Refactoring for Changeability: A way to go?

Birgit Geppert, Audris Mockus, and Frank Rößler
Avaya Labs
Software Technology Research
{bgeppert, audris, roessler}@research.avayalabs.com

Abstract

Legacy systems are difficult and expensive to maintain due to size, complexity, and age of their code base. Business needs require continuously adding new features and maintaining older releases. This and the ever present worry about feature breakage are often the reason why the sweeping changes for reversing design degradation are considered too costly, risky and difficult to implement. We study a refactoring carried out on a part of a large legacy business communication product where protocol logic in the registration domain was restructured under ongoing development of a parallel branch of the legacy code. We pose a number of hypotheses about the strategies and effects of the refactoring effort on aspects of changeability and measure the outcomes. The results of this case study show a significant decrease in customer reported defects and in effort needed to make changes.

1. Introduction

Legacy systems are difficult and expensive to maintain due to size, complexity, and age of their code base. The original creators sometimes leave the organization, leaving the knowledge gap and the legacy of a hard to understand code base. Improving such code base, thus, poses enormous challenges. Complicating things, new features are being added, which requires supporting the old and new code over long periods of time. As a consequence, making the necessary sweeping changes is often considered too risky and difficult to implement.

In an effort to overcome the reluctance of restructuring complex legacy code, we have demonstrated a restructuring approach on the registration software of a large legacy business communication product. The restructured code is now available to customers. Registration implements a standard-based communication protocol with several proprietary extensions. We have applied the SELEX architectural pattern (Section 2.3) for defining the target design and software refactoring [4] as an approach for migrating the original code towards the new design. In order to avoid disruption of ongoing development, we performed the refactoring on a parallel branch of the code base and synchronized the two branches later on.

The business purpose of this refactoring effort was to improve intelligibility, changeability, and quality of the code, but without imposing a significant performance penalty and without breaking or changing existing functionality. In [7], we reported on our measurement results at a stage where the refactored code had just entered system testing, i.e., before alpha and beta testing and system deployment. We found that an expected performance degradation did not occur. On the contrary, we measured a performance improvement of around 10%, which was achieved simply as a byproduct of the structural changes and not by an intentional optimization effort. Furthermore, our unit testing results strongly suggested that software refactoring indeed enabled us to migrate the code with little risk of feature breakage. Nevertheless, those findings did not yet include results from system verification and potential problems from the field, because they were not available at the time. Also, concerning changeability, we could only report on our experiences with those changes submitted in parallel to our refactoring project. Now, one year later with the product being sold to customers, we are in a better position for a more thorough analysis.

To systematically study the impact of our refactoring on changeability of the code, we need to pose a number of hypotheses and quantify our findings. For that we analyze change data that was collected since the system was deployed (available to customers since mid 2004).

The remainder of the paper is organized as follows. We start from describing the project context in Section 2, including the background of the product, the nature of the refactored domain, a brief introduction to our re-engineering approach, and a discussion of our hypotheses. We follow with quantitative methodology in Section 3 and present the results in Section 4. Finally, we discuss validation in Section 5 and conclusion in Section 6.

2. Project Context

2.1 Product

We examine the call processing software installed on many Avaya telephony systems. This software system is an established product and embodies several decades of knowledge and experience in the telephony field. In a recent release, the software contains approximately seven million lines of code mostly in C and C++. The software development organization deploys major releases on a fixed schedule, with subsequent dot releases that bundle patches and refinements to the system.

Multiple releases are in the field and are used by tens of thousands of customers, many of whose businesses depend on the high availability of the product. This makes the software exceedingly difficult to enhance while maintaining the smooth operation of the hardware/software combinations deployed.

2.2 Registration Domain

The refactored target domain implements the protocol logic for registering certain types of IP phones and backup IP telephony servers. The subsystem consists of a standard-based communication protocol with numerous proprietary extensions. It has around 30 KLOC not including code generated from ASN.1 specifications and a 3rd party protocol stack for lower-level packet transmission functionality.

The registration code has been developed in C++ over the past 5 years with contributions from around 40 different developers. During that time the code had undergone numerous enhancements, and many more enhancements are expected. With each enhancement we take a chance of design degradation, a common and well-known problem in software engineering.

