University of Alberta
Department of Computing Science
CMPUT 790 Project Course Final Report

Assessing the Maintainability Benefits of Design Restructuring using Dependency Analysis

Prepared By: Robert M. Leitch

Academic Supervisor: Dr. Eleni Stroulia
Industry Sponsor: Dr. Phil Gray, MacDonald Dettwiler & Associates

December, 2002

ABSTRACT

The design phase of a commercial software project is tightly constrained by resources – there is a fixed amount of time and money to invest in design. The most powerful motivation on most projects is cost reduction, with the goal of finding the cheapest way to deliver the most functionality in the shortest time. Under this sort of pressure, inadequate time is spent reasoning about the long term impact of low-level software design decisions. This could lead to sub-optimal designs that are overly expensive to maintain over the project lifecycle. A major difficulty faced by developers is justifying additional investment in design at the expense more immediately beneficial activities such as adding new functionality. The reason for this is that the long-term benefits of design are hard to quantify and validated models, tools, and techniques to predict these benefits do not currently exist. A long-standing technique for improving an existing design is diligent restructuring through local code transformations, more recently re-discovered as "Refactoring". While there is general agreement that refactoring is beneficial, it is not currently possible to quantify the tradeoff between the up-front cost of restructuring and the expected downstream savings. In this work we hypothesize that a good design will cost less to maintain than a poor design. Our goal is to develop a well-defined method for estimating the costs and benefits of refactoring so that the decision of whether or not to restructure can be better informed. Specifically, we attempt to predict the Return on investment (ROI) for a planned refactoring activity where ROI = (Maintenance Savings from Proposed Refactoring) / (Development Cost of Proposed Refactoring). If the ROI is greater than or equal to one, then the refactoring will be cost effective. Well-known and validated models such as COCOMO can be used to calculate the denominator in the equation above given a defined code restructuring plan. Determining the maintenance savings of the proposed design change is a more difficult challenge. To calculate the numerator we must find a model that predicts maintenance effort for a given design. The most common approach to this problem is to try and relate design metrics to observed maintenance costs through regression analysis. One can theoretically determine which metrics influence maintenance effort and then strategically modify future designs to optimize the selected metrics. After surveying the literature regarding these approaches, we conclude that while metrics can be used to identify outlier design components and provide relative ranking of designs, there are currently no suitable predictive models of absolute maintenance effort. One reason for this is the nature of software maintenance. Corrective maintenance effort is directly related to latent defects or faults in the system, while perfective and adaptive maintenance are directly related to system enhancement in response to functional evolution or environmental changes. There is some evidence that the perfective effort category accounts for the majority of maintenance cost. Because this type of maintenance is influenced by factors external to the system, it is not obvious that such effort can be predicted by design metrics. In addition, there is a lack of agreement regarding which metrics can predict system fault density. Our attempt to construct a viable, general-purpose predictive model of maintenance effort from published regression analysis results was unsuccessful. We propose an alternative strategy for predicting maintenance savings by assuming that maintenance activities occur randomly with respect to the design. Through the use of code dependency analysis, we predict the average regression testing costs for competing designs and quantify the potential effort savings per maintenance activity. We demonstrate this approach with two exploratory Java case studies: a trial academic system with 740 SLOC and a commercial database application containing 2.5 KSLOC. The case study results provide measurements of the effects of restructuring on parameters such as mean code re-test impact, number of system data and control dependency paths, and system size. In addition, we estimate the break-even point in terms of the number of maintenance activities to achieve ROI > 1 for the proposed design transformations. Our results show that common low-level source code transformations change the system dependency structure in a beneficial way, allowing recovery of the initial refactoring investment over a number of maintenance activities.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Scope

This document is the final report for the CMPUT 790 Project Course, partially fulfilling the requirements for the degree of Master of Science (Software Technology).

1.2 Acknowledgements

The author wishes to thank Dr. Eleni Stroulia from the University of Alberta and Dr. Phil Gray from MacDonald Dettwiler and Associates (MDA) for their review and contributions toward this work. Thanks also to the NLIS project team at MDA for allowing access to their source code and documentation for the case study.

1.3 Acronyms and Abbreviations

AA	Assessment and Assimilation Increment
ACAP	Analyst Capability (COCOMOII.2000 parameter)
AHF	Attribute Hiding Factor
AIF	Attribute Inheritance Factor
APEX	Applications Experience (COCOMOII.2000 parameter)
AT	Automated Translation
C&K	Chidamber and Kemerer Object-Oriented Metrics Suite
CBO	Coupling Between Objects
CM	Percent Code Modified (COCOMOII.2000 parameter)
COCOMO	Constructive Cost Model
COF	Coupling Factor
CPLX	Product Complexity (COCOMOII.2000 parameter)
DATA	Database Size (COCOMOII.2000 parameter)
DIT	Depth of Inheritance Tree
DM	Percent Design Modified
DOCU	Documentation Match to Life-Cycle Needs (COCOMOII.2000 parameter)
FLEX	Development Flexibility (COCOMOII.2000 parameter)
IM	Percent of Integration Required for Adapted Software
KSLOC	Thousands of SLOC
LCOM	Lack of Cohesion of Methods
LTEX	Language and Tool Experience (COCOMOII.2000 parameter)
MDA	MacDonald Dettwiler and Associates
MHF	Method Hiding Factor
MIF	Method Inheritance Factor
MOOD	Metrics for Object Oriented Design
NLIS	National Land Information Service
NOC	Number of Children (i.e. immediate sub-classes)

OO	Object Oriented
PCAP	Programmer Capability (COCOMOII.2000 parameter)
PCON	Personnel Continuity (COCOMOII.2000 parameter)
PLEX	Platform Experience (COCOMOII.2000 parameter)
PMAT	Process Maturity (COCOMOII.2000 parameter)
POF	Polymorphism Factor
PREC	Precedentedness (COCOMOII.2000 parame ter)
PVOL	Platform Volatility (COCOMOII.2000 parameter)
RELY	Required Software Reliability (COCOMOII.2000 parameter)
RESL	Architecture/Risk Resolution (COCOMOII.2000 parameter)
RFC	Response for Class
ROI	Return on Investment
RUSE	Developed for Reusability (COCOMOII.2000 parameter)
SCED	Required Development Schedule (COCOMOII.2000 parameter)
SITE	Multi-site Development (COCOMOII.2000 parameter)
SLOC	Source Lines of Code (logical source statements)
STOR	Main Storage Constraint (COCOMOII.2000 parameter)
SU	Software Understanding Increment
TEAM	Team Cohesion (COCOMOII.2000 parameter)
TIME	Execution Time Constraint (COCOMOII.2000 parameter)
TOOL	Use of Software Tools (COCOMOII.2000 parameter)
UNFM	Programmer Unfamiliarity
WMC	Weighted Method Complexity

2 Literature Survey

2.1 Introduction

The question of what constitutes "good" or "cost-effective" design is something that confronts software developers and software project managers on a daily basis. It would be extremely useful to develop a method for these practitioners to quantitatively assess the cost-benefit aspects of their proposed design decisions. A challenging goal is therefore to establish a basis for making intelligent design decisions from the "value-added" perspective. As stated by Boehm and Sullivan in their analysis of the current state of Software Economics [23], most software design decisions today are made by finding the minimum cost approach as opposed to searching for the "maximum value" solution. In the field of software engineering, we are currently seeing a paradigm shift as more research is focused on the measurement of added value in design and architecture.

The goal of this project is to explore the cost-benefit equation related to low-level software design restructuring (i.e. "refactoring"). The key hypothesis of this project is that there exists a relationship between specific design properties and downstream development effort. In other words, if we invest in refactoring to strategically modify the design properties of a system, then we should expect a future payoff. In this project we have constrained the problem by focusing on the maintenance effort required for legacy systems. The problem is summarized by the following question: Is it worth the cost to re-structure the design of a legacy system in order to save future maintenance effort?

2.2 Definitions

Before outlining our approach to investigating this problem, we would like to clearly define a number of important concepts.

"Return on Investment" (ROI) is the measure of the effectiveness of proposed design changes from a cost-benefit perspective. Since we are interested in creating a predictive model, we will be attempting to assess the ROI in advance of actually implementing changes to a legacy system. In this context, our definition of refactoring ROI is the following:

ROI_p = (Maintenance Savings from Proposed Refactoring) / (Development Cost of Proposed Refactoring)

From this definition, ROI_p greater than one indicates a positive return on investment (i.e. projected benefits outweigh the costs), while ROI_p less than or equal to one indicates a break-even or negative return on investment. Note that we choose to measure both the Maintenance Savings and the Development Cost in terms of effort (e.g. person-months), as this is the parameter most relevant to project managers. In addition, ROI is typically defined with a specific time horizon – in this study we do not limit the time horizon when trying to determine the break-even point for the investment. In practice, in order to

achieve a positive ROI the break-even point must be considered against a pre-determined time threshold.

The IEEE standard definitions for software maintenance and maintainability, as documented in [26], are the following:

Maintenance: The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.

Maintainability: The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.

Coleman, Ash, Lowther, and Oman [15] extend these standard definitions to identify three main "areas of focus" for the software maintenance process:

- *Corrective maintenance*: Maintenance performed to correct faults in hardware or software.
- *Adaptive maintenance*: Software maintenance performed to make a computer program usable in a changed environment.
- *Perfective maintenance*: Software maintenance performed to improve the performance, maintainability, or other attributes of a computer program.

In this study, we use the Corrective/Adaptive/Perfective classifications as defined above when referring to maintenance effort.

2.3 Proposed Methodology

In order to address the cost-benefit question, we first need to establish a viable approach for attacking the problem.

The first phase of the project considered the use of existing empirical models as published in the software engineering literature. In order to establish a predictive model of refactoring ROI_p, the following empirical models are required:

a) Models relating design metrics to system maintenance effort (Maintenance Effort Prediction Category).

b) Models relating low-level design restructuring operations to changes in design metrics (Refactoring Impact Category).

c) Models that predict the cost of implementing design re-structuring operations (Refactoring Cost Prediction Category).

For the first category of empirical models, the idea is to find one or more models that relate maintenance effort (as the independent variable) to one or more specific software design metrics (as dependent variables). Such models would be validated by empirical data from the field. We hoped to find models that would allow us to predict the future maintenance effort for a given legacy system – the input to a model would be design metrics from the code, and the output would be maintenance effort.

For the second category of empirical models, the goal is to find a way to estimate the impact of refactoring without actually having to perform the refactoring. These models would be based on experiments that evaluate the impact (as measured by specific design metrics) of low-level design restructuring operations.

The third category of empirical models is more straightforward. In this category, we are searching for a model to estimate the up-front development cost of re-engineering the legacy system. These are the costs associated with analyzing the system for restructuring, re-coding, and re-testing the modified parts of the code. These models should also take into account the impact of automated tools on the re-engineering cost, as well as associated cost items such as developer training (if this is a new technology).

Using these three empirically derived models, the proposed methodology for this analysis is described by the following sequence:

a) Select a legacy system for analysis.

b) Find empirical models in the literature that relate specific design metrics to downstream maintenance effort. At a minimum, such models must be relevant to the type of legacy system under consideration.

c) Establish a baseline quality measurement of the legacy system by extracting design metrics from the code. These metrics are specified by the experimental independent variables from the models identified from step b).

d) Using the baseline quality measurement as the input, run the predictive model(s) chosen in step b). This will establish a baseline maintenance effort prediction for the system.

e) Analyse the legacy code for design re-structuring opportunities. For example, search the code for instances where refactoring could be applied as defined by Fowler et al in their Refactoring text [1].

f) Find empirical models in the literature that relate design re-structuring activities to changes in software design metrics.

g) Establish the predicted design quality improvement for the legacy system by using the model(s) found in step f) in conjunction with the design restructuring opportunities

identified in step e). In other words, establish a predicted set of design metrics for the legacy system that would result from the proposed restructuring activities.

h) Re-using the maintenance effort prediction model from step b), establish the predicted maintenance effort for the "improved" legacy system - the input to this run is the set of modified design metrics established in step g).

i) Compare the maintenance effort results obtained in steps d) and h) above. The difference between these results is the benefit associated with the planned refactoring operations.

j) Find an empirical model from the literature that can predict the development effort required to re-structure the legacy system as per the list of refactoring opportunities established in step e).

k) Using the model identified in step j), establish the predicted cost of modifying the legacy system to implement the proposed restructuring.

l) Calculate the Return on Investment (ROI) of the proposed design re-structuring by dividing the result from step i) with the result from step k). If the ROI is greater than one, this indicates that the re-structuring operation has a net benefit and should be implemented.

This methodology is described at a high level in Figure 1, which uses a data-flow-diagram representation.

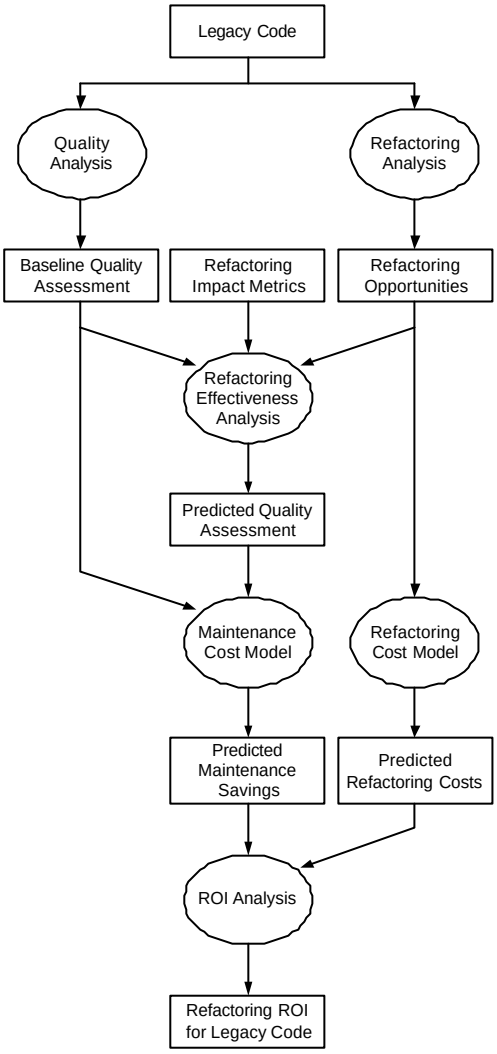

Figure 1 – Refactoring ROI Analysis Methodology

Note that this process could be used iteratively to find an optimal set of refactorings that produce the biggest downstream benefit for the legacy system. In other words, this algorithm could be re-executed for different combinations of refactoring opportunities in order to determine the set of transformations that maximize the ROI. In order to support this approach the selected empirical models must be accurate when dealing with relatively small system changes (i.e. "micro" re-engineering operations).

2.4 Model Evaluation Criteria

When evaluating a model that predicts maintenance effort from design properties, it is very important to establish how the model addresses each of the three types of software maintenance (perfective, corrective, adaptive). A predictive effort model can only be considered complete if it handles each of the key maintenance process aspects.

There is some evidence in the literature that Perfective maintenance accounts for the majority of overall maintenance effort in a project. Examples of this evidence include the following:

- Polo, Piattini, and Ruiz (PPR01) [11] report on the maintenance effort distribution for 55 applications (database programs written in COBOL for the banking industry) over two years of operations. These programs range from very small applications of a few thousand SLOC to very large applications containing over a million SLOC. The study indicates that 88% of the total maintenance effort for these programs was expended in "Perfective" maintenance, while 12% of the effort resulted from Corrective maintenance. This study only classified effort into these two general categories, suggesting that "Perfective" effort is actually the sum of Perfective and Adaptive effort.
- Basili et al (BAS96) [13] studied the maintenance effort for the flight dynamics software developed by the Goddard Space Flight Center. This code base includes over 100 applications totaling approximately 4.5 million SLOC, mostly written in FORTRAN. On average, they report that 61% of the total maintenance effort was spent on "enhancement" which is equivalent to Perfective maintenance according to our definition. Of the larger applications in the study, the percentage of Perfective maintenance effort varied between 51% and 89% of the total effort. "Correction" accounted for an average of 14% of the total effort, while "Adaptation" accounted for 5% of the total effort. The remaining 20% is classified as "Other" - this category contains effort that could not be neatly classified (for example training effort for the maintainers as a result of a port to a new environment).

This evidence suggests that Perfective maintenance, which is directly related to the "evolvability" properties of the software, is the dominant factor that must be addressed by a predictive model.

Another interesting issue is the distribution of maintenance effort among the individual modules that make up each application. There is some evidence in the literature that suggests a small percentage of modules account for a large percentage of the application's maintenance effort. This evidence is mostly for Corrective maintenance, which is closely related to fault density in the delivered code. For example, Fenton and Ohlsson [14] state that 10% of the modules account for between 80% and 100% of the observed operational failures. They suggest that the Pareto principle (i.e. the 80/20 rule) could apply to fault density (it follows that this would then apply to Corrective maintenance effort). If this is the case, then a predictive maintenance model used to judge refactoring benefits must look at the code down to at least the module level. For example, if one were to apply preventative restructuring uniformly over all the modules of a system, it may be that only 10% of this refactoring has a beneficial impact to the overall corrective maintenance effort.

In summary, there are some general criteria that must be applied when trying to construct a general-purpose predictive maintenance model:

- The model must address the Corrective/Perfective/Adaptive aspects of the process, with the Perfective aspect possibly being the most important.
- The model must look at system design properties at least down to the module level.
- The model must be applicable to applications of different overall sizes and types.
- The model must be applicable across development environments (i.e. organizations).

The last two criteria are almost certainly not achievable given the current state of research. Every study that offers a predictive model also clarifies that the results cannot be extended to other environments or other classes of systems.

2.5 Literature Survey Results

2.5.1 Maintenance Effort Prediction Models

Briand and Wüst [16] have performed a comprehensive analysis of the empirical studies that exist concerning software quality models for Object-Oriented systems. This analysis examines every previous study in the software engineering literature in which scientific attempts were made to correlate software quality metrics with some form of dependent variable (e.g. fault-proneness, development effort, maintenance effort, etc.). Reviewing Table 1 of their analysis, we find four independent studies in the literature that attempt to correlate design metrics with maintenance or "rework" effort:

- AM96: Abreu and Melo, [18]
- CDK98: Chidamber, Darcy, and Kemerer, [19]
- LH93: Li, Henry, [25],
- WH98: Wilkie and Hylands, [27]

Regarding the design metrics used in these studies as independent variables, AM96 analyses a set of "MOOD" metrics while the remaining studies use various forms of the famous "C&K" metrics that have been derived by Chidamber and Kemerer [20].

2.5.1.1 AM96 Summary

This study measures the correlation of the "Metrics for Object Oriented Design" (MOOD) set with resulting defect density and rework effort. The data analysis technique employed by the authors is univariate linear regression. The experimental data for the study includes measurements of rework effort and defect density from eight small C++ systems developed in a university environment (the initial developers were students). A waterfall development process was followed, using Object-Oriented analysis, design and coding techniques.

The eight software systems under investigation were small applications ranging in size from approximately 5 KSLOC to 14 KSLOC. Each of these systems was designed and implemented by a separate group of students. The paper does not provide much detail regarding the functional requirements of these systems, but does say that they are information management systems to support a hypothetical video rental business.

The MOOD set includes the following metrics (see the paper itself for extensive definitions): Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Polymorphism Factor (POF), and the Coupling Factor (COF). Of these metrics, MIF, COF, and POF were found to have good correlation with defect density and "normalized" rework.

