Application Architecture Discovery - Towards Domain-driven, Easily-Extensible Code Structure

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—The architecture of a software system and its code structure have a strong impact on its maintainability – the ability to fix problems, and make changes to the system efficiently. To ensure maintainability, software systems are usually organized as subsystems or modules, each with atomically defined responsibilities. However, as the system evolves, the structure of the system undergoes continuous modifications, drifting away from its original design, leading to functionally non-atomic modules and intertwined dependencies between the modules.

In this paper, we propose an approach to improve the code structure and architecture by leveraging the domain knowledge of the system. Our approach exploits the knowledge about the functional architecture of the system to restructure the source code and align physically with the functional elements and the reusable library layers. The approach is validated by applying to a case study which is an existing financial system. The preliminary analysis for the case-study reveals that the approach creates meaningful structure from the legacy 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]. A Retail Banking Software System maintained by Tata Consultancy Services in active use by a nationalized bank is in a similar evolution state. 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 incorporate new, modern technologies, 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]. The Product Manager of the banking system, based on analysis of the process data and his experience, 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 principal research question for our study is: how to leverage domain knowledge for re-structuring a large-scale program?

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

II. BACKGROUND

Restructuring of software systems is an important problem and there is substantial literature on that topic. Glorie et. al [14] 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. Hutchens and Basili [10] identify potential modules by clustering that is based on data-bindings between procedures. Schwanke [13] also identifies potential modules using call dependencies. 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. Marcus et. al [6] use identifier names and internal comments in the code to create a corpus of source artefacts. Using Latent Semantic Indexing, the user can fire queries to identify concepts in the source code. In the context of COBOL, Van Deursen and Kuipers [9] 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 a very generic assumption and inference.

When considering related work, we found that the techniques that enable users to restructure [4,5,6] and refactor
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.

We propose an approach to segregate the large program
based on the domain functions. This results in multiple pro-
grams, each of which performs distinct domain functions,
or denotes common library (utility) functions. The resulting
programs are smaller and functionally self-contained. The
major contribution of this paper is the description of our
experience of applying our technique in an industrial setting
on a large scale legacy application.

III. 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.

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 ex-
cisting code to develop feature variants. The parameters primar-
ily represent the transaction codes (examples are Cash Posting,
Cheque Deposit, Cash Distribution) and transaction attributes
(examples are Correction, Backdating, Deposit, Withdrawal).
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 parameterized design.

B. The Problem

While the overall system is extremely large, running into
few million lines of code that span across few thousand
COBOL programs, the focus was 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 pro-
grams?
- 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?

IV. 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) Identifying the subset of features and sub-features ap-
licable to the program(s) under analysis.
3) Identifying the parameters (if any) used in the code to
implement the features and sub-features. The conjec-
ture is that any product implementation makes use of
parameters to design and implement features and their
variations.
4) Identifying and marking the paragraphs / code for each
feature / sub-feature, using program analysis and the
knowledge of the parameters used in the code.
5) Analyzing the marked paragraphs with respect to the
features / sub-features to determine which paragraphs
can be split into separate programs.
6) 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 pro-
grams 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 devel-
opment 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 Gen-
eral 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 1, and was validated by the development team.

It was evident that the program ML0000 was the starting
program for all the functionalities 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 subsystem 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. 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.

C. 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 subsystem. 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) identified 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.

D. 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
E. Analyzing relations

5. For each 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 \textbf{Foreach} \( AF \in \text{LAF} \)

- If \( AF \) does not belong to any \( \text{PG-LAF} \)
  - Then, Create a new program (PG-LAF) that implements \( AF \)
  - Else, set \( \text{PG-LAF} \) to the one containing \( AF \)

\textbf{Foreach} \( PC \)

- If \( PC \) belongs to some \( \text{PG-LAF-PC} \)
  - Then process Next PC (CONTINUE)
  - Else, Next step

- If \( PC \) belongs to \( \text{PC-AF} \)
  - Then add \( PC \) to program \( \text{PG-LAF} \) thus creating relation \( \text{PG-LAF-PC} \)
  - Else, Next step

- If \( PC \) belongs to another application function \( \text{PC-LAF} \)
  - Then, add each application function from \( \text{PC-LAF} \) to \( \text{PG-LAF} \)
  - Else, Next Step

- If \( AF \) contains another application function completely (has entry in the \( \text{LAFC} \) part of \( \text{AF-LAFC} \)) and \( AF \) is marked as a domain utility
  - Then, replace \text{PERFORM} of \( AF \) with \text{CALL} to \( AF \)

\textbf{V. RESULTS}

Figure 1 depicts the state of the program \( \text{ML0000} \), the large General Ledger program, at a high level, before applying the splitting algorithm. The program \( \text{ML0000} \) has spaghetti 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 in spite of the subsystem having a design with groups of programs, each group being responsible for a set of related features. Program \( \text{DB0000} \) implements all the Debit related features, while program \( \text{CR0000} \) implements all Credit related features. In addition, there are sets of programs which are reusable 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 \( \text{ML0000} \) 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 \( \text{ML0000} \) 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 2 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 EDIP 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 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 3 and Figure ???. They describe the structural arrangement of the code, whereas Table I and Table II show the statistics with respect to lines of code and number of Paragraph metrics.