The original design of the target subsystem had four separate modules dealing with following concerns:

- packet transmission,
- error-control,
- message en/decoding, and
- message processing.

Message processing was further structured according to the different messages that are sent over the net and following concerns have been separated out in different member functions:

- dispatch incoming messages,
- extract parameters of an incoming message and trigger message processing,
- process an incoming message,
- trigger necessary steps for sending an outgoing message,
- trigger underlying protocol state machine, populate message structure, and trigger encoding.

It should be obvious that the original design encapsulated different steps in processing of an incoming message. While being a common approach to software design, “steps in processing” did not adequately support desirable quality attributes such as changeability and intelligibility. The goal of the refactoring effort was to set the focus more on these attributes.

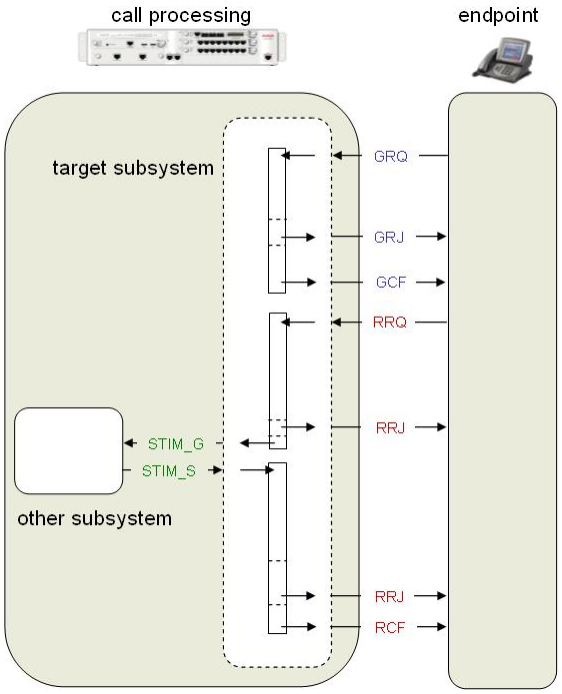

Figure 1: Registration Protocol : Excerpt from original design

2.3 Re-Engineering Approach

SELEX Architecture Pattern

Communication protocols often have highly optimized designs, and therefore tend to be tightly coupled, difficult to change and hard to understand. For instance, the registration protocol encompasses many protocol operations such as detecting network address translation (NATDiscovery), authenticating IP phones (Authentication), binding an IP phone's address to its extension (AssociateStation), and checking that a phone is correctly administered on the switch (AdminValidation). Protocol operations are carried out while exchanging a minimum set of messages. The target subsystem communicates with other subsystems and with the endpoint that wants to register. Figure 1 illustrates part of the message exchange among these components. Each of the messages carries information for several protocol operations, in particular, the four protocol operations mentioned above. For instance, an incoming GRQ message contains both, NATDiscovery and Authentication information. Figure 2 lists the mapping of protocol operations to messages.

	GRQ	GRJ	GCF	RRQ	RRJ	RCF	STIM_G	STIM_S
NATDiscovery	X	X	X					
Authentication	X	X	X	X	X	X		
AssociateStation				X	X	X	X	X
AdminValidation				X	X	X		

Figure 2: Message Mapping

According to the original design (Section 2.2), each incoming message (GRQ, RRQ, and STIM_S) has one member function responsible for processing the message (Figure 1). Thus, we have an n:m mapping between protocol operations and member functions in the code. For instance, the Authentication functionality is scattered across the GRQ and RRQ member functions and the RRQ member function participates in both Authentication, AssociateStation, and AdminValidation. In order to modify, add, or delete a specific protocol operation we therefore need to analyze and change multiple member functions that deal with many unrelated concerns. This is a sign of poor information hiding [14]. We would rather like to encapsulate protocol operation-specific code in separate modules so that changes are confined to the corresponding modules only. This is the intent of the SELEX (SEquence & multipLEX) pattern, which provides a general way for modularizing complex communication protocols without imposing a significant performance penalty.