Since we are seeking a predictive model of maintenance effort, the "rework" results from this study are the most relevant. "Rework" as defined in the study is the effort expended by the students in the "repair" phase of the project in order to fix problems detected by an independent professional test team. This definition does not fully coincide with the definition and scope of "maintenance effort" established for this project. For example, the rework effort measured by AM96 does not include the impact of system enhancement and evolution which represent the dominant components of overall maintenance costs (as we have stated previously). In other words, the "changeability" of the systems was not investigated. As well, the rework effort in AM96 is really repair effort from a final stage of testing – it does not represent effort to repair latent faults uncovered by users in the field. As a result, while this study provides some valuable data, it does not meet all our criteria for use in the refactoring cost/benefit equation. It is certainly not clear that these results can be applied to predict real maintenance effort for other systems – the only concrete conclusion is that for the metrics and specific systems analyzed, there was some significant correlation.

The table below summarizes the results from AM96, including the linear equations predicting rework effort based on MIF, COF and POF.

Linear Model Properties	MIF	COF	POF
Slope (person-hours/KSLOC)	-19.64	77.31	-75.76
Y-intercept (person-hours/KSLOC)	15.64	4.20	12.41
Pearson "r" coeff. of correlation	-0.78	0.91	-0.71

Table 1 – AM96 Model Results

The COF metric has the highest coefficient of correlation, and appears to be the most promising candidate to predict rework effort. Closer analysis of the COF data reveals that the average error of the model (i.e. the average difference between the modeled value and the measured value for the eight experimental systems) is approximately 37%. Across the eight systems, the modeling error ranges from 11% to 106%.

2.5.1.2 CDK98 Summary

This study is an exploratory investigation into the predictive powers of the "C&K" OO design metrics with respect to three dependent variables: programmer productivity, "rework" effort, and design effort.

As stated in their paper, the six "C&K" metrics can be described by the following simplified definitions:

- NOC: count of the immediate subclasses of a class,
- WMC: count of the methods in a class,
- DIT: maximum path length from the current class to the root of the class hierarchy,
- CBO: count of the number of couples between the current class and other classes (e.g. external method invocations),
- RFC: the total number of methods that could possibly be executed as a result of a message arriving at the current class,
- LCOM: count of the number of method pairs in a class that do not share a common instance variable minus the count of the method pairs in a class that do share one or more instance variables (i.e. unlike method pairs minus similar method pairs).

As seen from the above definitions, the C&K design metrics are at the class level.

The study attempts to correlate these six design metrics with productivity and effort data measured from three commercial banking applications (all developed within a single organization). The first system (TPM) is a 15 KSLOC C++ database application possessing 45 classes. The second system (FIS) is a 2.7 KSLOC "Objective C" application possessing 27 classes. The third system (SLB) was not yet coded; the metrics were derived from design documentation. The paper does not describe any details regarding the development process used for each system.

Regarding dependent variables, "rework" effort was analyzed for the FIS system only. Immediately, due to the small size of this single application, we can anticipate that the results will not be extendable to a general-purpose predictive maintenance model.

In this study, "rework" is defined as the effort to make a class usable in another project. This is more a measurement of reusability than maintainability – this definition does not obviously include effort to correct defects, nor does it fully account for maintenance due to system evolution. This definition introduces a major problem when trying to interpret the rework results presented in CDK98. It is not clear whether the rework effort was expended as a result of the class design properties or instead as a direct result of the new functional requirements associated with the adaptation of the class to the new project.

In terms of the predictive power of the C&K metrics, the CDK98 study concludes that the CBO and LCOM metrics are correlated with rework effort. More specifically, "high" values of the CBO and LCOM metrics have a positive correlation with rework effort. Regarding the definition of "high", the authors apply the 80/20 Pareto principle. The

other metrics do not show significant correlation – in particular, the inheritance-based metrics (NOC and DIT) do not appear to be useful effort predictors.

In summary, the results of CDK98 are not suitable to include in a general-purpose predictive maintenance model. The reasons for this are:

- The sample size is extremely small (one application),
- The system size is extremely small (2.7 KSLOC), and
- The study's definition of "rework" does match very well with the standard definitions of software maintenance that include corrective and perfective components.

2.5.1.3 LH93 Summary

Henry and Li [25] studied the correlation of a number of OO design metrics with maintenance effort, using project data from two commercial systems over three years. The systems were coded in a language called "Classic-Ada" – the first system was approximately 4 KSLOC in size, while the second system contained 16 KSLOC. In addition to investigating the C&K metrics, the authors looked at coupling metrics for OO systems and code size metrics at the class level. The basic approach was to develop a regression model relating effort measurements to different groups of the analyzed metrics.

The authors define their dependent variable (i.e. maintenance effort) as the "number of lines changed per class in its maintenance history". From their regression model the authors conclude that the metrics are good predictors of the dependent variable - most of the change in the effort measurements can be explained statistically by changes within the group of independent variables. The study also provides evidence that while size metrics alone have predictive power for maintenance effort, the other design metrics are also strong contributors over and above what can be predicted by code size. The authors go on to refine their analysis and arrive at a smaller group of eight metrics (the compact model) that have potential for predicting maintenance effort.

While this study provides evidence that there is likely a relationship between the code changes in these systems and design metrics, it is not easy to apply these results in a meaningful way to our cost-benefit analysis. For starters, the study does not present a predictive model – it only suggests that there is a correlation within the collective group of the metrics and that there is the potential of determining a model. In addition, there is no breakdown of the "effort" measurements into perfective, corrective, and adaptive categories. The nature of the maintenance activities measured in this study is not evident from the paper.

As well, the definition of the dependent variable in this study presents a problem when trying to use the results in a cost-benefit analysis. The issue is whether or not counting changed lines of code is a fair measure of maintenance effort. We suggest that this is not necessarily a good measure, mainly for the following reason: when correcting a defect in

the maintenance phase (a corrective change), it may take a substantial amount of time to isolate and determine the solution to the problem. For example, it may take several days of programmer effort to arrive at and implement a very small code change in a poorly designed system. By measuring only the end result of the change, this method does not account for such expense. This may also be the case for perfective maintenance - if a system is designed such that it is not easily maintainable, then it may require more effort to enhance than other equivalent systems. The dependent variable used in [25] will not pick up on this relative difference between systems. These factors make the LH93 results difficult to use in our cost-benefit equation, especially when we wish to rate the relative merits of competing designs. The bottom line is that we cannot see a way to extend these results to actually predict maintenance cost.

There is another issue with this definition of maintenance effort – by only looking at the number of lines of code changed within each class, it becomes extremely difficult to assess the effects of perfective maintenance. For example, the fact that a particular class has many lines of code changed over its maintenance history does not necessarily mean that the class is poorly designed. It could be that the changes are the result of functional enhancements that have nothing to do with "bugs" or latent defects in the code. In this way, perfective maintenance is not really accounted for in this approach – the maintenance is assumed to be corrective in nature.

2.5.1.4 WH98 Summary

Wilkie and Hylands [27] examined the C&K metrics suite in detail and studied the change in these code metrics for a commercial C++ application over its 2.5 year maintenance history. The system under investigation contained approximately 25000 lines of code within 114 classes. The dependent variable in the study was maintenance effort (classified either as "bug fix" or "enhancement" effort), where maintenance effort is defined as the number of lines of code changed in each class. This definition is similar to the definition from LH93, with the same problems regarding the application of the results to an absolute cost-benefit model. Records of actual person effort were not available to the researchers.

The results of this study provide insight into what happens to a system under maintenance – the metrics collected from the code over time reflect the architectural impact of maintenance activities. Each of the C&K metrics was trended over the maintenance period (at the class level), and the study presents the results for each metric in the form of frequency charts. The authors found that the WMC metric tended to increase with each new version of the system, suggesting that maintenance increases program complexity. (Note that the study considers two adapted forms of the C&K WMC metric, one based on Cyclomatic Complexity and the other based on Halstead Software Science metrics – the complexity-based WMC clearly increased over time). The RFC and LCOM metrics also tended to increase over time. LCOM in particular appeared to be a good predictor of future class evolution. There was no decisive trend in the other C&K metrics during the maintenance period.

In addition to the trending analysis, the authors performed a stepwise multivariate regression analysis to determine whether the metrics were correlated with fault proneness in the system. The results of this regression analysis indicate that WMC (Halstead-based) and DIT are the only metrics that can significantly predict fault proneness. More precisely, the study concludes that 48% of the variation in corrective maintenance effort can be explained by a combination of the WMC and DIT metrics. This result is not as strong as the result from the LH93 study discussed above where the authors concluded that the measured changes in the C&K suite could account for a much larger percentage of the maintenance effort variation.

Enhancement effort (i.e. perfective maintenance) was also measured in the study, and the authors again applied a multivariate regression analysis to determine the predictive powers of the C&K metrics suite. The authors could not find a significant contributor in the C&K metrics for enhancement effort prediction – the only significant metric appeared to be RFC which could account for only 13% of the observed variation in enhancement effort. The authors make a very interesting point: because perfective maintenance is driven by external requirements changes (e.g. for business reasons), the complexity based metrics cannot be expected to predict the resulting enhancement effort. This poses a fundamental problem when we try to quantify system maintenance effort over time – applicable design metrics for perfective effort may not exist.

The authors conclude that the C&K metrics suite is useful for examining the design impact of changes during the maintenance phase, and the metrics should be used as a tool for detecting "outlier" classes that may warrant restructuring. However, there is no strong evidence from this study that a predictive maintenance effort model can be built from the C&K metrics.

2.5.1.5 Other Studies

As mentioned above, Polo, Piattini, and Ruiz (PPR01) [11] attempted to correlate maintenance effort with module size and number of modules using a databank of 55 COBOL applications. However, they could not derive a meaningful effort prediction equation from their data, and instead produced a complex equation designed to give a binary result. This result (either a 1 or a 0) indicates whether or not the particular application will present above average or below average maintenance challenges. While this is possibly useful to predict "outlier" systems that require special remedial attention, this equation is not useful in our refactoring cost-benefit analysis.

Coleman, Ash, Lowther, and Oman [15] developed an equation using polynomial regression to calculate a system's "maintainability index" based on four input metrics. These metrics are: Halstead's volume metric, extended V(G) (McCabe Complexity), average lines of code, and number of comment lines per submodule. The maintainability index equation stated in [15] is:

$$MI = 171 - 5.2*\ln(aveVol) - 0.23*aveV(G) - 16.2*\ln(aveLOC) + 50*\sin(SQRT(2.46*perCM))$$

The above equation yields a number, and the authors contend that a high number indicates good maintainability while a low number indicates poor maintainability. Based on the Hewlett-Packard project data used in the study, the authors conclude that an MI greater than 85 represents good maintainability while an MI less than 65 represents poor maintainability that requires attention or some sort of corrective action in the design.

Unfortunately, this equation does not relate the maintainability index back to maintenance effort. As such, the "Coleman-Oman" model is a tool that seeks to identify candidates for refactoring. This tool does not predict the relative maintenance effort for two competing designs. However, it does indicate whether a design is "good" or "bad" with respect to potential downstream maintenance. The index also provides way to check the impact of software changes over time – in this sense it could be very useful for design trending and monitoring. Unfortunately, the index does not provide a means for predicting absolute maintenance effort for a given design. It is therefore difficult to use this index in the refactoring cost-benefit analysis model.

2.5.1.6 General Conclusions for Maintenance Effort Prediction Models

Based on an analysis of the available literature regarding empirically-derived maintenance effort prediction models, we make the following general observations:

- No general-purpose models exist for predicting maintenance effort based on system design properties.
- There is some agreement (but not clear, consistent results) regarding what design metrics are interesting from a maintenance perspective. In particular, the C&K metrics suite has potential for predicting corrective maintenance effort.
- There is little evidence of direct correlation between design metrics and perfective maintenance effort. As perfective maintenance often accounts for the majority of costs, this presents a serious problem.
- There is some empirical evidence from the four studies described above that establishes correlation between design properties and maintenance effort, but the models from these studies cannot be directly applied to other systems or other organizations.
- The main result of the empirical research into maintenance effort prediction is that design metrics can identify "outliers", or rogue system components that may warrant remedial attention. This is a useful result, but it does not help establish a model to compare the absolute maintenance costs of various design approaches.
- There is a lack of consistency regarding the definition of dependent variables between studies in this area. For example, the definitions for "rework" in AM96 and CDK98 do not match so the results cannot be usefully compared. In addition, the definition of maintenance effort in LH93 and WH98 is not a direct measure of person effort and does not match the definitions from AM96 and CDK98.
- There is only a small volume of scientific data available in the literature – the four substantial studies described above only provide insight into 14 systems (written in

various languages). The empirical studies are based on limited sample sizes and the system sizes are small (most under 10 KSLOC). This makes it difficult to apply the results to other larger systems, and it is also difficult to draw general conclusions from the pool of results.

- The main accomplishment of these empirical studies is that they demonstrate how an organization might go about creating a self-calibrated model that predicts maintenance effort. If an organization religiously collects the right data and continuously re-calibrates its model, there is hope of making decent effort predictions. However, it is clear that no universal model has so far been identified.

2.5.2 Refactoring Impact Models

Models in this category attempt to relate low-level design transformations to changes in quality metrics. We are interested in a predictive model – we do not want to actually perform the refactoring (i.e. make the investment) in order to assess the potential quality impact.

Tahvildari, Kontogiannis, and Mylopoulos [3] have created a software re-engineering framework that considers the impact of design level and source code level transformations on system performance and maintainability metrics. Their premise of legacy system re-engineering is very similar to the refactoring we are investigating in this project. In their study, the authors link certain design transformations to maintainability using "soft" goal interdependency graphs constructed according to the Non Functional Requirement (NFR) framework. The soft goal graph presented in [3] makes connections between maintainability and design qualities such as: high modularity, high cohesion, low coupling, low I/O complexity, good commenting and naming practices, high encapsulation, and high reuse. This graph was constructed by the authors based on a literature review.

When analyzing maintainability, the authors use some familiar metrics to assess the pre and post-transformation states of the system: lines of code, the Halstead suite of metrics, McCabe cyclomatic complexity, and the number of comment lines per module. They use these metrics to compute three different maintainability indices, two of which are versions of the Coleman-Oman model [15] described previously.

The source-code level transformations associated with improving maintainability are assumed to be the following:

- Maximize Cohesion (splitting methods and functions),
- Minimize Coupling (introduce parameter passing and reduce global data flow),
- GOTO Statement Elimination,
- Global Data Type Elimination,
- Dead Code Elimination.

In addition, the study suggests that some architectural-level changes in the form of design pattern introduction will also improve maintainability. The target design patterns selected for this purpose (from the repository in [28]) include: Factory Method, Abstract Factory, Composite, Facade, Iterator, and Visitor. The basis for selecting these particular patterns is not totally clear in the paper.

The study provides some experimental data from two systems: an election tabulation system (4.25 KLOC) that has been maintained for approximately 30 years, and the GNU AVL libraries (4 KLOC) written in C. The study suggests that these are "medium" sized systems, but they are actually very small compared to most industrial or business programs.

Results of the experimental application of re-engineering to these systems indicate that most of the source code level transformations had a positive impact on the maintainability indices, especially the elimination of dead code. In addition, four of the design patterns also had a positive impact on the maintainability indices.

We encounter considerable difficulty when trying to use these results in our cost-benefit analysis. Firstly, the study does not quantify the amount of transformation applied to the code. In other words, how much of the code was changed to produce the stated improvements? Secondly, the results are obtained for two small systems with very specific initial properties. There is no way to say whether different systems (with different initial states in terms of the quality metrics) will yield similar improvements when transformed in this way. While this research is interesting and focuses attention on the process of achieving quality objectives, the results are not useful for our predictive cost-benefit model.

As the study in [3] is the only substantial work we could find that tries to quantify the maintainability impact of specific refactoring operations, we conclude that there is currently no way to predict the impact of proposed refactoring based on an initial set of design quality conditions.

2.5.3 Refactoring Cost Prediction Models

A model to predict refactoring effort for a legacy system must account for the following development activities:

a) Identification of candidate refactoring opportunities in the legacy code.
b) Selection of design changes for implementation.
c) Update of design documentation (if necessary) to reflect the changes.
d) Code and unit test of all potentially affected modules.
e) Update of test specifications and plans (if necessary) to reflect the design changes.
f) Regression testing of the system.

Note that the amount of system regression testing required is subjective and depends on the nature of the application. Some projects may require full regression testing (e.g. for a safety or mission-critical system) in order to guarantee that the refactoring has not introduced any new errors. With the advent of automated tools to support refactoring the effort model must also take into account manual execution of these processes versus tool-based execution.

The refactoring activities list above is simply an example of a standard waterfall development model. For such processes, there are many useful software cost estimation models in the literature. The most successful of these is the COCOMO model first created by Barry Boehm in 1981. This model has been revised and updated since its original release to reflect changes in software engineering technology and methodology. The latest release of the model is called COCOMO II.2000, described by Boehm et al in [9]. This latest model is based upon a dataset of metrics collected from 161 software projects.

The difficulty in using COCOMO is that it relies upon subjective inputs from the user regarding a wide range of model input parameters. Even so, the predictive accuracy of the model is shown to be within 30% of the project actuals 75% of the time [9]. With local calibration of the model parameters to a particular organization, this percentage rises to 80%. This level of accuracy may be suitable for predicting refactoring costs for our model. Certainly, we can make conservative assumptions regarding the input parameters in order to find an upper bound for the development cost. The question is how to use COCOMO to best represent the process of refactoring.

While the COCOMO model does not explicitly predict refactoring effort, it does incorporate a model to deal with software re-use. The model considers varying degrees of re-use, from none to "adapted" software to complete re-use. Adapted software is code that is not directly re-used, but must be modified to some extent in order to form part of a new product. The refactoring of a legacy system can be thought of as an exercise in software adaptation. We are transforming an application by changing parts (but not all) of the design. This is very similar to the COCOMO definition of adapted software, and we can attempt to apply the re-use model to predict the re-engineering effort.

As a result, we consider the COCOMO II.2000 model to be useful in our ROI analysis and we propose to use this model in performing sample ROI calculations for a case study.

2.6 Literature Survey Conclusions

Of the three types of empirically-derived models required in order to analyze the refactoring cost-benefit equation, we can find usable models for only one category: refactoring effort cost prediction. While there is some published data in the other two categories, there is not enough data to construct viable models for use in our predictive study. Specifically, general-purpose models that predict maintenance effort from design

properties do not currently exist. Similarly, models that can accurately predict the quality impact of design transformations do not exist.

Due to the lack of available models, our proposed approach for calculating the refactoring ROI is not viable at this time. In order to make progress on this problem we must find an alternate method of assessing the benefit side of the re-engineering cost-benefit equation.

3 Revised Methodology Definition

3.1 Introduction

In section 2, we showed that it was not feasible to construct a general purpose cost-benefit model for refactoring based on existing empirical models that rely on design metrics. In this chapter, we explore an alternate approach to assessing the benefits of design re-structuring that is based upon code dependency analysis techniques.

When reasoning about what makes a particular design more maintainable than another design, we must consider the economic impact of the maintenance process. A "good" design from a maintainability perspective should cost less to maintain than a "poor" design. The major difficulty in directly correlating design properties to maintenance cost data is the nature of the maintenance task itself. It is extremely difficult to determine whether the cost of a particular maintenance activity is related directly to the software design itself or some other environmental factor (e.g. poorly understood requirements, unanticipated functional evolution, etc.). In addition, when trying to create a predictive model of maintenance cost it is also extremely difficult to estimate important factors such as fault density and defect distribution within a delivered system. In fact, there is very little support for some commonly held beliefs in the software engineering community regarding fault density as it relates to code size and complexity [14].

