like credit by cash and credit by batch. The feature is controlled by two parameters - Transaction code (TRN-NO) and Transaction attributes (EDIP bits). There are 16 different values for transaction codes, and a single ED1P bit which controls the two sub-features - Credit by Cash and Credit by Batch. This information was produced by the application of step 1 and step 2 of the program splitting algorithm described in the methodology section.

The marking of the code elements described in step 3 is largely automated. Few limitations in the automation are due to imprecision in static program analysis and lack of complete

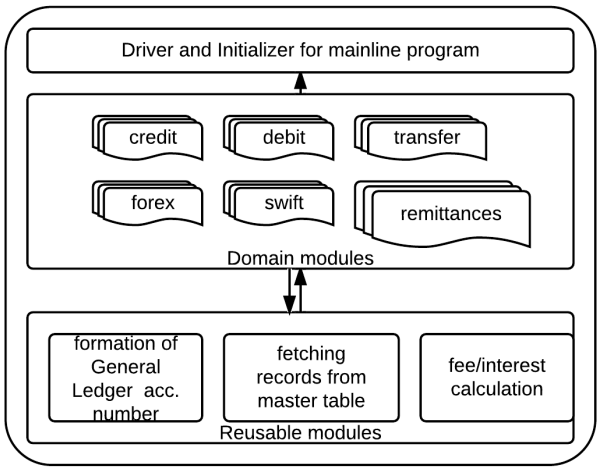

Fig. 7: Proposed Code Organization of the General Ledger Program

support for all the COBOL statements. Though the tools work on a common intermediate representation, they were originally designed for another programming language. The COBOL language has its specificities of Paragraphs, Sections, Performs, Evaluates and other statements with entirely unique semantics, the automation for which is in progress. Complete automation of this step is, however, feasible.

Applying step 4 for detecting elements and analysing the relations between them, gives us the two configurations for the code arrangement of Credit by Cash and Credit by Batch sub-features depicted below in Figure 8 and Figure 9. They describe the structural arrangment of the code, whereas Table X and Table XI show the statistics with respect to lines of code and number of Paragraph metrics.

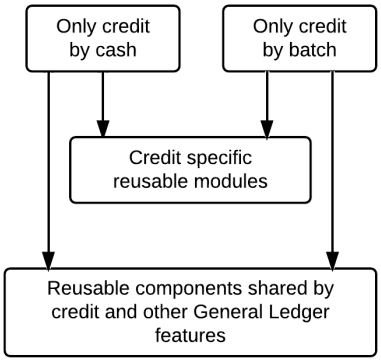

Fig. 8: Config. 1: Code organization of the credit by cash and credit by batch

Functionality	number of paras	Total lines of code
Only credit by cash	2	191
Only credit by batch	1	82
Credit specific reusable modules	11	82
Reusable modules (credit + other features)	104	13970

TABLE X: Statistics for code organization 1

In the first arrangement (Figure 8), we propose a layered

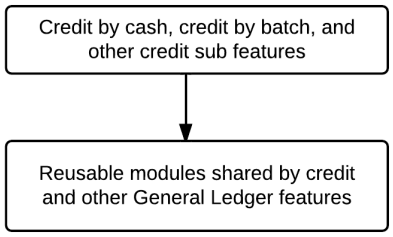

Fig. 9: Config. 2: Code organization of the credit by cash and credit by batch

architecture with three layers, where the utilities used by multiple features of the General Ledger functionality are pushed to the lowest level. The application architecture also proposes a layer consisting of Credit specific reusable features. Finally, the top layer consists of the sub-features of Credit, in this case Credit by Cash and Credit by Batch

Functionality	number of paras	Total lines of code
Credit by cash, credit by batch, other credit features	14	973
reusable modules	104	13970

TABLE XI: Statistics for code organization 2

The second arrangement (Figure 9) differs from the first one in the number of layers. In this arrangement, the Credit specific reusable features and the utility features shared by Credit module and other General Ledger functions are merged into a single layer. In addition, at the top layer, there is no separation between Credit by Cash, Credit by Batch and other Credit sub-features and they are merged together as Credit features.

The choice of the code organization depends on the size of the Paragraph-cluster representing the functionality, and the change history of the feature in question. In the case-study, for code arrangement 1, as shown in table X, the size of Paragrapah-clusters for sub-features pertaining to Credit by Cash and Credit by Batch is 2 and 1 respectively. Alternatively, by combining the para-clusters for both these features we have the resultant cluster of 14 paras. We recommend second approach of combining para-clusters to have a final cluster of 14 paras, which is still cohesive and easy to evolve. However, if the feature has undergone a lot of changes in the past and is anticipated to have frequent changes, then inspite of the small size of Paragraph-cluster, it is advisable to choose the first arrangement. After adopting the proposed arrangement and physically splitting the programs, any change request related to Credit by Cash and Credit by Batch features, will be carried out locally on the newly formed programs for these features and will have little or no impact on the General Ledger module.