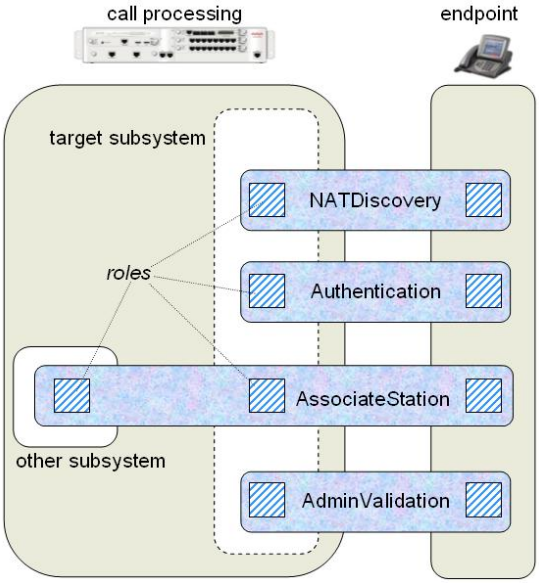

Figure 3: Registration Protocol: Collaboration-based View

In SELEX terminology, we call the encapsulation of a protocol operation a *collaboration* [5]. Note that a collaboration encapsulates the behavior of a protocol operation not only for one agent, but all agents involved in accomplishing that function. Figure 3 illustrates the collaboration-based view for our example. We have three agents, namely the endpoint, the target subsystem, and another subsystem. The target subsystem together with the endpoint implements the NATDiscovery, Authentication, and AdminValidation collaborations. For AssociateStation the target subsystem also needs to collaborate with the other subsystem. Due to their distributed nature, collaborations consist of several *roles* which are played by the involved agents. For instance, NATDiscovery has two roles, one played by the target subsystem, the other one played by the endpoint. A collaboration includes a *micro-protocol* that defines the interactions among its roles. The micro-protocol defines how collaboration roles exchange *micro-messages* in order to accomplish their common task. Each collaboration role processes one or more such micro-messages. Thus, a collaboration role consists of a number of message handlers each responsible for processing one micro-message.

Collaborations are the architectural components in SELEX. There are also two compositional mechanisms for connecting collaborations. Composition has two aspects to it. First, collaborations do not execute in isolation. Their roles may depend on each other in the sense that one role provides data and conditions that

other roles rely on. As a consequence, composition must enforce a certain execution order among message handlers, which is called *protocol sequencing*. Second, even though micro-protocols define their own message sequences, it is not always effective for the composite protocol to transmit these messages separately over the net. As a consequence, composition must be able to simultaneously execute various micro-protocols over the same message sequence, which is called *protocol multiplexing*. The elements that implement the compositional logic are called *protocol demultiplexers*..

The SELEX pattern structures a complex communication protocol into simpler micro-protocols. Looking at it from a different angle, micro-protocols are the building blocks from which more complex protocols can be composed. One advantage of the SELEX pattern is that compositional logic can be defined separately from the building blocks. In particular, this allows assembling a family of protocols from a collection of micro-protocols [6]. The family members vary in the micro-protocols selected and their specific composition. Thus, for a family of protocols we also have a family of protocol demultiplexers encapsulating the compositional logic for single family members.

For the registration domain we identified a pool of around 20 collaborations and used them for assembling a family of around 15 different registration protocols. Figure 4 compares part of the old design with the corresponding part of the new design. On the left hand side we have the original target subsystem with three large message handlers. In the old design the family members' compositional logic was mixed with the logic for multiple protocol operations and also distributed over multiple large message handlers. On the right hand side we have the new SELEX design, which cleanly separates protocol operations and compositional logic. We have four different collaboration roles, one for each protocol operation, with altogether six small message handlers. The fact that we assemble not only one protocol but rather a family of around 15 protocols is illustrated by the set of protocol demultiplexers and classification functions for identifying the right protocol demultiplexer from an incoming message.