A potential way around this uncertainty is to assume, for predictive purposes, that the maintenance process is random with respect to the design – a maintenance activity could strike a legacy system at any location in the design with equal probability. The activity could fall into to any one of the maintenance process categories defined in Chapter X (i.e. perfective, corrective, or adaptive).

Maintenance costs will be driven by the number of maintenance activities that occur for a given system over its operational lifetime, as well as the cost to complete each activity.

In the field of software change impact analysis, techniques have been developed to evaluate the scope of design modifications that occur during the maintenance process. These techniques fall into two general categories: traceability analysis and dependency analysis. We will focus on the latter, which can be used effectively to estimate the amount of regression testing required for a system assuming a part of the system experiences change.

Potential cost savings in the regression testing of a system under maintenance represent real, tangible benefits that can be linked to code structure through dependency analysis. We will show in this chapter how this property can be exploiting to estimate the relative maintenance effort of competing designs, leading to a feasible cost-benefit analysis for refactoring a legacy system.

In this chapter we will describe the dependency analysis methodology, and then present preliminary results from a trial case study of a small Java system.

3.2 Proposed Methodology

In this section we define the following algorithm as a method to calculate the refactoring ROI:

1. Using the dependency analysis technique defined in [29], create procedure-level dependency graphs for the legacy system. These graphs include data and control dependencies.

2. Review the legacy system code and identify opportunities for refactoring, based on the criteria established in [1].

3. Review the list of refactoring opportunities and arrive at a detailed re-structuring plan for the legacy system, identifying new and modified procedures in the design.

4. Based on the restructuring plan, create revised data and control dependency graphs for the legacy system representing the predicted state of the system after implementing the proposed restructuring.

5. For the legacy system prior to refactoring, construct a list of procedures and the number of lines of code in each procedure. The percentage of the overall code contained in each procedure represents the probability of a random maintenance activity striking that procedure.

6. For each procedure in the system, identify from the dependency graphs how much additional code in the system must be checked or re-tested as a result of a change to the procedure. If the dependency graph is expressed as a matrix with each element identified by {row, column}, a dependency exists from row to column if there is a "1" in the matrix at this location. The list of procedures dependent on a particular procedure corresponds to the list of "1s" in the matrix column associated with that procedure. This column multiplied by the amount of code in each procedure yields the expected amount of code to be regression tested as a result of the maintenance activity.

7. For each procedure in the system, take the result from step 6 above and multiply this result by the percentage of the overall code contained in that procedure. The sum of all these results for the system yields the average expected regression testing impact as a result of a maintenance activity.

8. Repeat steps 5-7 above for the predicted legacy system state post-refactoring.

9. Estimate the effort required to implement the restructuring plan identified in step 3 using the COCOMOII.2000 model.

10. Using an assumed maintenance scenario for both designs, calculate the anticipated maintenance costs for regression testing again using the COCOMOII.2000 model.

11. Subtract the maintenance cost for the proposed restructured system from the predicted cost of the baseline system design to determine the effort savings associated with the restructuring.

12. Divide the result from step 11 by the estimated cost of refactoring from step 9 in order to determine the Return on Investment.

3.2.1 Dependency Analysis Description

The technique we have used for procedure-level dependency analysis is based on the language-independent methodology defined by Loyall and Mathison in [29]. The concepts of data and control dependency are formally defined in [29], but we can summarize these definitions as follows:

A procedure X is data-dependent on procedure Y if procedure X uses data (either a global variable or passed parameter) that has been produced or modified by procedure Y.

A procedure X is control-dependent on procedure Y if procedure Y invokes procedure X. In [29], they make a distinction between ordinary control dependence and "strong" control dependence – for the purposes of this analysis we assume that any procedure invocation implies dependence.

It must be noted that the analysis technique from [29] does not explicitly cater to object-oriented languages. In our study, we perform a case study using a Java-based system. The main assumption made in applying this analysis technique to a Java program is that the class-level data definitions as well as the class creation method are associated with a single entity (i.e. they form a single "procedure" in the analysis). In addition, when assessing the code size of subclasses we did not include methods that were directly inherited from the superclass – only "new" code was considered in the analysis of the sub-class. As well, when considering a procedure that instantiates a class (i.e. creates a new object), we consider the class creation method to be control dependent on the instantiating procedure.

In creating the top-level dependency graph for a legacy system design, we first create two separate graphs – the direct data dependency graph and the direct control dependency graph. These graphs are "adjacency" matrices that capture directed dependencies between each distinct pair of nodes (with each node representing a procedure in the system). These graphs do not capture the overall connectivity – in other words, they do not answer the question of whether or not there is a dependency path in the graph between two particular nodes. In order to answer this question we calculate the transitive closure of each graph using Warshall's algorithm. This results in the overall data and control dependency graphs for the system.

In order to evaluate the regression testing impact when modifying a particular procedure, we perform the logical "OR" of the overall data and control dependency graphs. Before performing this operation, it is necessary to take the transpose of the control dependency graph. The reason for this is that a control dependence from procedure X to procedure Y means that that Y must be checked if X changes. On the other hand, a data dependence from procedure X to procedure Y means that X must be checked if Y changes. By taking the transpose of the control graph prior to the combination with the data graph, we ensure consistency from a regression testing impact perspective.

This approach to dependency graph construction is conservative as described in [29], and will generally overestimate the number of dependency paths within the system.

3.3 Trial Case Study – Traffic Light Simulation

3.3.1 Introduction

In this section we attempt to apply the preceding methodology to a small Java system in order to assess the effectiveness of the approach. The trial system was created in a student environment as part of a graduate course in Object-oriented analysis and design. The code followed a typical OO development cycle, including user requirements definition through use-case analysis, development of a class model, and dynamic state modeling prior to implementation. The application is a real-time traffic light control system for a 4-way intersection, including a graphical simulation of the intersection operation. This includes control of the lights and a simulation of car traffic through the intersection.

For the purpose of counting "Source Lines of Code" (SLOC) in a procedure, we use the definition of a logical source statement as defined in the COCOMOII.2000 model [9]. Using this definition, the entire legacy program contains 740 SLOC, broken down into 6 classes and 29 procedures as shown in the tables below. Note that these tables represent the code properties prior to any restructuring. The SLOC count was performed manually by the programmer.

Class	SLOC	% of Total	No. of Proc.	Avg. Proc. Size
UserInterface	411	55.5%	4	103
SignalController	164	22.2%	7	23
Direction	95	12.8%	6	16
Light	23	3.1%	5	5
TurnArrow	23	3.1%	2	12
WholeNumberField	24	3.2%	5	5
TOTAL	740	100.0%	29	26

Table 2 – Traffic Light Simulation Class Summary

Proc. No.	Class_Name.Proc_Name	SLOC	Cum.	% of Total
1	UserInterface	306	306	41.4%
2	UserInterface.actionPerformed	80	386	10.8%
3	UserInterface.updateTime	18	404	2.4%
4	UserInterface.main	7	411	0.9%
5	SignalController	27	438	3.6%
6	SignalController.restart	3	441	0.4%
7	SignalController.pause	1	442	0.1%
8	SignalController.getAction	3	445	0.4%
9	SignalController.executeAction	15	460	2.0%
10	SignalController.setDirectionStates	49	509	6.6%
11	SignalController.run	66	575	8.9%
12	Direction	29	604	3.9%
13	Direction.Initialize	7	611	0.9%
14	Direction.setState	18	629	2.4%
15	Direction.setSubState	18	647	2.4%
16	Direction.getState	1	648	0.1%
17	Direction.updateLaneStatus	22	670	3.0%
18	Light	9	679	1.2%
19	Light.turnOn	3	682	0.4%
20	Light.turnOff	3	685	0.4%
21	Light.getPreferredSize	1	686	0.1%
22	Light.paintComponent	7	693	0.9%
23	TurnArrow	3	696	0.4%
24	TurnArrow.paintComponent	20	716	2.7%
25	WholeNumberField	7	723	0.9%
26	WholeNumberField.getValue	5	728	0.7%
27	WholeNumberField.setValue	1	729	0.1%
28	WholeNumberField.createDefaultModel	1	730	0.1%
29	WholeNumberDocument.insertString	10	740	1.4%

Table 3 – Traffic Light Simulation Procedure Summary

3.3.2 Refactoring Opportunities

The Refactoring text by Fowler [1] provides guidelines for identifying code problems that warrant restructuring – he refers to these problems as typical "bad smells in code". In order to identify opportunities for refactoring in the Traffic Light Simulation code, we manually inspected the code based on the problem definitions in [1]. A brief definition of these "bad smells" appears below, summarized from Chapter 3 of [1]:

Duplicated Code: the same code structure appears in more than one place.

Long Method: methods containing higher than average lines of code, including many parameters, temporary variables, loops, and conditional structures.

Large Class: a class with too many instance variables (i.e. a class that is trying to do too much).

Long Parameter List: too much data passed to a method in parameters (instead only enough information should be passed to the method so it can retrieve other data it needs).

Divergent Change: evidence that a class has been changed in distinctly different ways for different reasons (i.e. the class should likely be split up).

Shotgun Surgery: for a particular kind of maintenance change, many classes are impacted (this is the opposite of Divergent Change).

Feature Envy: methods that are more interested in external classes than the class they are in (usually with respect to another class's data).

Data Clumps: small groups of data items that appear together in multiple places within the code (could be candidates for new classes).

Primitive Obsession: the overuse of primitive types when small objects should be developed.

Switch Statements: the presence of switch (or case) statement blocks.

Parallel Inheritance Hierarchies: every time a subclass of one class is created, a subclass of another class must also be created.

Lazy Class: a class that is not doing enough to justify its existence.

Speculative Generality: code containing all sorts of hooks or special cases to handle things that aren't required.

Temporary Field: an object with an instance variable that is set up in only certain circumstances.

Message Chains: when an object must go through many other objects (e.g. a cascading series of "get" operations) to get the information it needs.

Middle Man: too much delegation in the code.

Inappropriate Intimacy: classes that have too much detailed knowledge of each other.

Alternative Classes with Different Interfaces: methods with different signatures that do the same thing.

Incomplete Library Class: library classes that do not contain enough functionality for your immediate purposes.

Data Class: classes with accessible fields (through get/set methods), but very little real behaviour.

Refused Bequest: subclasses that don't need some of the data or methods inherited from the super class.

Comments: comments can sometimes indicate that a particular piece of code is not clear and should be refactored to make it's purpose more obvious.

The following table provides the resulting list of weak areas in the code according to these definitions.

No.	Class	Start Line No.	End Line No.	Problem
1	UserInterface	1	306	Long Method
2	UserInterface	69	131	Duplicated Code, Feature Envy
3	UserInterface	312	342	Duplicated Code
4	UserInterface	307	386	Long Method
5	SignalController	35	49	Switch Statement
6	SignalController	51	98	Switch Statement, Duplicated Code, Feature Envy
7	SignalController	106	161	Long Method, Switch Statement, Duplicated Code
8	Direction	139	152	Switch Statement
9	Direction	173	186	Switch Statement
10	Direction	197	255	Long Method

Table 4 – Traffic Light Simulation Refactoring Opportunities

3.3.3 Refactoring Plan

In response to the list of opportunities for code improvement, we constructed a set of specific refactorings (from the master list of standard refactorings in [1]) in order to remove some of the problems. Note that we did not address the issue of "switch statements" from the list above. The following table details the proposed restructuring to be applied to the baseline Traffic Light Simulation code. Note that in the Modified Code column, a negative number indicates code that is removed from the original procedure as a result of the refactoring.

Class.Proc Name	New Code	Modified Code	Proposed Refactoring	Refactoring Description
UserInterface	4	0	Extract Method	Add new proc: UserInterface.addLightsToPanel
	16	-64		Remove duplicate code (68-131) and replace with calls to new proc
UserInterface	9	0	Extract Method	Add new proc: UserInterface.addDirectionLabel
	4	-26		Remove duplicate code (132-157) and replace with calls to new proc
UserInterface	10	0	Extract Method	Add new proc: UserInterface.addTimeLabel
	1	-10		Remove code from main proc (158-167) and add call statement for new proc
UserInterface	27	0	Extract Method	Add new proc: UserInterface.createSimControlPanel
	1	-27		Remove code from main proc (173-197) and add call statement for new proc
UserInterface	81	0	Extract Method	Add new proc: UserInterface.createCarInitPanel
	1	-81		Remove code from main proc (199-279) and add call statement for new proc
UserInterface	17	0	Extract Method	Add new proc: UserInterface.setPanelPositions
	1	-17		Remove code from main proc (286-302) and add call statement for new proc
UserInterface.actionPerformed	9	0	Extract Method	Add new proc: UserInterface.initDirectionState
	4	-28		Remove code from main proc (312-318, 320-326, 328-334, 336-342) and add call statements for new proc
SignalController.setDirectionStates	13	0	Move Method	Add new proc: Direction.switchStates
	4	-49		Remove code from main proc (51-98) and add call statements for new proc
SignalController.run	14	0	Extract Method	Add new proc: SignalController.triggerCycleActions
	4	-56		Remove code from main proc (106-161) and add call statements for new proc

Table 5 – Traffic Light Simulation Refactoring Plan

3.3.4 Summary of Design Refactoring Impact

Based on the refactoring plan defined above, a detailed code analysis was performed to predict the new structure of the software after implementation of the plan (the analysis involved pseudo-coding the changes and new interfaces, but not compiling). The table below summarizes the top-level structural changes, including the overall code size, number of procedures, and average procedure size within each class before and after restructuring.

	SLOC			No. of Proc.			Avg. Proc. Size		
Class Name	Before	After	Change	Before	After	Change	Before	After	Change
UserInterface	411	343	-17%	4	11	175%	103	31	-70%
SignalController	164	81	-51%	7	8	14%	23	10	-57%
Direction	95	108	14%	6	7	17%	16	15	-3%
Light	23	23	0%	5	5	0%	5	5	0%
TurnArrow	23	23	0%	2	2	0%	12	12	0%
WholeNumberField	24	24	0%	5	5	0%	5	5	0%
TOTAL	740	602	-19%	29	38	31%	26	16	-38%

Table 6 – Summary of Refactoring Impact

The key predicted results in this table regarding the proposed design changes for the Traffic Light Simulation application are:

- Overall code size would be reduced by 19%,
- The total number of procedures in the system would increase by 31%, and
- The average procedure size would be reduced by 38%.

The table below shows the predicted procedure-level structure of the code after refactoring. Note that procedures 1-29 represent the same set of procedures in the original code, while procedures 30-38 are the new procedures planned in the refactored code.

Proc. No.	Class_Name.Proc_Name	SLOC	Cum.	% of Total
1	UserInterface	105	105	17.4%
2	UserInterface.actionPerformed	56	161	9.3%
3	UserInterface.updateTime	18	179	3.0%
4	UserInterface.main	7	186	1.2%
5	SignalController	27	213	4.5%
6	SignalController.restart	3	216	0.5%
7	SignalController.pause	1	217	0.2%
8	SignalController.getAction	3	220	0.5%
9	SignalController.executeAction	15	235	2.5%
10	SignalController.setDirectionStates	4	239	0.7%
11	SignalController.run	14	253	2.3%
12	Direction	29	282	4.8%
13	Direction.Initialize	7	289	1.2%
14	Direction.setState	18	307	3.0%
15	Direction.setSubState	18	325	3.0%
16	Direction.getState	1	326	0.2%
17	Direction.updateLaneStatus	22	348	3.7%
18	Light	9	357	1.5%
19	Light.turnOn	3	360	0.5%
20	Light.turnOff	3	363	0.5%
21	Light.getPreferredSize	1	364	0.2%
22	Light.paintComponent	7	371	1.2%
23	TurnArrow	3	374	0.5%
24	TurnArrow.paintComponent	20	394	3.3%
25	WholeNumberField	7	401	1.2%
26	WholeNumberField.getValue	5	406	0.8%
27	WholeNumberField.setValue	1	407	0.2%
28	WholeNumberField.createDefaultModel	1	408	0.2%
29	WholeNumberDocument.insertString	10	418	1.7%
30	UserInterface.addLightsToPanel	4	422	0.7%
31	UserInterface.addDirectionLabel	9	431	1.5%
32	UserInterface.addTimeLabel	10	441	1.7%
33	UserInterface.createSimControlPanel	27	468	4.5%
34	UserInterface.createCarInitPanel	81	549	13.5%
35	UserInterface.setPanelPositions	17	566	2.8%
36	UserInterface.initDirectionState	9	575	1.5%
37	Direction.switchStates	13	588	2.2%
38	SignalController.triggerCycleActions	14	602	2.3%

Table 7 – Procedure Summary After Refactoring

3.3.5 Dependency Graphs Before Refactoring

This section includes the data and control dependency graphs for the Traffic Light Simulation code prior to refactoring. In addition, the overall dependency graph (logical "OR" of the data graph and the transposed control graph) is presented.

Each graph represents a matrix of size (N, N), where N is the number of procedures in the system. A "1" in entry (r, c) of the matrix indicates that procedure "r" depends on procedure "c" (represented by a symbol in the chart). The row and column numbers in the graphs refer to the procedure numbers in the tables presented earlier.