In the first arrangement (Figure 3), we propose a layered 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.

The second arrangement differs from the first one in the number of layers. In this arrangement, the Credit specific
have demonstrated how this technique created the two code
modules, because of its functional architecture. We
structuring a parameterized, large, complex software from

about the
data so gathered would be compared with similar information
and the efforts and time spent after the code restructuring. The
in terms of number of change requests executed in parallel,
tative validation, we aim to collect the change request logs
subsystem.
features of General Ledger.
module.
features and will have little or no impact on the General Ledger
be carried out locally on the newly formed programs for these
related to Credit by Cash and Credit by Batch features, will
first arrangement. After adopting the proposed arrangement
small size of Paragraph-cluster, it is advisable to choose the
is anticipated to have frequent changes, then inspite of the
if the feature has undergone a lot of changes in the past and
or products developed using explicit parameters. If the ap-
lication architecture of such systems does not use parameter-
ization, 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++) es-
tablishes that parameterized application architecture is usually
employed, though not completely adhered to.
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 re-
structuring process using DSM techniques[15].

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
[3] James M. Neighbors, Finding Reusable Components in Large Systems,
WCRC.
[4] N. Anquetil, Timothy Lethbridge, Experiments with Clustering as a
Software Remodularization Method, WCRC.
to software partitioning, recovery and restructuring. Journal of Systems
and Software (2004), pp 227-244.
approach to concept location in source code, WCRC, pp 214-223, IEEE
[7] Martin Fowler, Refactoring: Improving the design of existing code,
Addison Wesley, 1999.
and a case study, ICSM, pp357-366 2004. (To be dropped)
[9] A. van Deursen, and T. Kuipers, Identifying objects using cluster and
concept analysis, Proceedings of International Conference on Software
[10] D. H. Hutchens, and V. R. Basil, System structure analysis: clus-
tering with data bindings, IEEE transactions on Software Engineering,
tering tool for recovery and maintenance of software system struc-
tures,Proceedings of International Conference on Software Maintenance,
matic clustering to produce high-level system organizations of source
code,Proceedings of International Workshop on Program Comprehension,
modularity, Proceedings of the International Conference on Software
Engineering, pp 83-92, 1991
Splitting a Large Software Archive for Easing Future Software Evolution
of Software Maintenance and Evolution: Research and Practice, pp 113-
141, 2009.
pendency 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

<table>
<thead>
<tr>
<th>Functionality</th>
<th>number of paras</th>
<th>Total lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only credit by cash</td>
<td>2</td>
<td>191</td>
</tr>
<tr>
<td>Only credit by batch</td>
<td>1</td>
<td>82</td>
</tr>
<tr>
<td>Credit specific reusable modules</td>
<td>11</td>
<td>82</td>
</tr>
<tr>
<td>Reusable modules ( credit + other features)</td>
<td>104</td>
<td>13970</td>
</tr>
</tbody>
</table>

TABLE I
STATISTICS FOR CODE ORGANIZATION 1

<table>
<thead>
<tr>
<th>Functionality</th>
<th>number of paras</th>
<th>Total lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Credit by cash, credit by batch, other credit features</td>
<td>14</td>
<td>973</td>
</tr>
<tr>
<td>Reusable modules</td>
<td>104</td>
<td>13970</td>
</tr>
</tbody>
</table>

TABLE II
STATISTICS FOR CODE ORGANIZATION 2

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 I, the size of Paragraph-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.

VI. CONCLUSION AND FUTURE WORK
In this paper, we have presented our experiences of re-
structuring a parameterized, large, complex software from
financial domain, 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 monolithic 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 quanti-
tative validation, we aim to collect the change request logs
in terms of number of change requests executed in parallel,
and the efforts and time spent after the code restructuring.
The data so gathered would be compared with similar information
about the General Ledger subsystem before code restructuring.
Qualitative feedback would 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 ap-
plication architecture of such systems does not use param-
eterization, 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++) es-
tablishes that parameterized application architecture is usually
employed, though not completely adhered to.
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 re-
structuring process using DSM techniques[15].

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.

<table>
<thead>
<tr>
<th>Functionality</th>
<th>number of paras</th>
<th>Total lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only credit by cash</td>
<td>2</td>
<td>191</td>
</tr>
<tr>
<td>Only credit by batch</td>
<td>1</td>
<td>82</td>
</tr>
<tr>
<td>Credit specific reusable modules</td>
<td>11</td>
<td>82</td>
</tr>
<tr>
<td>Reusable modules ( credit + other features)</td>
<td>104</td>
<td>13970</td>
</tr>
</tbody>
</table>