The SELEX pattern allows designing, developing, and changing collaborations independently from each other. We can also reason separately about

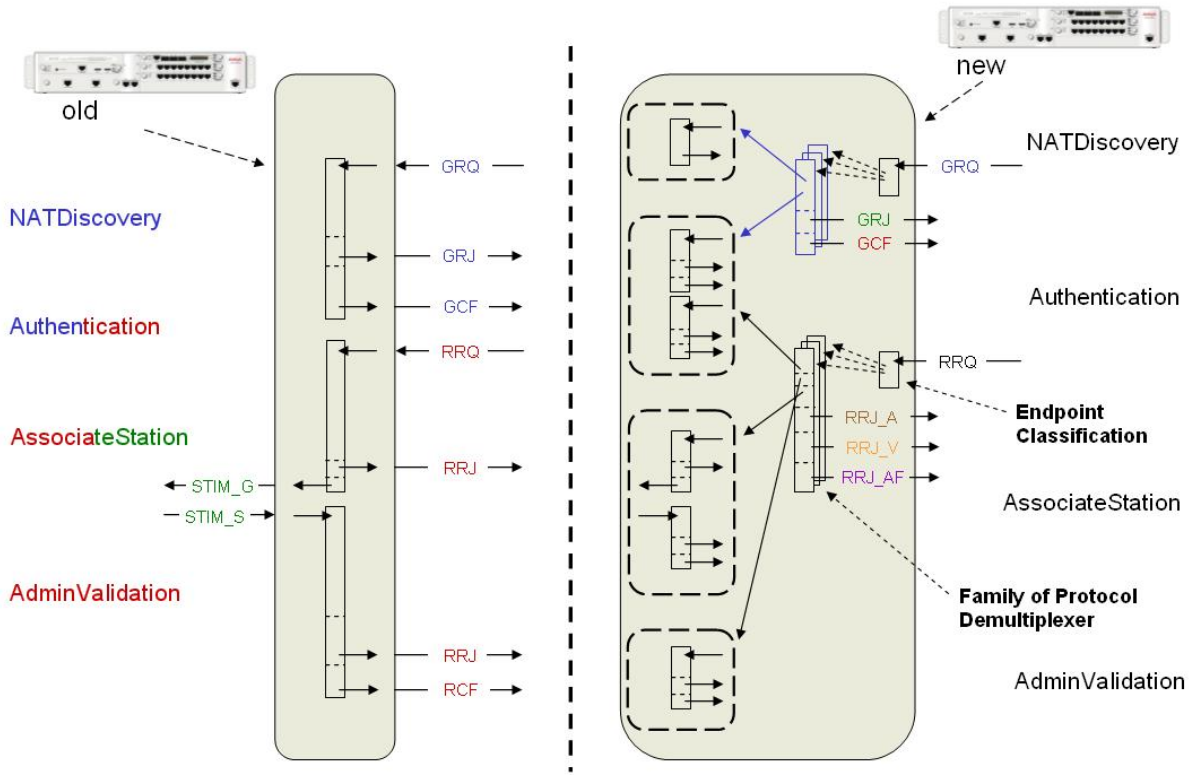

Figure 4: Registration Protocol: Old Design vs. New Design (Excerpts)

collaborations, their composition, and eventually the entire protocol and protocol family. This supports an incremental development and testing of complex (families of) communication protocols.

Software Refactoring

We applied software refactoring for migrating the original registration code towards the new SELEX design. Software refactoring [4] is the process of improving the code structure of an existing software system without changing its external behavior. It is conducted in a sequence of simple, fail-safe code transformation steps (called refactorings). If strictly applied, software refactoring allows to clean up existing code and to turn bad code into well-designed code with only little risk of introducing bugs.

There are many refactorings described in the literature. A common example is the JAVA refactoring "Extract Method" [4]. It explains how to pull a code fragment out of a larger method. The mechanics of a refactoring explain in all detail how to perform the refactoring. For instance, the mechanics of "Extract Method" start with:

1. "Create a new method, and name it after the intention of the method."
2. "Copy the extracted code from the source method into the new target method."
3. "Scan the extracted code for references to any variables that are local in scope to the source method. These are local variables and parameters to the method."

In our refactoring project, we applied several non-standard refactorings which are not published yet. For those cases, we needed to prove to ourselves that the code transformations are behavior-preserving. If we could not find a convincing argument, we did not perform a code transformation.