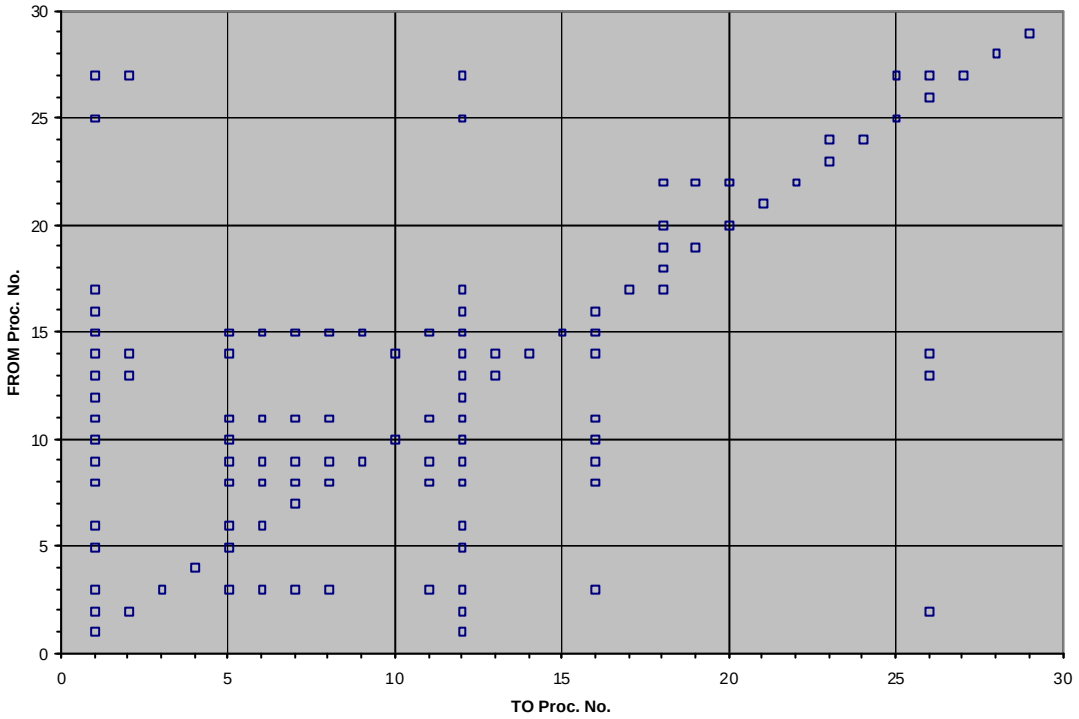

Figure 2 – Data Dependency Graph Before Refactoring

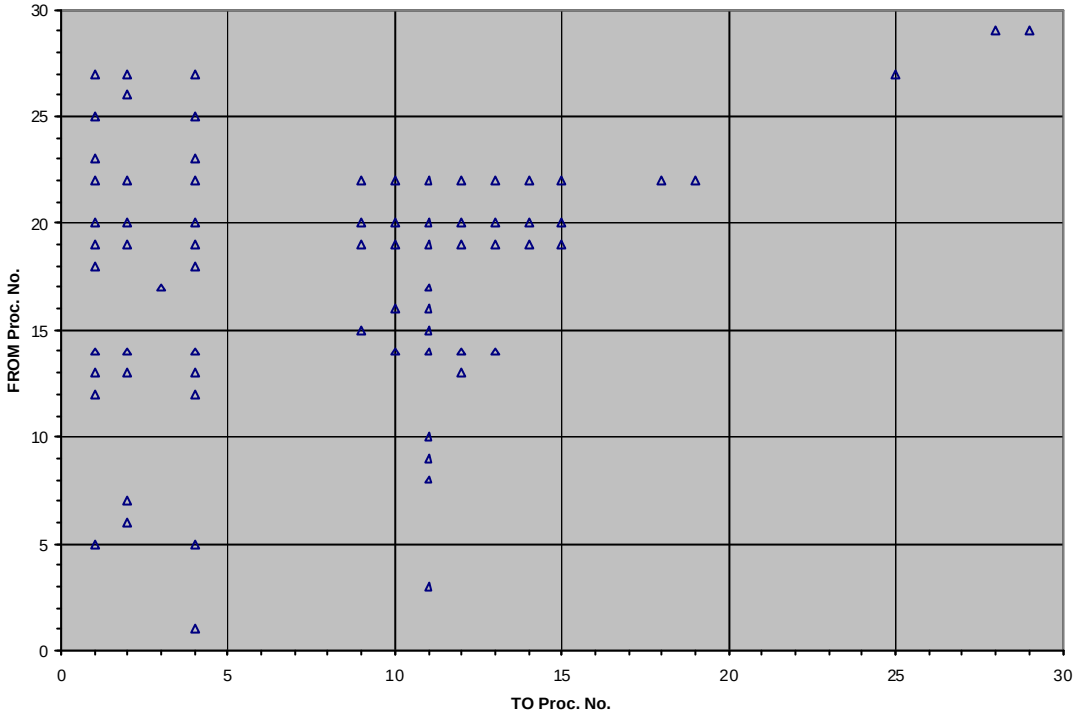

Figure 3 – Control Dependency Graph Before Refactoring

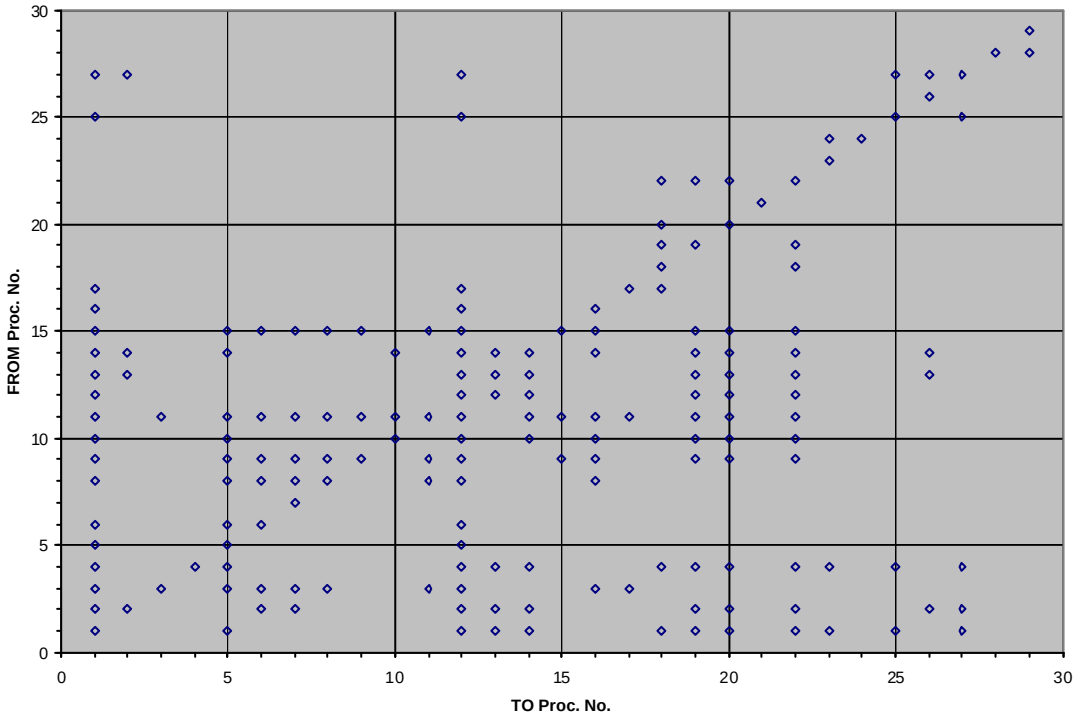

Figure 4 – Combined Dependency Graph Before Refactoring

3.3.6 Predicted Dependency Graphs After Refactoring

This section shows the predicted dependency graphs for the restructured system.

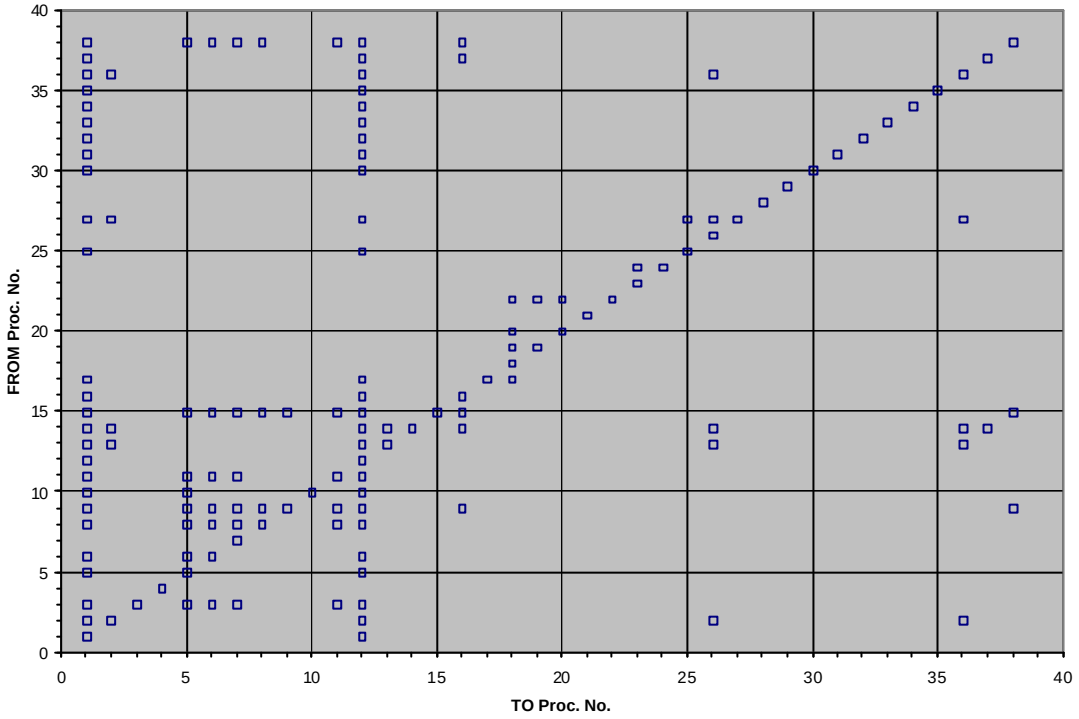

Figure 5 – Data Dependency Graph After Refactoring

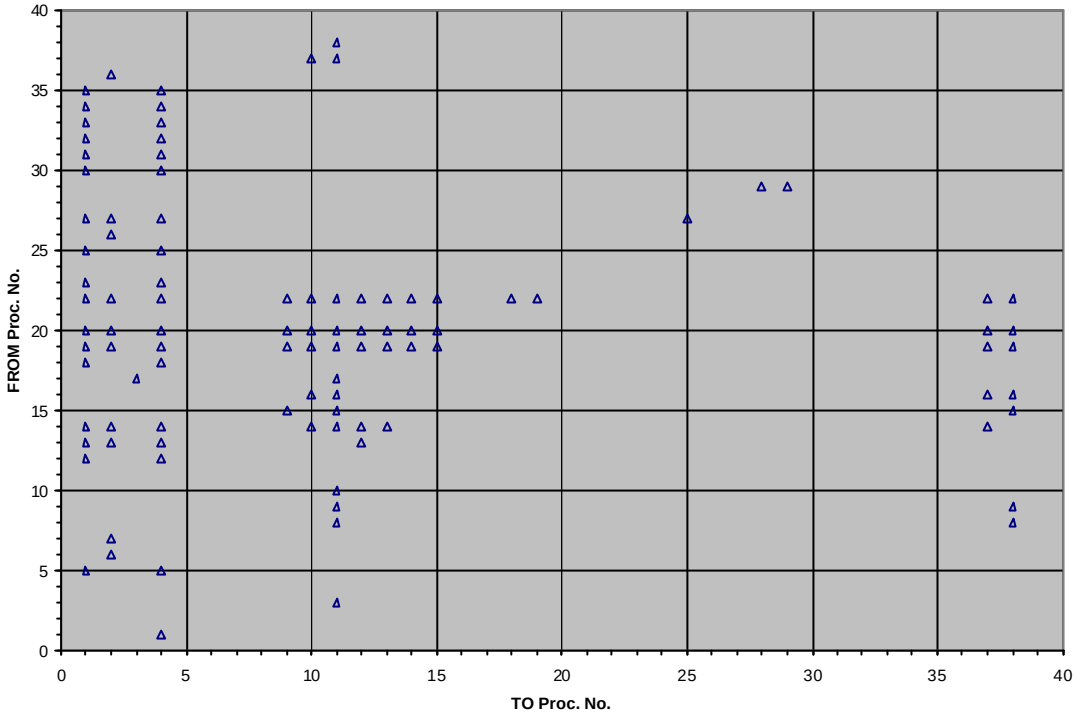

Figure 6 – Control Dependency Graph After Refactoring

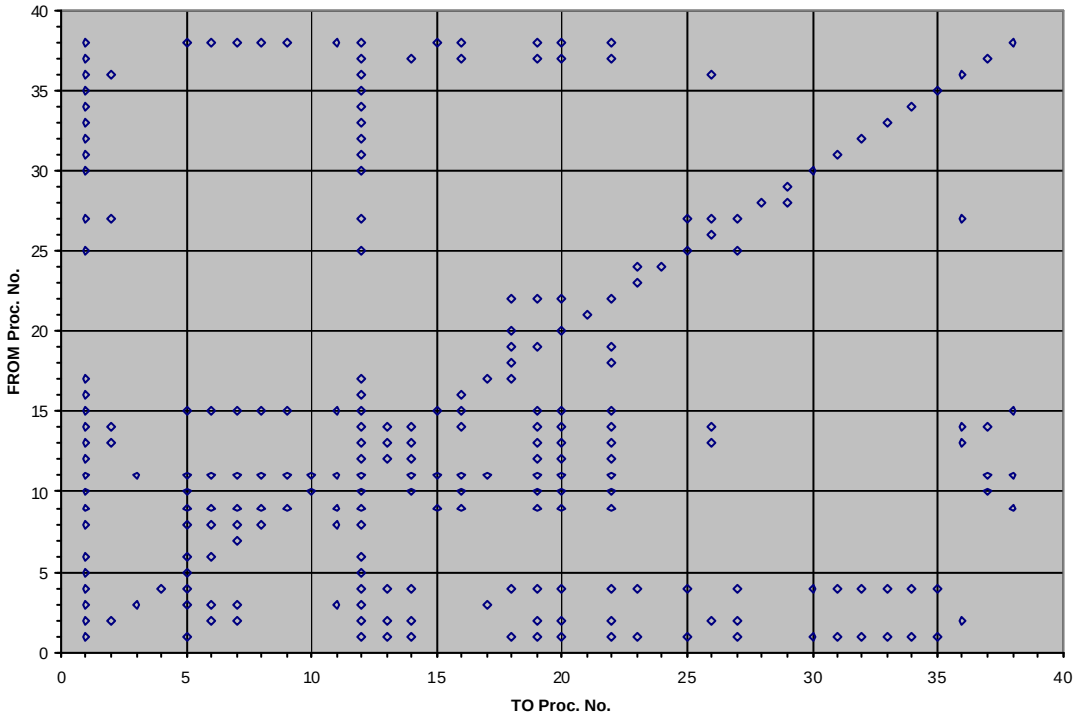

Figure 7 – Combined Dependency Graph After Refactoring

3.3.7 Dependency Graph Comparison

The table below provides a comparison between the dependency graphs before and after refactoring, measuring the number of dependency paths shown in each graph. This result shows that the density of dependency paths in the restructured graphs is lowerthan for the original design.

	BEFORE		AFTER		
Graph	**No. of Dependencies**	**Matrix Fill Ratio**	**No. of Dependencies**	**Matrix Fill Ratio**	**% Change**
Data Dependency Graph	112	13.3%	147	10.2%	-23.6%
Control Dependency Graph	73	8.7%	101	7.0%	-19.4%
Overall Dependency Graph	179	21.3%	241	16.7%	-21.6%

Table 8 – Dependency Graph Comparison

3.3.8 Code Re-test Impact Before Refactoring

The following table shows the change impact (as measured in lines of code to be re-tested) as a result of a maintenance activity striking each procedure in the original design. The table includes a probability calculated directly as the size of the procedure as a percentage of the overall system lines of code – this assumes that any line of code in the procedure could be hit by a maintenance activity with equal probability. As well, the table assumes that a particular maintenance activity will strike only one procedure directly. The re-test impact of the procedure is assumed to be all the lines of code in the procedure itself plus all the lines of code that depend on the modified procedure (according to the combined dependency graph in Figure 4). Taking an example from Figure 4, if procedure #15 is struck by a maintenance activity, then we determine from the graph that all the code from procedures 9, 11, and 15 must be re-tested. The probability of procedure #15 being hit by a maintenance event is the proportion of its code size to the overall application size (2.4% from Table 9). The contribution towards the mean system re-test impact is the sum of the procedure's impact multiplied by the probability of occurrence. The contributions from all procedures are then summed to provide the mean re-test impact for the system.

Proc. No.	Class_Name.Proc_Name	Re-Test Impact (SLOC)	Probability	Contrib. to Mean
1	UserInterface	677	41.4%	279.9
2	UserInterface.actionPerformed	106	10.8%	11.5
3	UserInterface.updateTime	84	2.4%	2.0
4	UserInterface.main	7	0.9%	0.1
5	SignalController	530	3.6%	19.3
6	SignalController.restart	203	0.4%	0.8
7	SignalController.pause	201	0.1%	0.3
8	SignalController.getAction	120	0.4%	0.5
9	SignalController.executeAction	99	2.0%	2.0
10	SignalController.setDirectionStates	133	6.6%	8.8
11	SignalController.run	120	8.9%	10.7
12	Direction	677	3.9%	26.5
13	Direction.Initialize	447	0.9%	4.2
14	Direction.setState	562	2.4%	13.7
15	Direction.setSubState	99	2.4%	2.4
16	Direction.getState	188	0.1%	0.3
17	Direction.updateLaneStatus	106	3.0%	3.2
18	Light	357	1.2%	4.3
19	Light.turnOn	605	0.4%	2.5
20	Light.turnOff	605	0.4%	2.5
21	Light.getPreferredSize	1	0.1%	0.0
22	Light.paintComponent	614	0.9%	5.8
23	TurnArrow	336	0.4%	1.4
24	TurnArrow.paintComponent	20	2.7%	0.5
25	WholeNumberField	321	0.9%	3.0
26	WholeNumberField.getValue	111	0.7%	0.8
27	WholeNumberField.setValue	401	0.1%	0.5
28	WholeNumberField.createDefaultModel	1	0.1%	0.0
29	WholeNumberDocument.insertString	11	1.4%	0.1
	MEAN RE-TEST IMPACT (SLOC):			408

Table 9 – Code Re-test Impact Before Refactoring

3.3.9 Code Re-test Impact After Refactoring

The following table shows the change impact at the procedure level for the restructured design.

Proc. No.	Class_Name.Proc_Name	Re-Test Impact (SLOC)	Probability	Contrib. to Mean
1	UserInterface	539	17.4%	94.0
2	UserInterface.actionPerformed	91	9.3%	8.5
3	UserInterface.updateTime	32	3.0%	1.0
4	UserInterface.main	7	1.2%	0.1
5	SignalController	228	4.5%	10.2
6	SignalController.restart	141	0.5%	0.7
7	SignalController.pause	139	0.2%	0.2
8	SignalController.getAction	64	0.5%	0.3
9	SignalController.executeAction	61	2.5%	1.5
10	SignalController.setDirectionStates	18	0.7%	0.1
11	SignalController.run	82	2.3%	1.9
12	Direction	539	4.8%	26.0
13	Direction.Initialize	222	1.2%	2.6
14	Direction.setState	253	3.0%	7.6
15	Direction.setSubState	61	3.0%	1.8
16	Direction.getState	97	0.2%	0.2
17	Direction.updateLaneStatus	54	3.7%	2.0
18	Light	156	1.5%	2.3
19	Light.turnOn	310	0.5%	1.5
20	Light.turnOff	310	0.5%	1.5
21	Light.getPreferredSize	1	0.2%	0.0
22	Light.paintComponent	319	1.2%	3.7
23	TurnArrow	135	0.5%	0.7
24	TurnArrow.paintComponent	20	3.3%	0.7
25	WholeNumberField	120	1.2%	1.4
26	WholeNumberField.getValue	96	0.8%	0.8
27	WholeNumberField.setValue	176	0.2%	0.3
28	WholeNumberField.createDefaultModel	1	0.2%	0.0
29	WholeNumberDocument.insertString	11	1.7%	0.2
30	UserInterface.addLightsToPanel	116	0.7%	0.8
31	UserInterface.addDirectionLabel	121	1.5%	1.8
32	UserInterface.addTimeLabel	122	1.7%	2.0
33	UserInterface.createSimControlPanel	139	4.5%	6.2
34	UserInterface.createCarInitPanel	193	13.5%	26.0
35	UserInterface.setPanelPositions	129	2.8%	3.6
36	UserInterface.initDirectionState	91	1.5%	1.4
37	Direction.switchStates	49	2.2%	1.1
38	SignalController.triggerCycleActions	61	2.3%	1.4
	MEAN RE-TEST IMPACT (SLOC):			216

Table 10 – Code Re-test Impact After Refactoring

3.3.10 Summary of Dependency Analysis Results

This section illustrates the probability distributions of the regression testing impact for the Traffic Light Simulation code before and after refactoring, assuming a random maintenance activity as defined in section 3.