Repeating a similar exercise for all the application functions encoded in the General Ledger program enables the parallel development of mulitple change requests in the distinct sub-features of General Ledger.

VII. Threats to validity

The effectiveness of the model relies heavily on the quality of the data collected. If the quality is not ensured, the inferences and activities could be futile or sometimes might even have negative impact. However, there are guidelines which can reduce the impurity in the data collection. First of all, it is important to be aware of the danger of making inferences without attempting to understand environmental factors that might affect the data. As an example, we noticed a fairly small percentage of user interface issues in the case study. This was surprising, since user interfaces are most common error sources [21]. After interacting with the desginers, it was clear from the discussion that the team had already anticipated changes and designed the user-interface so that the code is more amenable to future changes. Thus, we did not misinterpret that user interfaces were not a significant problem, but got a better understanding of team capabilities. Second, the Hawthrone effect [15] cannot be completely neglected. When project personnel become aware that an aspect of their behavior is being monitored (in the form of data gathered), their behavior changes. Also, it is important to convince the project team about the purpose of the entire process, as their are associated human factors. Treating this activity as a normal phase of software development, which in the long-term may simplify their tasks, and not as a lever for evaluation of the practitioner's performance is important. Lack of automation of the data collection and analysis process, or considering this as an offline process may lead to erroneous and incosistent data, hampering further analysis. Integrating data collection and validation procedures into the version control system and automating as much data collection as possible is the key to avoid such issues. For measuring data-based improvements after applying the methodology, it is important to ensure that the resources available before and after the implementation of technique are comparable. The resources could be environmental or human. It is important to be aware of the continually changing goals and interest of projects for which this activity is performed. Generalizing analysis without considering these inputs might lead to inflated results. For instance, a fault may persist in the system for a long time. But it does not mean that it took that long for fixing the fault. The reason could be that there were other priorities of importance during that period. While these are some measures to keep check of data quality, the proposed restructuring techniques have their roots in the quality of the domain knowledge supplied to them. Thus, working closely with development team helps to increase the quality of the resultant functional components.

VIII. Conclusion and Future Work

In this paper, we have proposed a model for analysis of Change Request and Failure data. Inferences drawn from the model can help us to identify the improvement phases of evolution and also suggest corrective measures for the identified code. The corrective measures can initiate reshaping and reorganizing the identified (troublesome) code to attain characteristics, which will enable it to change quickly with fewer efforts to adapt to the changing environment and requirements. This can help the organization save huge costs to transform or re-develop the system. We demonstrated the use of a model to figure out hotspots that disturb the evolution, in a large, complex software from financial domain. Further, we presented our experiences of restructuring such hotspots in the system, based on its functional architecture. We have contributed a technique which describes the use of code

characteristics and domain knowledge to achieve meaningful functional partitions of large monilithic piece of code. We have demonstrated how this technique created the two code organizations for *Credit* related features in the *General Ledger* subsystem.

Since there is no gold standard for evaluation, for quantitative validation, we aim to collect the change request data (described in above sections) in terms of number of change requests executed in parallel, and the efforts and time spent after the code restructuring. The data so gathered will be compared with similar information about the *General Ledger* subsystem before code restructuring. Qualitative feedback will be in the form of informal discussions with the development team about how difficult or easy it is for them to make changes to the restructured code.

The proposed technique will work for enterprise systems or products developed using explicit *parameters*. If the application architecture of such systems does not use parameterization, then we will need to evolve a variation of the technique to analyze the non-parameterized systems. However, our initial study of a large enterprise inventory management system (written using RPG language), and another engineering system (printer controller software written using C/C++) establishes that parameterized application architecture is usually employed, though not completely adhered to.

While analyzing the code, the assumption is that a single *para / code unit* implements a single functionality. Our observation is that this assumption is not always correct, and triggers the need of intra-code-unit program analysis in the long run.

We are in the process of conceptualizing and enhancing the automation to accept as input the specifications describing the domain information and thus automate the complete restrucuting process using DSM techniques [13].

ACKNOWLEDGMENT

The authors would like to thank the development team maintaining the financial system (case-study) for their valuable inputs and feedback during the entire course of this work.

REFERENCES

[1] M. M. Lehman, J. F. Ramil, P. D. Wernick, and D. E. Perry, *Metrics and Laws of Software Evolution - The Nineties View*, Proceedings of the 4th Software Metrics Symposium, pp 20-32, 1997.