2.4 Hypotheses

To study the impact of our refactoring on changeability of the code, we pose a number of hypotheses about the effects and strategies of the refactoring effort and quantify our findings.

H 1. The quality of the software in terms of customer reported defect rate is likely to improve because the software is more transparent to maintain and there are fewer issues remaining as refactoring is likely to discover existing latent issues. We consider the number of field problems found and the root cause of these problems to test this hypothesis.

H 2. The refactoring reduces the effort required to make changes because the software is more transparent to maintain. We estimate change effort and we assess the amount of code that may need to be considered to make the change.

H 3. The refactoring reduces the scope of changes within the refactored domain but does not change the scope of changes outside the domain. We consider number of files touched in a change, number of lines added, and number of lines in the files that are modified as various measures of scope.

3. Methodology

We mainly use measures and factors captured in two databases containing information captured during product development including change requests and code changes and a customer report database which we use to identify software changes done as a result of customer reported problems.

3.1 The Software Change Process

When a change to the software system is needed, a work item is created. Work items range in size from very large work items, such as releases, to very small changes, such as a single delta (modification) to a file. Figure 5 shows how a typical change management system works. In the discussion below we use the word "change" to refer to changes at the granularity of an MR unless otherwise noted.

The project we consider in this paper employs a version control system (VCS) which maintains versions of the source code and documentation, and a change request management system (CMS) system that keeps track of individual requests for changes, (known as modification requests, or MRs). Whereas a delta is intended to keep track of lines of code that are changed, an MR is intended to be a change made for a single purpose. Each MR may have many deltas associated with it. The project under consideration used the Sablime system for change tracking and an internally developed system for most of the version control. It is possible to trace all software modification to an MR. The modifications are typically made for one of the following reasons.

1. Repairing previous changes that caused a failure during testing or in the field.
2. Introducing new features to the existing system.

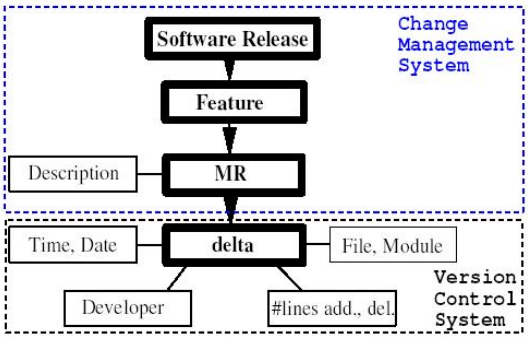

Figure 5: Hierarchy of changes and associated data sources. Boxes with dashed lines define data sources (VCS and CMS), boxes with thick lines define changes, and boxes with thin lines define properties of changes. The arrows define an "is a part of" relationship among changes, e.g., each MR is a part of a feature.

3. Restructuring the code to make it easier to understand and maintain. (An activity more common in heavily modified code, such as in legacy systems.)

MRs are assigned, among other things, the MR type, a priority, release, resolver, and resolution status, which allows us to track and monitor changes to the software. We used these systems to obtain the changes in response to failures detected after the software was deployed. We used change comment to identify a few administrative changes that were done to deploy a new lint-type software tool.

We obtained a list of all changes related to registration domain over the last six major releases of software, the directories and files where registration related code was kept and identified a set of changes that were executed after the registration refactoring was complete. We also identified a subset of the files in the registration domain that were directly involved in refactoring.

Finally, we have identified changes that were done to fix a problem raised by a customer and the release of software the customer was running at the time they encountered the problem.

3.2 The Value of Analyzing Changes

The analysis of software changes has a number of distinct benefits that may not be immediately obvious.

- The data collection is nonintrusive, using only existing data and making analysis possible in commercial projects that are usually under intense schedule pressure and do not have time or resources to collect additional data.
- Long history on past projects is available, enabling comparison to what happened in the past and customization and calibration of the methods to the existing environment. Nonetheless, one must be mindful of changes to the environment and application that make comparisons problematic.
- The information is fine grained, at the MR/delta level. Such fine level data collection on a large scale would not be possible otherwise.
- The information is complete, all parts of software, documentation, test cases that are under version control are recorded.
- The way the version control system is used rarely changes, making data uniform over time.
- Even small projects generate large volumes of changes making it possible to detect even small effects statistically.
- The version control system is used as a standard part of the project, so the development project is unaffected by experimenter intrusion.