In Figure 8, Figure 10, and Figure 12, each data point represents the re-test impact of a single procedure versus the probability of that impact occurring for a random

maintenance event. Note that the impact data is expressed as a percentage of the total SLOC in the system rather than as an absolute SLOC number. For the combined graph in Figure 12, the code size reference is the original, unchanged version of the Traffic Light Simulation code.

Figure 9 and Figure 11 illustrate the distribution of re-test impact with respect to the total number of procedures in the system. Note that in these two distribution charts, the data has been sorted from the highest impact procedure to the lowest so that the procedure numbers in the graphs do not correspond to the absolute procedure identifiers.

When comparing the two distributions in Figure 12, we note that the refactored ("After") design shows reduced peaks in terms of maximum probability as well as maximum impact compared with the orginal ("Before") design. The entire distribution in the After case appears to be shifted down and to the left of the original distribution.

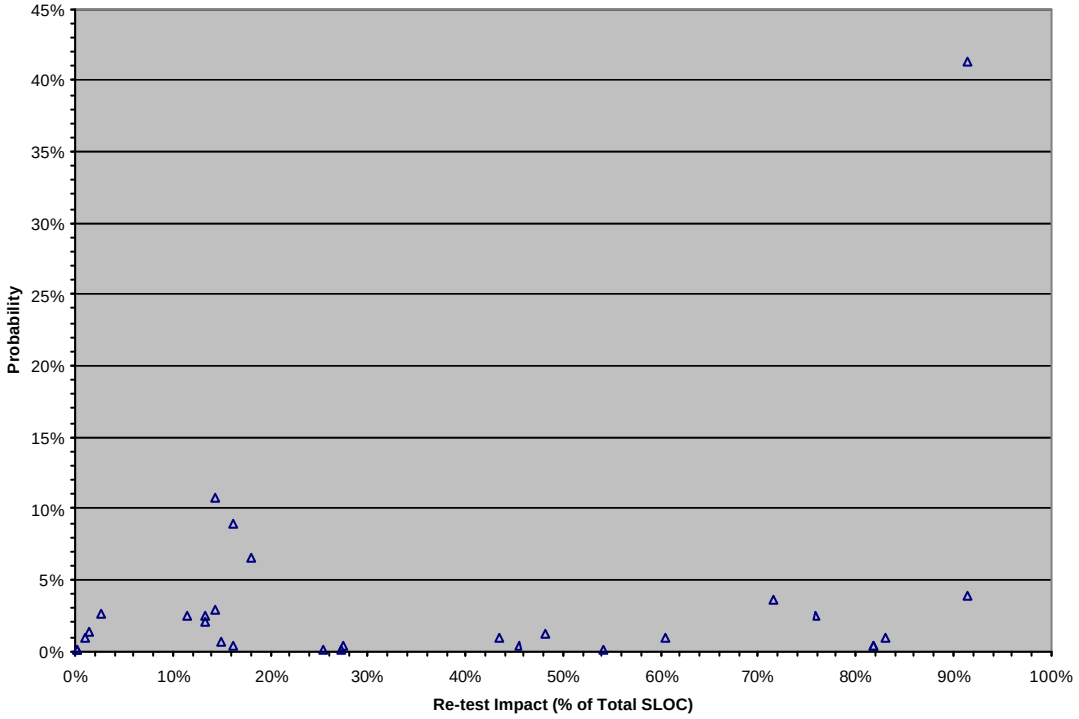

Figure 8 – Re-test Impact By Procedure Before Refactoring

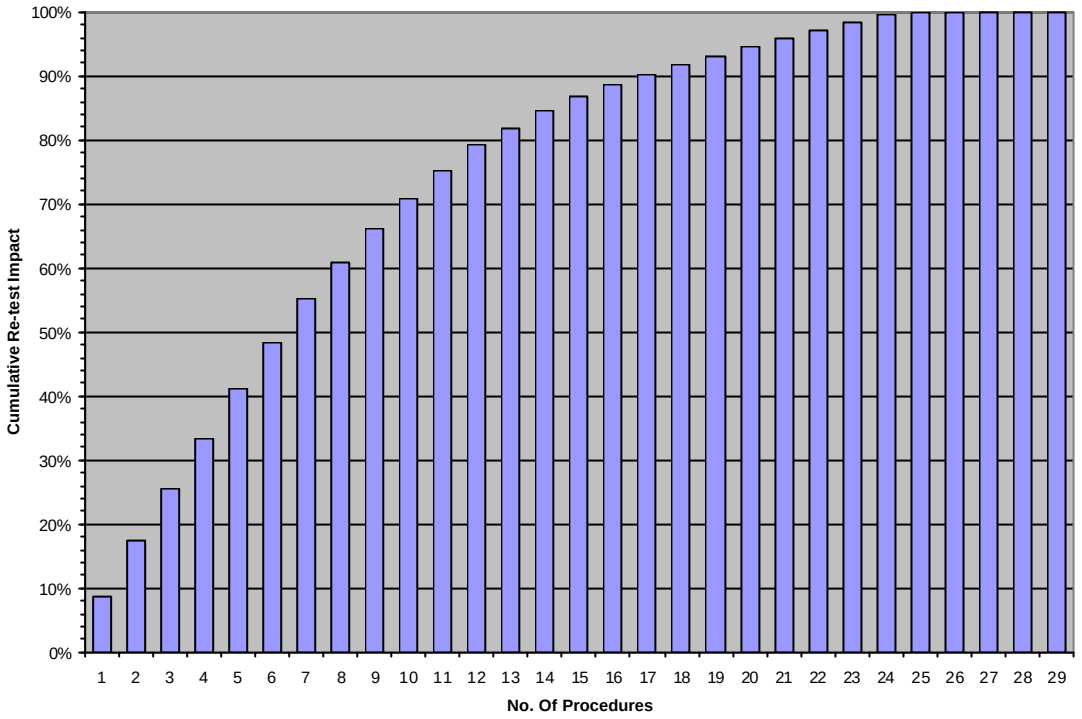

Figure 9 – Cumulative Re-test Impact Before Refactoring

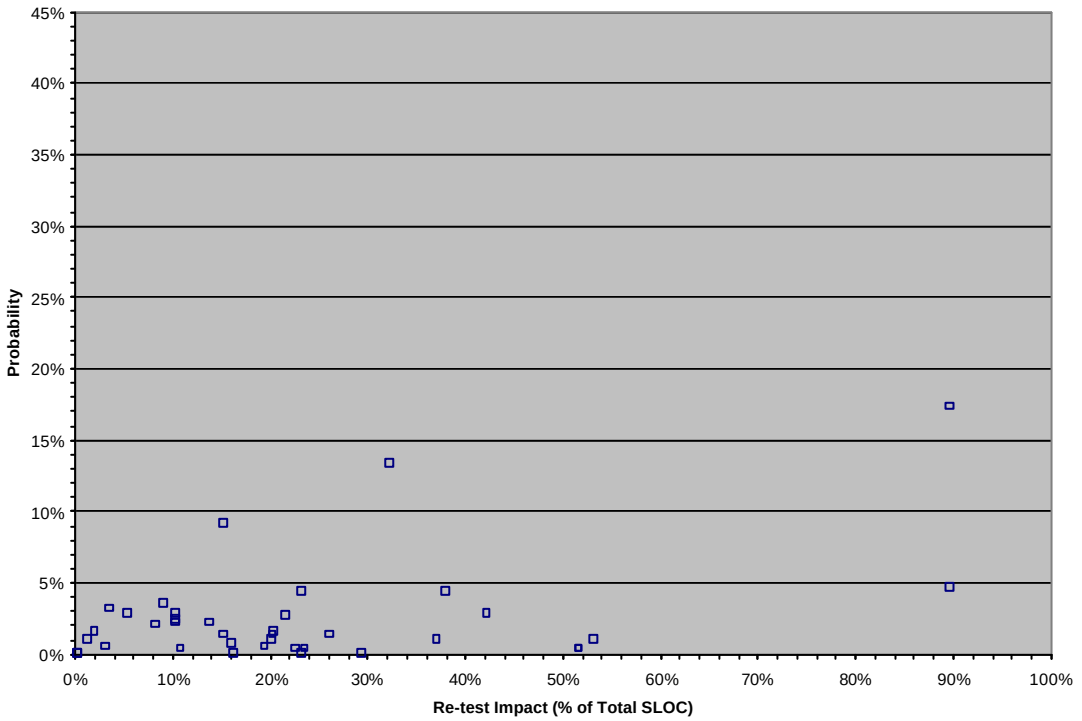

Figure 10 – Re-test Impact By Procedure After Refactoring

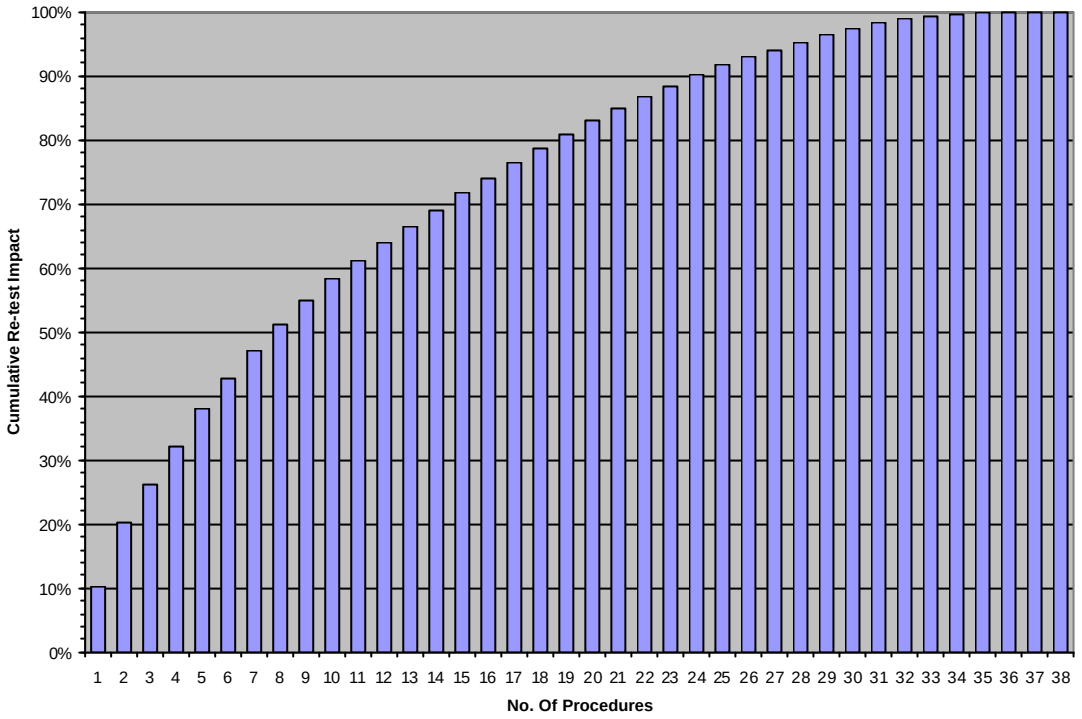

Figure 11 – Cumulative Code Re-test Impact After Refactoring

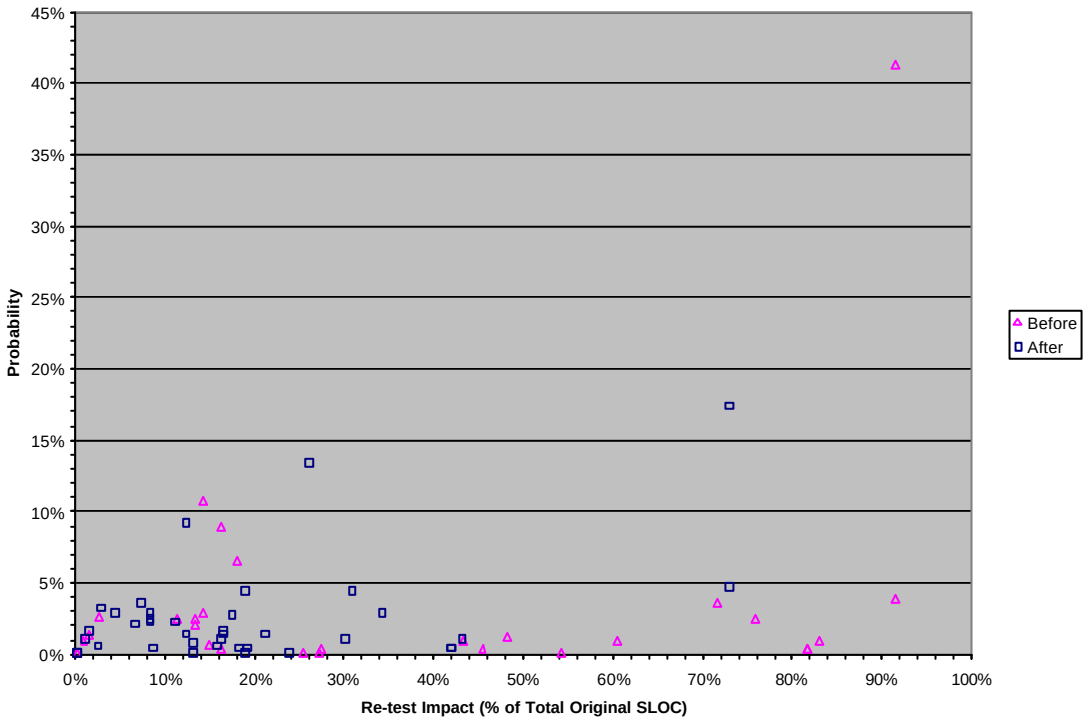

Figure 12 – Re-test Impact Comparison

As a direct result of the planned restructuring activities, we predict that the mean re-test impact for the Traffic Light Simulation software will decrease from 408 SLOC to 216 SLOC. This is a potential reduction of 47.1% in the average regression testing effort for a random maintenance activity.

A downside of this method is that it is conservative when estimating the code re-test impact. For example, consider the large UserInterface procedure in the original Traffic Light Simulation system design. As shown in Figure 4, 18 out of 29 procedures in the system depend on UserInterface. This procedure has many lines of code (306) – it is clear that not every line of code is linked to all the external dependencies listed in Table 9. Some lines of code may in fact have no external dependencies whatsoever, meaning that a change to that line of code does not require the substantial re-testing assumed for this large procedure. So while the potential exists for a large impact whenever UserInterface is modified, not every modification will result in the maximum amount of regression testing.

One solution to this problem is to perform line-by-line dependency analysis within the program to determine the "micro" dependencies beneath the procedure level. Due to the large amount of data involved, this is not practical to analyze for a large system – this approach was not investigated further.

However, even if we assume the wildly optimistic case that there are no external dependencies in any of the procedures and only the immediately impacted procedure must be re-tested, then the average re-test impact becomes 150 SLOC for the original design and 44 SLOC for the refactored design. This is a 71 % improvement, an even better result for the restructuring activity. It is clear that the regression testing effort will be less for the redesigned system.

3.3.11 Maintenance Effort Estimation

Now that we have established a method to quantify how much code must be regression tested for a maintenance activity, we must determine how to convert this benefit into an effort estimate. The refactoring cost-benefit equation components use person-effort units.

One approach is to use the COCOMOII.2000 model (introduced earlier in section 2). This model contains a method for estimating the effort associated with adapting a system (i.e. modifying part of a system taking advantage of a re-use opportunity). We can think of each maintenance activity as an exercise in software adaptation – when refactoring we are changing a relatively small part of the system, usually for perfective or corrective reasons. The COCOMOII.2000 re-use model input parameters include a special variable called "IM", the "Percentage of Integration Required for Adapted Software". This is essentially the amount of testing that must be done for the adapted code as compared to redeveloping the whole application. We can exploit the IM parameter to predict the

maintenance effort based on the regression testing impact (in SLOC) determined from the previous section.

3.3.12 Traffic Light Simulation Before Refactoring

This section presents a number of sample calculations of maintenance activity effort using the COCOMOII.2000 re-use model defined in [9] (primarily using the model equations defined in sections 2.2.4.2 and 2.3 of [9]). In the tables below we show the key input values used in the model – most of the numbers are hypothetical assumptions that would be replaced with local environment knowledge if applied to an industrial project. For example, in the effort adjustment factors (Table 11) we have assumed the "Nominal" value for each parameter.

The driving variables in our sample calculation are DM, CM, and IM from Table 14. These variables represent the amount of change occurring in the system during the maintenance activity. Because we don't know the exact size of our hypothetical maintenance activities, we have assumed a value of 5% for DM and CM (this is approximately 30-40 SLOC per activity for the Traffic Light Simulation system). IM is taken as the average percentage of the total code that must be re-tested as a result of the change, as determined from our analysis of the previous section. The data in these tables are used to configure the COCOMOII.2000 model.

Param.	Very Low	Low	Nominal	High	Very High	Extra High	SCORE
RELY	0.82	0.92	1.00	1.10	1.26	N/A	1.00
DATA	N/A	0.90	1.00	1.14	1.28	N/A	1.00
CPLX	0.73	0.87	1.00	1.17	1.34	1.74	1.00
RUSE	N/A	0.95	1.00	1.07	1.15	1.24	1.00
DOCU	0.81	0.91	1.00	1.11	1.23	N/A	1.00
TIME	N/A	N/A	1.00	1.11	1.29	1.63	1.00
STOR	N/A	N/A	1.00	1.05	1.17	1.46	1.00
PVOL	N/A	0.87	1.00	1.15	1.30	N/A	1.00
ACAP	1.42	1.19	1.00	0.85	0.71	N/A	1.00
PCAP	1.34	1.15	1.00	0.88	0.76	N/A	1.00
PCON	1.29	1.12	1.00	0.90	0.81	N/A	1.00
APEX	1.22	1.10	1.00	0.88	0.81	N/A	1.00
PLEX	1.19	1.09	1.00	0.91	0.85	N/A	1.00
LTEX	1.20	1.09	1.00	0.91	0.84	N/A	1.00
TOOL	1.17	1.09	1.00	0.90	0.78	N/A	1.00
SITE	1.22	1.09	1.00	0.93	0.86	0.80	1.00
SCED	1.43	1.14	1.00	1.00	1.00	N/A	1.00
						Product:	1.00

Table 11 – Assumed COCOMOII.2000 Effort Adjustment Factors

Param.	Very Low	Low	Nominal	High	Very High	Extra High	SCORE
PREC	6.20	4.96	3.72	2.48	1.24	0.00	3.72
FLEX	5.07	4.05	3.04	2.03	1.01	0.00	3.04
RESL	7.07	5.65	4.24	2.83	1.41	0.00	4.24
TEAM	5.48	4.38	3.29	2.19	1.10	0.00	3.29
EPML	7.80	6.24	4.68	3.12	1.56	0.00	4.68
						Sum:	18.97
						Exponent	1.10

Table 12 – Assumed COCOMOII.2000 Model Scale Factors

Parameter	Very Low	Low	Nominal	High	Very High	Ext. High	SCORE
SU - Structure	50	40	30	20	10	N/A	30
SU - Application Clarity	50	40	30	20	10	N/A	30
SU - Self-Descriptiveness	50	40	30	20	10	N/A	30
	None	Low	Nominal	High	Very High	Ext. High	SCORE
AA	0	2	4	6	8	N/A	4
	Comp. Familiar	Mostly Familiar	Some. Familiar	Consid. Familiar	Mostly Unfam.	Comp. Unfamiliar	SCORE
UNFM	0.0	0.2	0.4	0.6	0.8	1.0	0.4

Table 13 – Assumed COCOMOII.2000 Re-use Model Parameter s

Code	Meaning
DM	Percent Design Modified
CM	Percent Code Modified
IM	Percent of Integration Required for Adpated Software
SU	Software Understanding Increment
AA	Assessment and Assimilation Increment
UNFM	Programmer Unfamiliarity Increment
AT	Percentage of Code Automatically Translated

Table 14 – COCOMOII.2000 Re-use Model Parameter Definitions

3.3.13 Maintenance Savings Prediction

The table below summarizes the predicted maintenance effort savings for the Traffic Light Simulation application based on the proposed design restructuring and calculations using the COCOMOII.2000 model. Note that the results were obtained from a spreadsheet tool developed for this project based on the equations defined in [9].