[2] D. L. Parnas, *Software Aging*, Invited Plenary Talk, Proceedings of The 16th International Conference on Software Engineering, IEEE Press, 1994.

[3] James M. Neighbors, *Finding Reusable Components in Large Systems*, WCRE.

[4] N. Anquetil, Timothy Lethbridge, *Experiments with Clustering as a Software Remodularization Method*, WCRE.

[5] Lung, C., Zaman, M., Nandi, A.: *Applications of clustering techniques to software partitioning, recovery and restructuring*. Journal of Systems and Software (2004), pp 227-244.

[6] A. Marcus, A. Sergeyev, V. Rajlich, J. I. Maletic, *An information retrieval approach to concept location in source code*, WCRE, pp 214-223, IEEE Computer Society, 2004.

[7] Martin Fowler, *Refactoring: Improving the design of existing code*, Addison Wesley, 1999.

[8] A. van Deursen, and T. Kuipers, *Identifying objects using cluster and concept analysis*, Proceedings of International Conference on Software Engineering, pp 246-255, 1999.

[9] D. H. Hutchens, and V. R. Basili, *System structure analysis: clustering with data bindings*, IEEE transactions on Software Engineering, 11(8):749-757,1985.

[10] S. Mancoridis, B Mitchell, Y. Chen, and E. Ganser, *Bunch: A clustering tool for recovery and maintenance of software system structures*,Proceedings of International Conference on Software Maintenance, pp 50-59, 1999.

[11] S. Mancoridis, B Mitchell, C. Rorres, and Y. Chen, *Using automatic clustering to produce high-level system organizations of source code*,Proceedings of International Workshop on Program Comprehension, pp 45-52, 1998.

[12] Marco Glorie, Andy Zaidman, Arie van Deursen, and Lennart Hofland , *Splitting a Large Software Archive for Easing Future Software Evolution - An Industrial Eperience Report using Formal Concept Analysis*,Journal of Software Maintenance and Evolution: Research and Practice, pp 113-141, 2009.

[13] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson, *Using Dependency Models to Manage Complex Software Architecture*,Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp 167-176, 2005

[14] Hitesh Sajnani, Ravindra Naik, and Cristina Lopes, *Application Architecture Discovery - Towards Domain-Driven, Easily-Extensible Code Structure*, Working Conference on Reverse Engineering, Limerick Ireland, October 2011

[15] J. Brown, The Social Psychology of Industry. Baltimore, MD: Penguin, 1954

[16] Ravindra Naik, and Hitesh Sajnani, *Using Change History of Software To improve Evolvability*, Proceedings of India Software Engineering Conference, 2009

[17] Anvik, J., Hiew, L., and Murphy, G. C., *Who Should Fix This Bug?*, Proceedings of 28th international conference on Software engineering, Shanghai, China, 2006, pp. 361 - 370.

[18] Canfora, G. and Cerulo, L., *Impact Analysis by Mining Software and Change Request Repositories*, Proceedings of 11th IEEE International Symposium on Software Metrics, September 19-22 2005, pp. 20-29.

[19] Canfora, and G. Cerulo, L., *Fine Grained Indexing of Software Repositories to Support Impact Analysis*, Proceedings of International Workshop on Mining Software Repositories, 2006, pp. 105 - 111.

[20] Chen, K. and Rajlich, V., *Case Study of Feature Location Using Dependence Graph*, Proceedings of 8th IEEE International Workshop on Program Comprehension, Limerick, Ireland, June 2000, pp. 241-249.

[21] Thomas Zimmermann, Peter Weigerber, Stephan Diehl, and Andreas Zeller, *Mining Version Histories to Guide Software Changes*, In IEEE Transactions on Software Engineering (31): 429-445 (2005), June 2005, pp. 429-445.

[22] Silvia Breu, and Thomas Zimmermann, *Mining Aspects from Version History*, Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), Tokyo, Japan, September 2006, pp. 221-230.

[23] Huzefa H. Kagdi and Denys Poshyvanyk, *Who can help me with this change request?*, Proceeding of International Workshop on Program Comprehension, 2009, pp. 273-277

[24] Sunghun Kim, E. James Whitehead, and Yi Zhang, *Classifying Software Changes: Clean or Buggy?* ,IEEE Transaction on Software Engg., vol. 34, No.2, 2008

[25] Anushri Agrawal and Ravindra Naik, *Towards Assuring Non-recurrence of Faults Leading to Transaction Outages - an Experiment with Stable Business Applications*, Proceedings of the 4th India Software Engineering Conference, 2011, pp. 107-110