We believe that MRs are a very rich source of information about software development and that their analysis can evoke rewarding insights. Unfortunately drawing conclusions about characteristics such as effort and quality is fraught with challenges. We describe some of these challenges in the following sections. Basic to all of them is that special care must always be taken to obtain information on how version control and change management are used in the project so as not to misinterpret the MR classifications or misunderstand the process used to create, make progress on, and record information about MRs.

4. Changeability Results

Changeability involves a variety of factors and we consider several that we thought were important in software business. We start by investigating defect rates, continue with analyzing impact on effort, and then look at more structural aspects related to the scope of changes.

4.1 Defect Rate

H 1. The quality of the software in terms of customer reported defect rate is likely to improve because

the software is more transparent to maintain and there are fewer issues remaining as refactoring is likely to discover existing latent issues. We consider the number of field problems found and the root cause of these problems to test this hypothesis.

One of the main problems in legacy systems is "brittle" code, when any change is likely to generate faults. The reduction of fault-proneness is, therefore, directly related to changeability according to our interpretation.

Previously it was shown ([8], [12]) that the customer reported defects are predicted by the number of previous changes, so the number of defects have to be normalized to gage differences in quality. We did this by computing defect rates.

To make the comparison we obtained all changes related to registration domain over four releases prior to refactoring and compared them with changes for one release involving refactoring. We also obtained all changes done as a result of customer reported issues that were reported for the refactored release and four prior releases.

We had to adjust for the fact that the refactored release had been deployed only seven months while the four prior releases had longer deployment intervals. The shorter interval leads to less exposure and fewer defects are discovered by customers, see, for example, [13]. We have estimated that at least half of all defects reported for a release are discovered within seven months of deployment. This ratio depends on several factors, including how fast the deployment proceeds (i.e., what fraction of customers install the release within first few months of deployment) and how rapidly the probabilities that a later customer finds a new defect and that an existing customer finds defect long after installation decline over time from release deployment and customer installation respectively. It, therefore, is likely to be different in another product.

	pre-Release changes	cstmr rep. defects
pre-Refactoring	526	41
post-Refactoring	80	0

Table 1: Defect counts.

We found that the relative and absolute numbers (see Table 1) of field problems went down after refactoring. After we reduce the number of pre-refactoring release field problems by 50 percent to adjust for the fact that they had more exposure (as described above), the comparison of defect rates using Fisher's exact test, see, e.g., [1] for comparing proportions shows that the difference had p-value of .06 or we can reject the null hypothesis that post-refactoring defect rate is equal or higher than the rate for pre-refactoring releases with 90 percent confidence level.

Additionally, we inspected all alpha and beta problems and found that they related to issues in existing functionality and none caused by refactoring itself. Therefore, while refactoring discovered some complex issues, and, at this point, has not introduced new problems, it did not discover all the preexisting problems.

4.2 Effort

H 2. The refactoring reduces the effort required to make changes because the software is more transparent to maintain. We estimate change effort and we assess the amount of code that may need to be considered to make the change.

As illustrated in Figure 4, code for individual protocol operations, protocol composition, and protocol classification is well separated after refactoring suggesting that it may be easier to comprehend.

The effort is calculated as described in, for example, [2]. The unit (Person Month) of effort is divided among changes done by a single developer during a unit of time (one month). Therefore, changes made during months when a developer makes many other changes are assigned less effort than changes during months when the developer makes fewer changes.

We have compared effort between a sample of changes modifying files in the registration domain in the two post-refactoring releases with changes in a prior release. We have excluded three changes done to implement modification to avoid lint warnings because they do not pertain to our hypothesis.