Model Parameters	(Before)	(After)
Code Size (KSLOC)	0.740	0.602
DM (0-100)	5.0	5.0
CM (0-100)	5.0	5.0
IM (0-100+)	55.1	35.9
Equivalent KSLOC	0.213	0.131
Effort (Person-months)	0.538	0.313
Net Benefit/Maint. Activity	0.225	

Table 15 – Maintenance Effort Savings Summary

It should be noted that the COCOMOII.2000 model (as presented in [9]) is not well calibrated for systems this small – if an organization routinely generated such systems then a calibration could be performed to improve the accuracy of the results.

3.3.14 Refactoring Effort Estimation

Just as we applied the COCOMOII.2000 re-use model to specific maintenance activities in the previous section, we can apply the same model to the refactoring effort prediction itself for the Traffic Light Simulation code. This effort prediction represents the cost side of the cost-benefit analysis.

In this particular example we assume that there is no training costs or tool purchase costs associated with the refactoring activity. In reality, if new technologies are being introduced into an organization these sorts of costs must be accounted for. However we assume that the refactoring will be manually performed (no automated tools) and that the programmers already have the necessary knowledge to restructure the program.

We summarize here the sample calculation for the three key input parameters to the model based on the refactoring plan from Table 4:

- DM = (4 modified procedures)/(29 original procedures) = 13.79%
- CM = (220 new SLOC)/(740 original SLOC) = 29.73%
- IM = 100%

Note that so much of the code is impacted by the refactoring that the dependency graph indicates the entire system must be re-tested (i.e. we have set IM = 100%). Using the same default model settings shown previously, Table 16 shows the results of the refactoring effort prediction.

Model Parameters	Cost Calc.
Code Size (KSLOC)	0.740
DM (0-100)	13.8
CM (0-100)	29.7
IM (0-100+)	100.0
Equivalent KSLOC	0.437
Effort (Person-months)	1.18

Table 16 – Refactoring Effort Prediction

3.3.15 Cost-Benefit Equation

The refactoring ROI equation was previously defined in section 2. This equation can be restated as follows:

ROI = (# of Expected Maintenance Activities)*(Effort Savings per Activity) / (Refactoring Effort)

From our previous calculations:
Effort Savings per Activity = 0.225 person-months
Refactoring Effort = 1.18 person-months
Break-even point = 1.18/0.225 = 5.27 Activities

From the above result, the ROI will be greater than one if the number of Expected Maintenance Activities is greater than or equal to six. In other words, if six or more maintenance activities (bug fixes, enhancements, etc.) occur after the design restructuring takes place, then this refactoring becomes cost-effective.

This number is sensitive to the assumed size of each maintenance activity – in our sample calculation we assumed 5% of the code size is modified per activity (represented by the DM and CM model variables). A higher number for DM and CM will increase the maintenance savings and reduce the required number of events to "break even". A lower assumed number for DM and CM means that the investment will take longer to become cost effective.

This small example shows how legacy systems can be evaluated to determine the effectiveness of strategic design enhancements.

3.3.16 Class-level Dependency Analysis

There are a number of commercial tools available that support Java code dependency analysis. However, these tools typically generate dependency data at the class level and not the procedure level. Since tool support is essential for performing a refactoring cost-benefit analysis for a large system, it is therefore important to investigate the impact of performing such an analysis at the class level.

The hypothesis is that a class-level dependency analysis should yield similar overall results as a procedure-level analysis for the same system.

In order to test this hypothesis, the Traffic Light Simulation application was re-analyzed considering dependencies only at the class level. The code base and refactoring plan were identical to that considered for the procedure-level analysis.

The class-level dependency graphs before and after system restructuring were constructed using the same technique presented for the procedure-level graphs. The result is that the class-level graph is not predicted to change as a result of the proposed refactoring plan. The class-level dependency graph is shown in the table below.

		1	2	3	4	5	6
UserInterface	1	1	1	1	1	1	1
SignalController	2	1	1	1	1	1	1
Direction	3	1	1	1	1	0	1
Light	4	0	0	0	1	0	0
TurnArrow	5	0	0	0	0	1	0
WholeNumberField	6	1	1	1	1	0	1

Table 17 – Traffic Light Simulation Class-level Dependency Graph

Even though the class-level dependency graph is not predicted to change, the class-level code statistics will definitely change as a result of the refactoring. This results in a change to the mean regression testing impact for the system.

The class-level code size statistics and the predicted mean re-test impact for a random maintenance activity are presented in the tables below.

BEFORE Refactoring	SLOC	% of Total	Re-test Impact	Contrib. to Mean
UserInterface	411	55.5%	694	385
SignalController	164	22.2%	694	154
Direction	95	12.8%	694	89
Light	23	3.1%	717	22
TurnArrow	23	3.1%	598	19
WholeNumberField	24	3.2%	694	23
TOTAL:	740			692

Table 18 – Class-level Re-test Impact Before Refactoring

AFTER Refactoring	SLOC	% of Total	Re-test Impact	Contrib. to Mean
UserInterface	343	57.0%	556	317
SignalController	81	13.5%	556	75
Direction	108	17.9%	556	100
Light	23	3.8%	579	22
TurnArrow	23	3.8%	447	17
WholeNumberField	24	4.0%	556	22
TOTAL:	602			553

Table 19 – Class-level Re-test Impact After Refactoring

The mean re-test impact after refactoring is predicted to decrease from 692 SLOC to 553 SLOC, a decrease of 20.1%. The regression testing savings are caused by a reduction in the overall code size, meaning the peak re-test impact is smaller (i.e. primarily for the UserInterface and SignalController classes).

This result differs from the procedure-level result that predicted a decrease of 47.1% in the re-test impact. The class-level analysis is clearly less sensitive to this type of restructuring than the procedure-level analysis.

Using the COCOMOII.2000 model, the anticipated effort savings per maintenance activity based on the reduced re-test impact were calculated.

	Cost	Benefit Calc.	
Model Parameters	Calc.	(Before)	(After)
Code Size (KSLOC)	0.740	0.740	0.602
DM (0-100)	13.8	5.0	5.0
CM (0-100)	29.7	5.0	5.0
IM (0-100+)	100.0	93.5	91.9
Equivalent KSLOC	0.437	0.319	0.256
Effort (Person-months)	1.18	0.84	0.66
Net Benefit/Maint. Activity		0.18	
Break-Even Point (# Activities)		6.56	

Table 20 – Predicted Effort Savings (Class-level analysis)

The resulting maintenance effort savings is 0.180 person-months per event, which is less than the 0.225 person-months per event predicted from the procedure-level analysis. Based on this revised result, it would take seven maintenance activities before the restructuring investment becomes cost-effective, as opposed to six activities determined from the procedure-level analysis.

These results indicate that the hypothesis is false, and that results obtained at the class-level are different from the results obtained at the procedure-level. This one example suggests that the class-level analysis produces more conservative results than the procedure-level analysis.

3.3.17 Trial Case Study Conclusions

The proposed refactoring is predicted to decrease the overall code size (by 19%) and increase the number of procedures in the system (by 31%). In addition, the density of dependency paths in the system is predicted to decrease by approximately 22%. This decrease in density appears to result from the introduction of new procedures into the system possessing relatively few external dependencies. These new procedures are created by extracting code from larger original procedures, therefore generating very few new dependencies.

The refactoring appears to reduce the peaks along both axes of the impact probability distribution. Compared to the original distribution, the refactored distribution appears shifted down and to the left.

Procedure-level dependency analysis predicts that the mean regression testing impact (in terms of affected SLOC) of a random maintenance activity will decrease by approximately 47% as a result of the proposed design restructuring.

Cost estimation modeling using COCOMOII.2000 suggests that the restructuring will be cost effective if six or more maintenance events occur after the refactoring investment.

The results of the procedure-level analysis are not duplicated by a class-level analysis of the same design transformations. In general, the class-level analysis yields more conservative results regarding cost-effectiveness. It appears that the class-level approach is not as sensitive to the proposed design restructuring activities.

This trial case study suggests that this methodology can be applied and can yield meaningful results. The next step is to apply this methodology to a larger, commercial-grade software system.

4 Case Study – NLIS Channel Server

4.1 Introduction

The NLIS Channel Server is a production system built by MacDonald Dettwiler and Associates (MDA). For this case study, we have selected the Settlement Agent stand-alone component of NLIS – we refer to this component as the "system", even though it represents a relatively small part of the overall application. This software is responsible for processing land information database financial transactions as part of a web-based server system. This system has been operational for the past two years, and is currently in the maintenance phase of its lifecycle.

The version of the system analyzed in this case study consists of 27 Java classes containing 227 methods and approximately 2500 SLOC. Section 7.1.1 lists the detailed internal structure (classes, methods, and SLOC) of this software.

The NLIS source code was analyzed using the methodology described in section 3, following the same approach as used for the trial case study. Manual code inspections were performed in order to determine the refactoring plan and construct the system dependency graphs. The following sections present the results of the case study, following a similar format to the results of the trial case study from section 3.3.

Regarding regression testing of the NLIS system, the project team is typically conservative in assuming that the entire system must be re-tested as a result of any maintenance to the delivered product, regardless of the dependency graph. Many other mission or business-critical project teams follow a similar philosophy. Strictly speaking, our methodology may not be suited to such projects. Alternatively, this methodology might persuade some that full system testing may not be necessary for every change (if the dependencies are well characterized). Nevertheless, we feel the results of the case study are relevant and provide a good demonstration of our methodology.

Note also that the class-level results were not computed for the NLIS case study – only the procedure-level results are presented in this report.

4.2 Refactoring Plan and Design Impact

The following tables describe the refactoring opportunities uncovered during the analysis of the NLIS code, as well as the restructuring plan designed to address these opportunities.

In addition, the tables show the predicted impact of the proposed changes to the design and code structure (in terms of classes, procedures, and SLOC).

Note that, after further review and consideration, not every opportunity identified in the list in Table 21 led to a proposed design change. Some changes were judged too difficult to be worthwhile, and some would not have had an impact on the code dependency structure. These instances are clearly marked in Table 22.

As seen in Table 24, the net effect of the proposed refactoring is to decrease the overall code size by 3.0%, while increasing the number of procedures in the system by 2.6%.

Prob. No.	Problem	Unit Name	Comment
1	Large Class	STL_Agent	11 methods, 210 SLOC
2	Large Class	STL_Database	17 methods, 267 SLOC
3	Large Class	REP_Manager_Transactions	22 methods, 430 SLOC
4	Large Class	STL_Reports	21 methods, 441 SLOC
5	Duplicated Code	STL_Agent.run	
6	Long Method	STL_Agent.run	97 SLOC
7	Switch Statement	STL_Database.getPartyList	
8	Long Method	STL_Database.updateSettlementStatus	43 SLOC
9	Duplicated Code	STL_Database.updateSettlementStatus	
10	Duplicated Code	STL_Database.getSageSrSupplierCode	Similar to getSateMapSupplierCode method.
11	Duplicated Code	STL_Database.updateEndUserEFT	
12	Duplicated Code	STL_Database.isPartyInSync	
13	Duplicated Code	STL_Database.synchronizeEndUser	
14	Duplicated Code	STL_BusinessManager.buildAUDDISPartyList	Similar to buildPartyList method.
15	Duplicated Code	REP_Manager_Transactions.getAdministratorEmail	Similar to getMapVatRate
16	Long Method	REP_Manager_Transactions.getServiceType	106 SLOC
17	Duplicated Code	REP_Manager_Transactions.getServiceType	
18	Duplicated Code	REP_Manager_Transactions.getVatableAmount	Similar structure to getNonVatableAmount method.
19	Data Class	STL_EndUser	11 fields, very little behavior (get methods only)
20	Duplicated Code	STL_EndUser.invoicedByEmail	Similar to testMode method.
21	Feature Envy	STL_EndUser.postCredit	Retrieval of data from STL_Transaction class.
22	Duplicated Code	STL_EndUser.postDebit	Similar to postCredit method.
23	Data Class	STL_Transaction	24 fields, but no behavior (just get methods)
24	Long Method	STL_Reports.generateBacsTransactionFile	60 SLOC
25	Feature Envy	STL_Reports.generateBacsTransactionFile	Uses STL_EndUser to get data.
26	Long Method	STL_Reports.generateSageTransactionFile	
27	Feature Envy	STL_Reports.generateSageTransactionFile	Uses STL_Transaction to get data.
28	Duplicated Code	STL_Reports.getUsersManagers	Similar to userHasNonManagedTransactions method.
29	Duplicated Code	STL_Reports.createInvoiceStatements	Similar to createInvoices method.
30	Duplicated Code	STL_Reports.generateReport	
31	Feature Envy	STL_Reports.transactionAlreadyCreated	Method could be part of STL_EndUser/STL_Party.

Table 21 – NLIS Refactoring Opportunities

Prob. No.	Unit Name	Planned Refactoring	Comment
1	STL_Agent	None	
2	STL_Database	None	
3	REP_Manager_Transactions	None	
4	STL_Reports	Move Method	See prob. #27 below.
5	STL_Agent.run	Extract Method	3 new methods created (new proc. 228, 229, 230).
6	STL_Agent.run	Extract Method	Same as above.
7	STL_Database.getPartyList	None	
8	STL_Database.updateSettlementStatus	Extract Method	New method created (proc. 231).
9	STL_Database.updateSettlementStatus	Extract Method	Same as above.
10	STL_Database.getSageSrSupplierCode	Extract Method	Method deleted (calls changed to proc. 21).
11	STL_Database.updateEndUserEFT	Extract Method	New method created (proc. 232).
12	STL_Database.isPartyInSync	Extract Method	New method created (proc. 232).
13	STL_Database.synchronizeEndUser	Extract Method	New method created (proc. 232).
14	STL_BusinessManager.buildAUDDISPartyList	None	
15	REP_Manager_Transactions.getAdministratorEmail	Extract Method	Method deleted (calls changed to proc. 40).
16	REP_Manager_Transactions.getServiceType	None	
17	REP_Manager_Transactions.getServiceType	None	
18	REP_Manager_Transactions.getVatableAmount	None	
19	STL_EndUser	Move Method	3 new methods added with behavior (234, 235, 236)
20	STL_EndUser.invoicedByEmail	Extract Method	Same as above.
21	STL_EndUser.postCredit	None	
22	STL_EndUser.postDebit	Extract Method	Method deleted (calls changed to proc. 87).
23	STL_Transaction	Move Method	8 methods moved from STL_Reports (176 to 184).
24	STL_Reports.generateBacsTransactionFile	Extract Method	Created 2 new methods in STL_EndUser (234, 236)
25	STL_Reports.generateBacsTransactionFile	Move Method	Same as above.
26	STL_Reports.generateSageTransactionFile	Extract Method	Moved to STL_Transaction (new proc. 237).
27	STL_Reports.generateSageTransactionFile	Move Method	Same as above.
28	STL_Reports.getUsersManagers	None	
29	STL_Reports.createInvoiceStatements	Extract Method	Method deleted (calls changed to proc. 172).
30	STL_Reports.generateReport	Extract Method	New method created (proc. 233).
31	STL_Reports.transactionAlreadyCreated	Move Method	Moved to STL_EndUser (new proc. 235).

Table 22 – NLIS Refactoring Plan

Proc.		Code Changes (SLOC)				
No.	Proc. Name	Added	Modified	Deleted	Net	Sum
2	STL_Agent.run	7	0	36	-29	43
20	STL_Database.updateSettlementStatus	2	0	16	-14	18
22	STL_Database.getSageSrSupplierCode	0	0	8	-8	8
24	STL_Database.updateEnduserEFT	1	0	6	-5	7
25	STL_Database.isPartyInSync	1	0	4	-3	5
26	STL_Database.synchronizeEndUser	1	0	4	-3	5
28	STL_Database.getSystemTableParameter	1	0	3	-2	4
41	REP_ManagerTransactions.getAdministratorEmail	0	0	10	-10	10
85	STL_EndUser.invoicedByEmail	0	0	3	-3	3
87	STL_EndUser.postCredit	0	6	0	0	6
88	STL_EndUser.postDebit	0	0	6	-6	6
167	STL_Reports.generateBacsTransactionFile	2	0	40	-38	42
168	STL_Reports.generateSageTransactionFile	1	0	22	-21	23
172	STL_Reports.createInvoices	2	0	0	2	2
173	STL_Reports.createInvoiceStatements	0	0	22	-22	22
174	STL_Reports.generateReport	2	0	20	-18	22
186	STL_Reports.transactionAlreadyCreated	0	0	12	-12	12
228	STL_Agent.proc228	9	0	0	9	9
229	STL_Agent.proc229	7	0	0	7	7
230	STL_Agent.proc230	2	0	0	2	2
231	STL_Database.proc231	8	0	0	8	8
232	STL_Database.proc232	7	0	0	7	7
233	STL_Reports.proc233	11	0	0	11	11
234	STL_EndUser.proc234	20	0	0	20	20
235	STL_EndUser.proc235	12	0	0	12	12
236	STL_EndUser.proc236	20	0	0	20	20
237	STL_Transaction.proc237	22	0	0	22	22
	TOTAL:	138	6	212	-74	356

Table 23 – NLIS Predicted Code Changes

	SLOC			No. of Proc.			Avg. Proc. Size		
Class Name	Before	After	% Change	Before	After	% Change	Before	After	% Change
REP_Constants	143	143	0.0%	1	1	0.0%	143	143	0.0%
REP_ManagerTransactions	430	420	-2.3%	22	21	-4.5%	20	20	2.3%
RPT_PrintDate	23	23	0.0%	2	2	0.0%	12	12	0.0%
STL_Agent	210	199	-5.2%	11	14	27.3%	19	14	-25.5%
STL_BusinessManager	131	131	0.0%	8	8	0.0%	16	16	0.0%
STL_Constants	123	123	0.0%	1	1	0.0%	123	123	0.0%
STL_Database	267	247	-7.5%	17	19	11.8%	16	13	-17.2%
STL_DatabaseInterface	4	4	0.0%	16	16	0.0%	0	0	0.0%
STL_Day	14	14	0.0%	4	4	0.0%	4	4	0.0%
STL_EndUser	51	94	84.3%	14	16	14.3%	4	6	61.3%
STL_EndUserFactory	19	19	0.0%	2	2	0.0%	10	10	0.0%
STL_Party	69	69	0.0%	26	26	0.0%	3	3	0.0%
STL_PartyException	1	1	0.0%	1	1	0.0%	1	1	0.0%
STL_Reports	441	190	-56.9%	21	11	-47.6%	21	17	-17.7%
STL_Transaction	85	260	205.9%	27	37	37.0%	3	7	123.2%
STL_TransactionException	1	1	0.0%	1	1	0.0%	1	1	0.0%
STL_TransactionFactory	38	38	0.0%	2	2	0.0%	19	19	0.0%
STL_Util	48	48	0.0%	9	9	0.0%	5	5	0.0%
UTL_Constant	38	38	0.0%	1	1	0.0%	38	38	0.0%
UTL_CSVFormattedFile	9	9	0.0%	2	2	0.0%	5	5	0.0%
UTL_DBConnection	85	85	0.0%	11	11	0.0%	8	8	0.0%
UTL_DBConstant	53	53	0.0%	1	1	0.0%	53	53	0.0%
UTL_FCFFormattedFile	17	17	0.0%	2	2	0.0%	9	9	0.0%
UTL_FormattedFile	11	11	0.0%	3	3	0.0%	4	4	0.0%
UTL_Logger	45	45	0.0%	8	8	0.0%	6	6	0.0%
UTL_Mail	52	52	0.0%	8	8	0.0%	7	7	0.0%
UTL_Property	58	58	0.0%	6	6	0.0%	10	10	0.0%
TOTAL:	2466	2392	-3.0%	227	233	2.6%	10.9	10.3	-5.5%

Table 24 – NLIS Predicted Design Impact Summary

4.3 Dependency Analysis Results

This section presents the results of the data and control dependency analyses performed on the original system (the "Before" case) as well as the predicted state of the refactored system (the "After" case). The adjacency graphs, illustrating the direct data and control dependencies in the system prior to calculating the transitive closure, are presented here to help see the changes introduced by the refactoring.