There were 151 changes for the two post-refactoring releases and 292 changes for the two prior releases. Because change effort is highly skewed, we calculated two-sample t-test on the logarithm of effort and also Wilcoxon rank sum test with continuity correction (see, e.g., [10]) on the untransformed effort. The average of the logarithm for pre-refactoring releases was -1.12 and for post-refactoring releases was -1.23 log(Person Months) with t-value of -1.6 and p-value of 0.06. Wilcoxon rank sum test had the same p-value of 0.06.

Although it was shown that effort depends on the size of changes and on the productivity of developers [2], we did not have sufficient sample of changes to make such adjustments.

The findings of increased quality and decreased effort lead to wider adoption of the approach. In

Domain	Measure	post-Refactoring	pre-Refactoring	p-value
Registration	$\log(files+1)$	1.45	1.23	0.01
	$\log(delta+1)$	1.47	1.25	0.02
	$\log(linesAdded+1)$	3.42	3.10	0.12
	$\log(linesInFiles+1)$	6.89	6.81	0.89
Refactored	$\log(files+1)$	0.51	0.57	0.33
	$\log(delta+1)$	0.52	0.58	0.33
	$\log(linesAdded+1)$	1.52	1.87	0.09
	$\log(linesInFiles+1)$	3.58	4.74	0.00

Table 2: Averages for measures of change scope.

particular, another, much larger area is currently undergoing refactoring and a training course for a large group of developers is prepared and delivered by one of the co-authors.

4.3 Scope

H 3. The refactoring reduces the scope of changes within the restructured domain.

A number of change scope measures were introduced in [11]. There it was shown that the scope of changes in number of files, number of delta, and number of lines affected the probability that a change would contain a defect. Other work, e.g., [9], has shown that change scope in terms of number of files, number of delta, and number of lines affects the effort it takes to make a change. Finally, we were interested to investigate the effects of refactoring on such structural change properties.

The number of files reflects the number of distinct entities a developer must modify to implement a change. The need to modify a large number of files usually indicates complex changes except for administrative changes where simple and often identical modifications are made in many files, for example, to address a new lint warning. We have excluded such administrative changes from our analysis.

The number of delta. Previous work has found that the number of delta measures both the number of files and the number of lines and is a convenient single measure of change scope.

The number of lines added or changed. The large modification may require more effort and may be more likely to introduce a defect.

The number of lines in changed files. Making a modification requires developers to navigate to a particular location that needs a change. It is reasonable to assume that navigation in a very large file may present more difficulties to a programmer than navigation in a smaller file.

We collected all changes modifying files in the registration domain and compared the attributes of changes made before and after the refactoring. We also consider two definitions of the domain, one that involves all files in the domain and another definition that involves only the files affected by refactoring. We have used the more inclusive domain for the effort and quality analyses.

Table 2 shows the comparison where averages and the significance of the difference based on a two sample Wilcoxon rank sum test are shown.

We found that the number of files changed in a modification and the number of delta went up significantly after refactoring, if we consider entire registration domain. These measures have not changed significantly, if we consider only files related to refactoring.

The number of lines added and the number of lines in the modified files went down significantly when considering only files related to refactoring. There is no significant difference if we include all files related to registration domain.

One reason for the increase of the file and delta measures might be that post-refactoring releases contain a larger fraction of new feature changes than the earlier releases and new feature changes tend to be larger in scope than fixes. The fact that file and delta measures have not significantly changed, if we consider only files related to refactoring, seems intuitive. Even though functionality that used to be in a

single file was distributed over considerably more new files (cf. Figure 4), this distribution actually achieved a cleaner separation of concerns (Section 2.3). Since the number of files per concern, however, has not increased a lot, the number of files touched should not change significantly, if a modification request covers a single concern only.

An obvious reason why the number of lines in modified files that were touching refactored code went down, is the fact that the number of lines in all the refactored files did go down significantly as a result of refactoring (cf. Section 2.3).

In summary, the trend in change scope depends on the particular operationalization of scope. It is also related to a more complicated question of when functionality should be kept in a single versus multiple files and the question of optimal file size.

5. Validation

We have processed the change data to exclude modifications like administrative changes that do not pertain to the concept of changeability.