For the NLIS case study, the combined dependency graphs are not presented as these charts do not look substantially different from the very dense data dependency graphs presented below. However, the re-test impact was calculated in exactly the same manner as in the trial case study, using the logical OR of the data graph and the transposed control graph.

The summary comparison of the dependency graph fill ratio in Table 25 shows a different result than obtained in the trial case study. The NLIS results show only a very slight decrease (0.4%) in the overall density of dependency paths as a result of the restructuring, compared with a 22% reduction in the Traffic Light Simulation system. As well, the NLIS control dependency path density actually increased by 13.5% as a result of the refactoring.

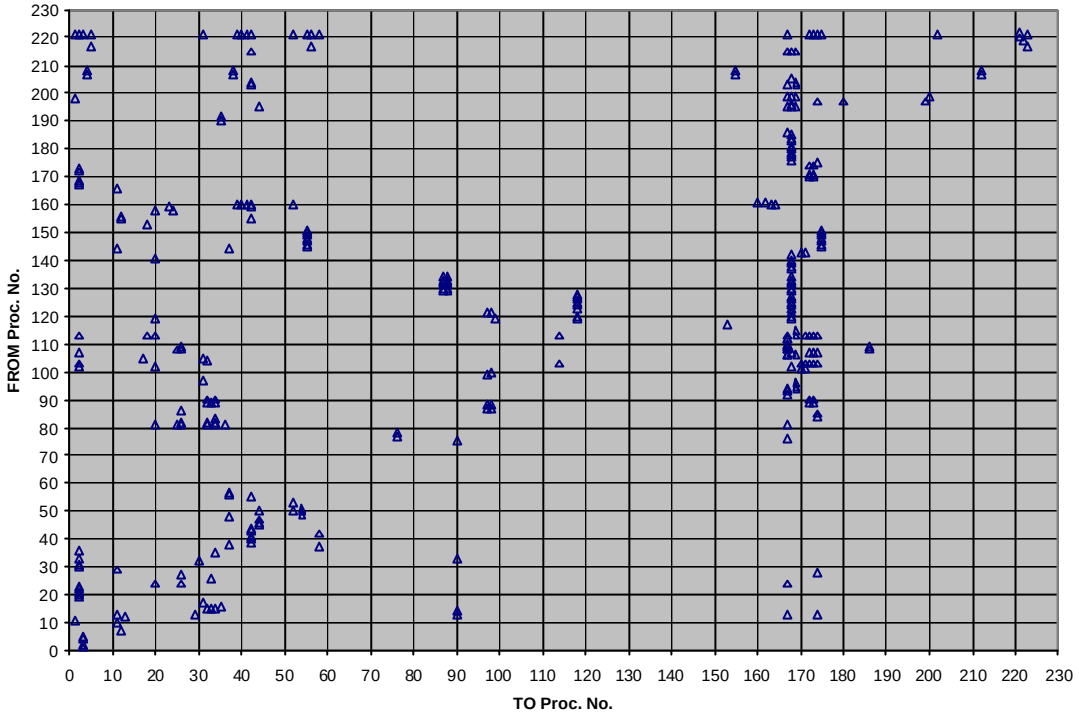

Figure 13 – NLIS Control Dependency Adjacency Graph Before Re factoring

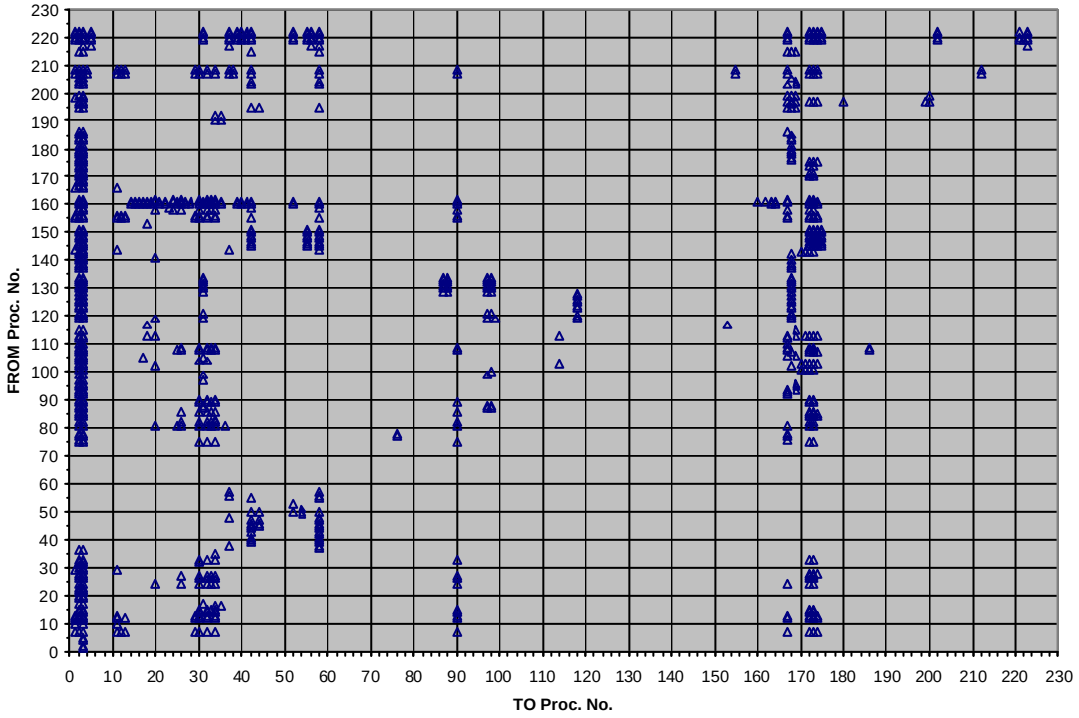

Figure 14 – NLIS Control Dependency Graph Before Refactoring

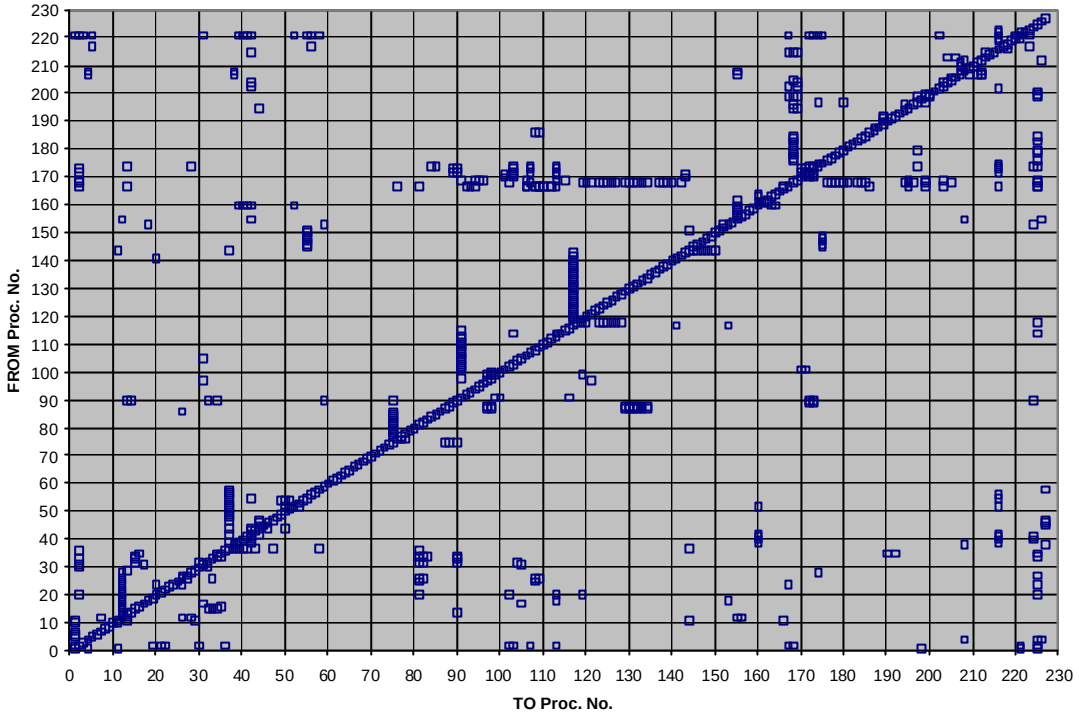

Figure 15 – NLIS Data Dependency Adjacency Graph Before Refactoring

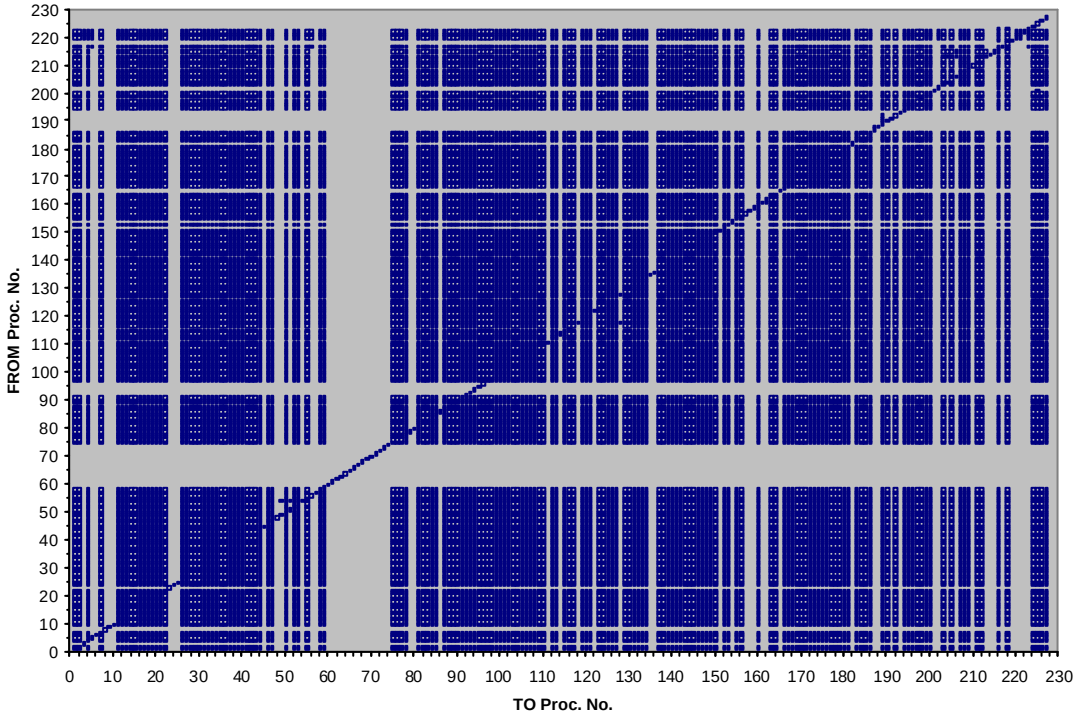

Figure 16 – NLIS Data Dependency Graph Before Refactoring

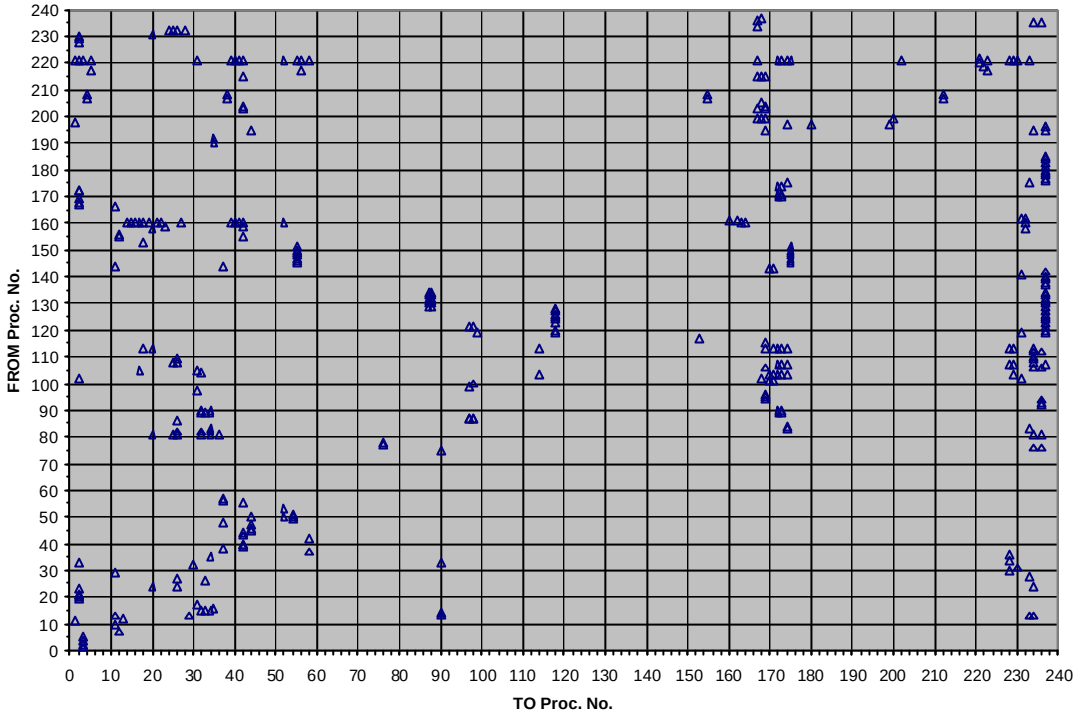

Figure 17 – NLIS Control Dependency Adjacency Graph After Refactoring

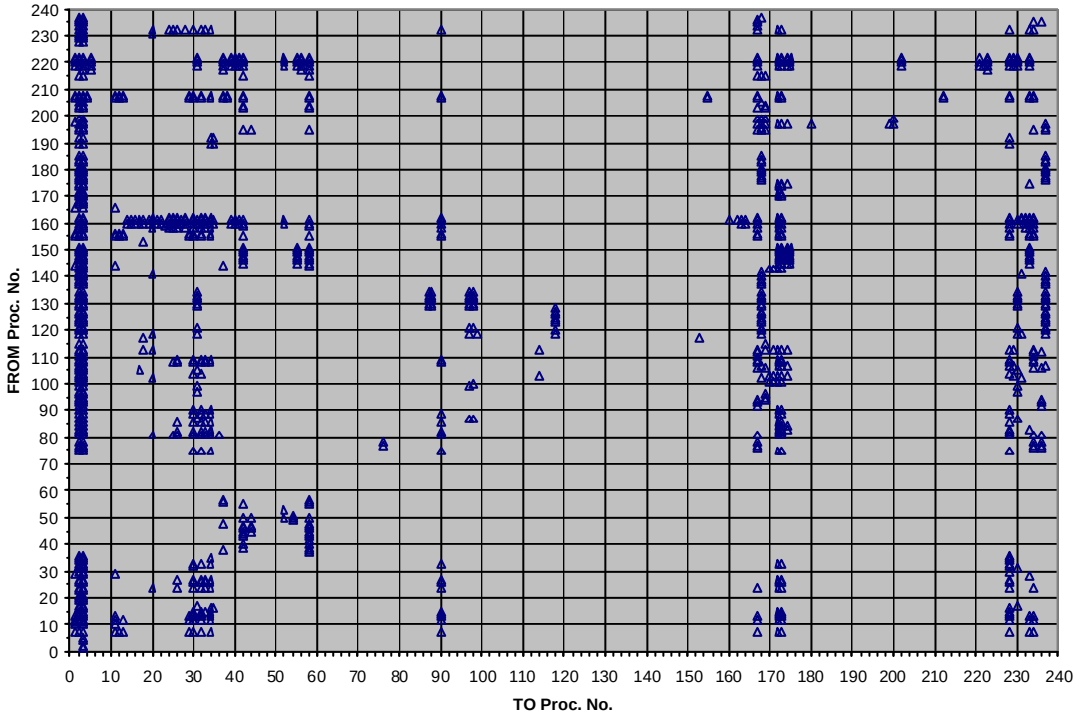

Figure 18 – NLIS Control Dependency Graph After Refactoring

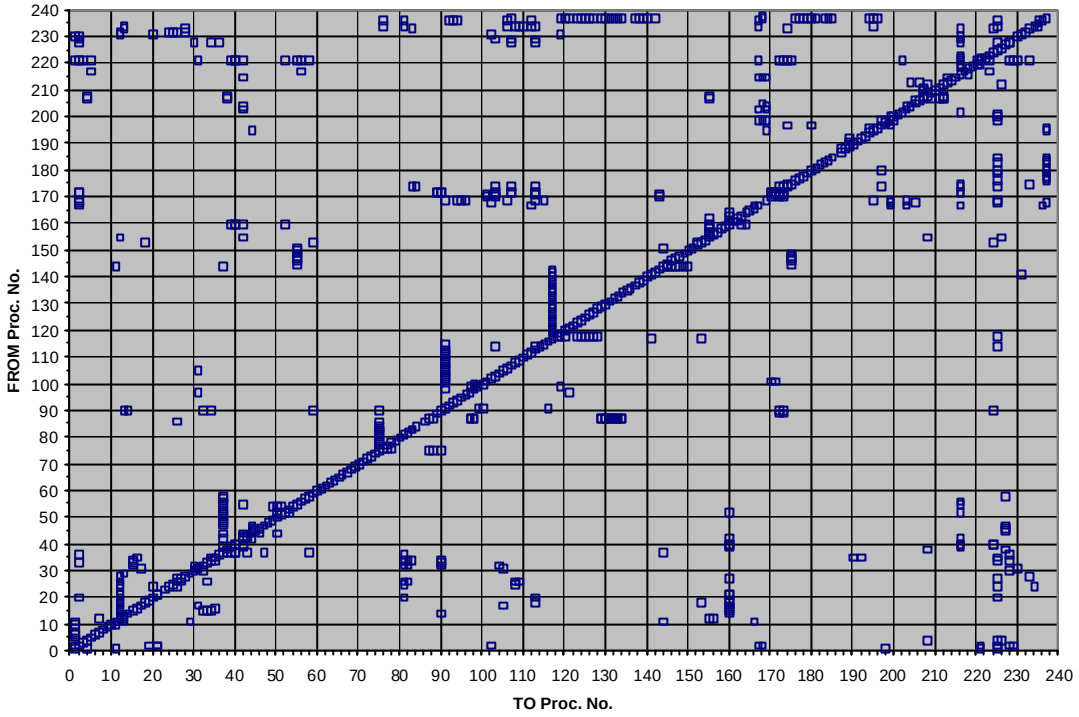

Figure 19 – NLIS Data Dependency Adjacency Graph After Refactoring

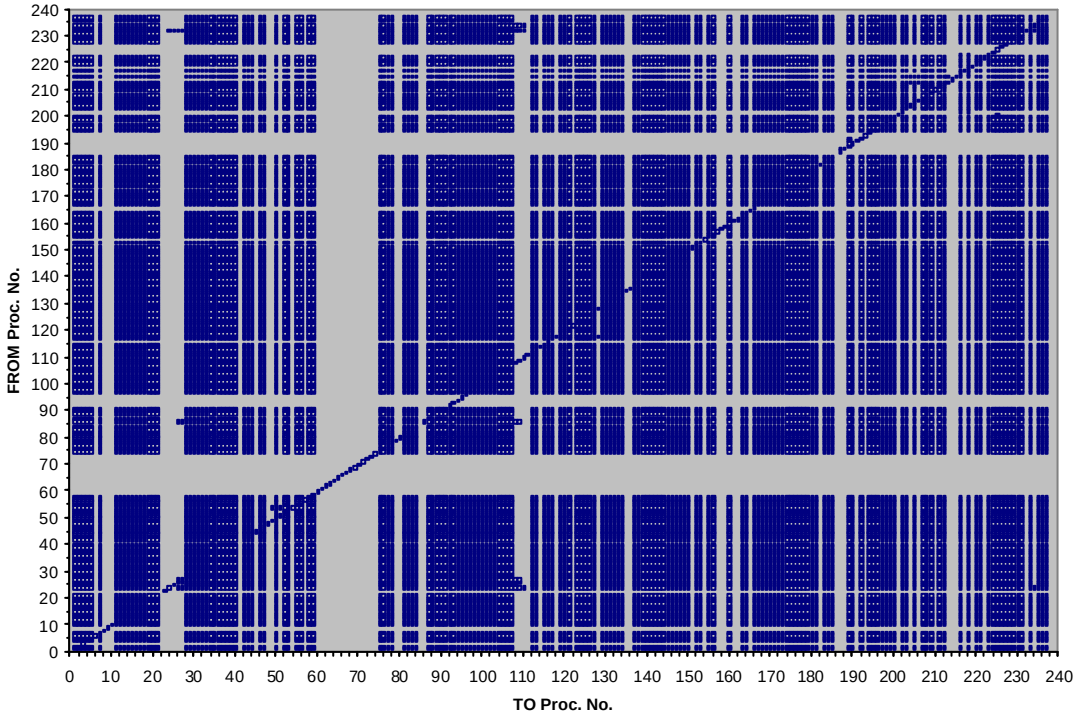

Figure 20 – NLIS Data Dependency Graph After Refactoring

	BEFORE		AFTER		
Graph	**No. of Dependencies**	**Matrix Fill Ratio**	**No. of Dependencies**	**Matrix Fill Ratio**	**% Change**
Data Dependency Graph	27957	54.3%	29512	53.9%	-0.7%
Control Dependency Graph	941	1.8%	1135	2.1%	13.5%
Overall Dependency Graph	28291	54.9%	29952	54.7%	-0.4%

Table 25 – NLIS Dependency Graph Fill Ratio Comparison

Comparing Figure 16 with Figure 20, it is clear that the NLIS refactoring plan as presented will not have a dramatic impact on the overall dependency structure of the system. The before and after data dependency graphs are both very dense with a similar pattern.

The following sections attempt to measure the difference in average re-test impact between the original system and the modified system using the above dependency relationships.

4.4 Re-test Impact Analysis Results

This section illustrates the probability distributions of the regression testing impact for the NLIS code before and after refactoring, assuming a random maintenance activity as defined in section 3.