We used several operationalizations of the scope, quality, and domain to make sure that the results are invariant with respect to operationalization. In particular, we had two definitions of which files constitute the registration domain and used several definitions for the scope of changes.

We have investigated distributions of the quantities used in statistical analysis to ensure that they are satisfied. The distribution of logarithm of effort and size measures did approximate normal distribution, however, we also present results for rank-based tests that do not require restrictive assumptions regarding underlying distributions.

Because of relatively small samples we could not adjust for all the factors that may affect our modeled quantities of defect rate, effort, and scope.

Obviously, possibilities to generalize from a case study are severely limited, however we found the results surprising and compelling enough to be worth publishing. Previous studies of the impact of technology application in legacy system context ([2][3]) in different projects arrived at similar results and we see this thereby supporting previous research indicating that new technology does not always fail when applied in a legacy context.

6. Conclusion

We investigate the impact of refactoring of a legacy product on changeability. We find a mixed picture when investigating change scope where the refactoring impact depends on how we define the domain and which measures we investigate. However, the customer reported defect rates and change effort decreased in the post-refactoring releases. In comparison to previous work that found four-times decrease in change effort [3] for a completely reengineered domain, and five percent decrease [2] when using a tool designed to make it easier to make changes to legacy code, in our case we observe an intermediate 11% decrease in change effort.

Previous work investigating the impact of a tool designed to make it easier to make changes to legacy code on customer reported defects [2] found halving the probability of a customer reported failure. In our case we do not yet have enough data precisely to estimate the rate of customer reported defects after refactoring, however it is sufficient to determine that it is significantly lower.

We hope that studies quantifying tool and technology impact will become more common in the future and will enable informed decision making by software project participants and provide scientific body of evidence on relationship between plethora of software technologies and even broader array of software products and projects.

7. References

[1] A. Agresti, *Categorical data analysis*, Wiley, New York, New York, 1990

[2] D. Atkins, T. Ball, T. Graves, and A. Mockus, "Using version control data to evaluate the impact of software tools: A case study of the version editor", *IEEE Transactions on Software Engineering*, 28(7):625-637, July 2002

[3] D. Atkins, A. Mockus, and H. Siy, "Measuring technology effects on software change cost", *Bell Labs Technical Journal*, 5(2):7-18, April-June 2000

[4] Fowler, M., et al, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000

[5] Geppert, B., Roessler, F., "Collaboration-based Design – Exemplified by the Internet Session Initiation Protocol (SIP)", Working IEEE/IFIP Conference on Software Architecture, The Netherlands, 2001

[6] Geppert, B., Roessler, F, "A Complementary Modularization for Communication Protocols – Enabling Technology for IP Telephony Product Lines", International Colloquium of the Sonderforschungsbereich 501 on Software Reuse – Requirements, Technologies and Applications, Kaiserslautern, Germany, 2003

[7] Geppert, B., Roessler, F., "Effects of Refactoring Legacy Protocol Implementations: A Case Study", Metrics 2004, Chicago, USA, 2004

[8] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history", *IEEE Transactions on Software Engineering*, 26(2), 2000

[9] T. L. Graves and A. Mockus, "Inferring change effort from configuration management data", Metrics 98: Fifth International Symposium on Software Metrics, pages 267-273, Bethesda, Maryland, November 1998

[10] M. Hollander and A. Wolfe, Douglas, *Nonparametric statistical inference*, John Wiley and Sons, New York, New York, 1973

[11] A. Mockus and D. M. Weiss, "Predicting risk of software changes", *Bell Labs Technical Journal*, 5(2):169-180, April-June 2000

[12] A. Mockus, D. M. Weiss, and P. Zhang, "Understanding and predicting effort in software projects", In 2003 International Conference on Software Engineering, pages 274-284, Portland, Oregon, May 3-10 2003. ACM Press

[13] A. Mockus, P. Zhang, and P. Li, "Drivers for customer perceived software quality", In ICSE 2005, St Louis, Missouri, May 2005. ACM Press

[14] Parnas, D.; "On the Criteria to Be Used in Decomposing Systems into Modules", in *Software Fundamentals*, D. Hoffman and D. Weiss, Eds., Addison Wesley, 2001