In Figure 21, Figure 23, and Figure 25, each data point represents the re-test impact of a single procedure versus the probability of that impact occurring for a random maintenance event. Note that the impact data is expressed as a percentage of the total SLOC in the system rather than as an absolute SLOC number. For the combined graph in Figure 25, the code size reference is the original, unchanged version of the NLIS code.

Figure 22 and Figure 24 illustrate the distribution of re-test impact with respect to the total number of procedures in the system.

Table 26 summarizes the average re-test impact for the two systems, and predicts that the re-test impact for the refactored system will be 5.8% less than for the original system. The data behind these calculations is listed in the case study appendix, in sections 7.1.2 and 7.2.2.

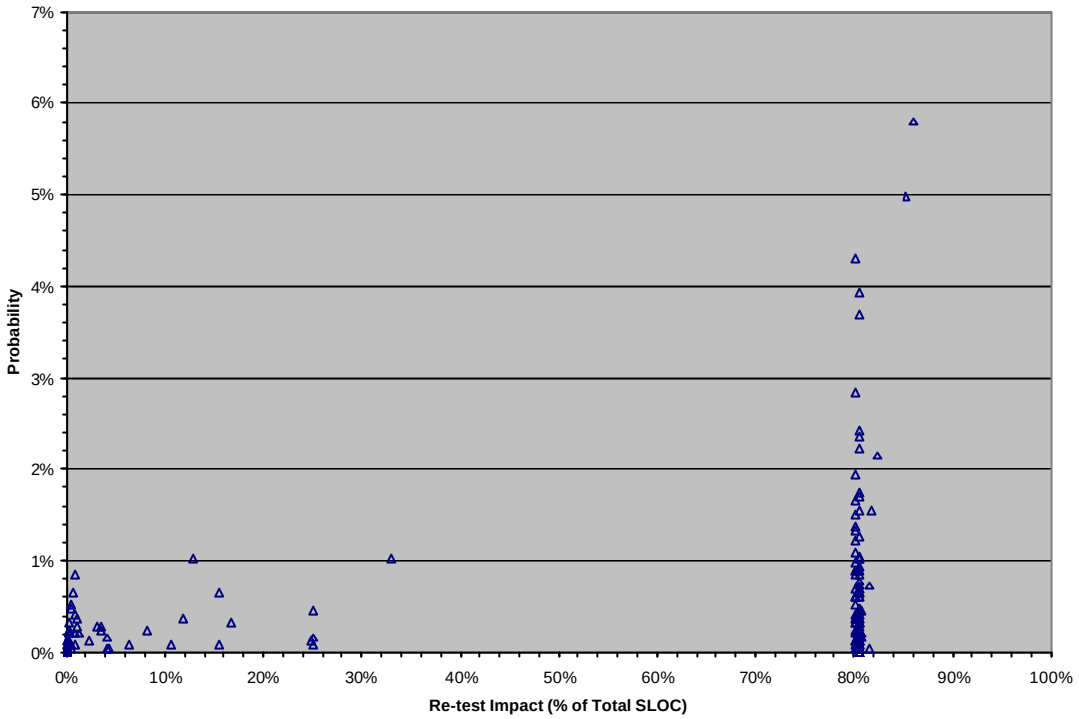

Figure 21 – NLIS Re-test Impact Probability Before Refactoring

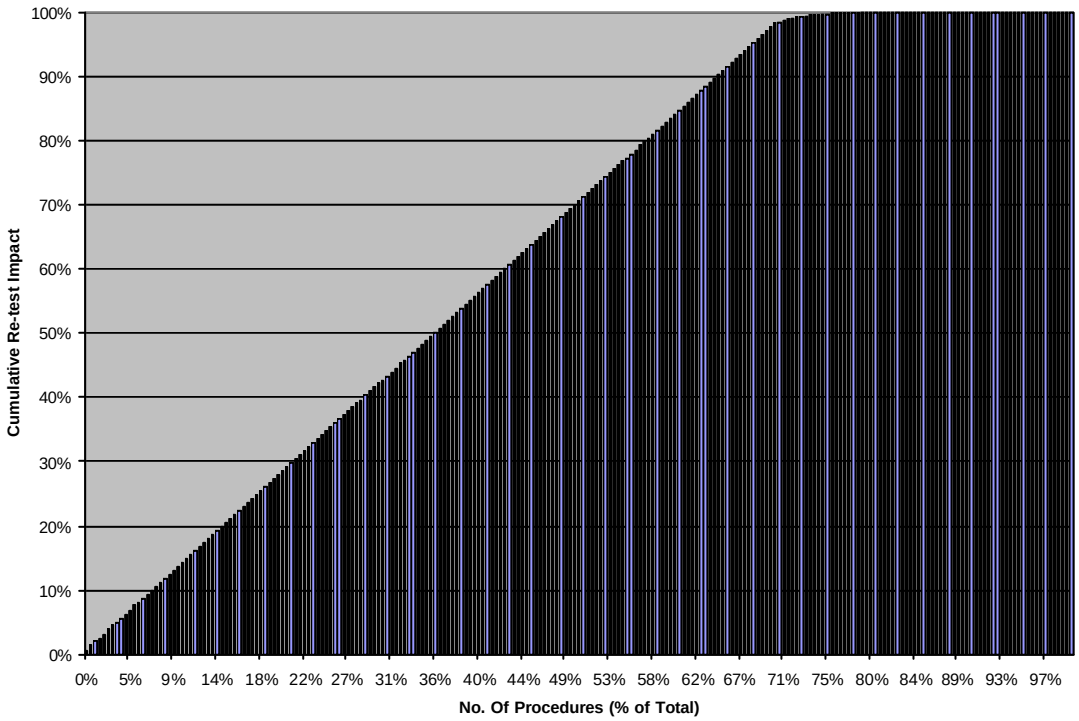

Figure 22 – NLIS Cumulative Re-test Impact Before Refactoring

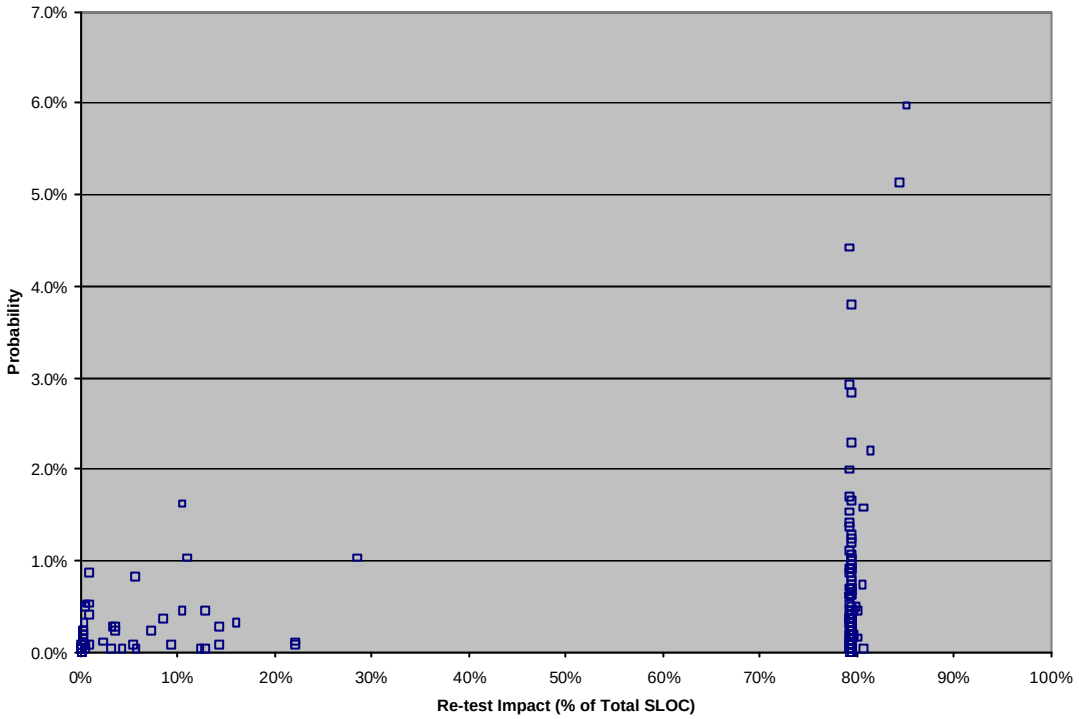

Figure 23 – NLIS Re-test Impact Probability After Refactoring

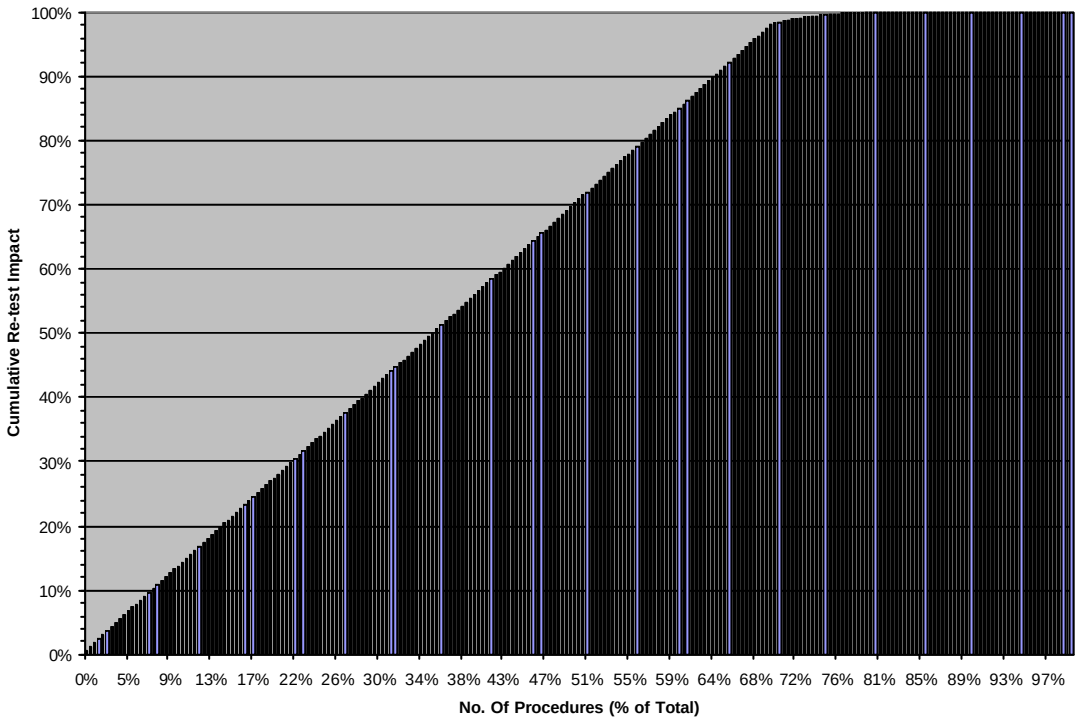

Figure 24 – NLIS Cumulative Re-test Impact After Refactoring

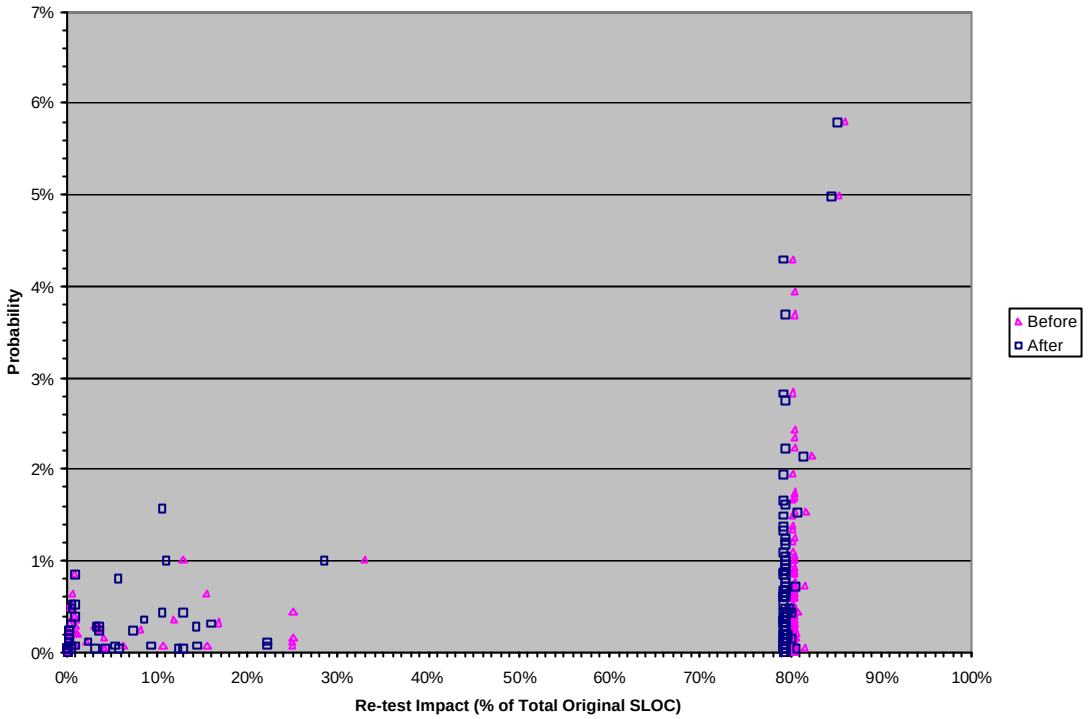

Figure 25 – NLIS Re-test Impact Probability Comparison

	Total SLOC	Mean Re-test Impact
Before	2466	1782
After	2392	1678
% Change	-3.0%	-5.8%

Table 26 – NLIS Mean Re-test Impact Summary

When comparing the results in Figure 25 with the equivalent results from the trial case study, we see a less pronounced difference between the Before and After impact probability distributions. For example, the peak probability in Figure 25 is the same for both distributions. However, the After distribution is clearly shifted to the left with respect to the Before distribution – this shift is especially apparent on either side of the 80% impact threshold.

4.5 Cost-Benefit Effort Calculations

4.5.1 COCOMO Model Assumptions

The tables below show the assumed parameters for the NLIS project used as input to the COCOMOII.2000 model.

Param.	Very Low	Low	Nominal	High	Very High	Extra High	SCORE
RELY	0.82	0.92	1.00	1.10	1.26	N/A	1.10
DATA	N/A	0.90	1.00	1.14	1.28	N/A	1.00
CPLX	0.73	0.87	1.00	1.17	1.34	1.74	0.87
RUSE	N/A	0.95	1.00	1.07	1.15	1.24	1.00
DOCU	0.81	0.91	1.00	1.11	1.23	N/A	1.00
TIME	N/A	N/A	1.00	1.11	1.29	1.63	1.00
STOR	N/A	N/A	1.00	1.05	1.17	1.46	1.00
PVOL	N/A	0.87	1.00	1.15	1.30	N/A	1.00
ACAP	1.42	1.19	1.00	0.85	0.71	N/A	0.85
PCAP	1.34	1.15	1.00	0.88	0.76	N/A	1.00
PCON	1.29	1.12	1.00	0.90	0.81	N/A	1.00
APEX	1.22	1.10	1.00	0.88	0.81	N/A	1.00
PLEX	1.19	1.09	1.00	0.91	0.85	N/A	1.00
LTEX	1.20	1.09	1.00	0.91	0.84	N/A	1.00
TOOL	1.17	1.09	1.00	0.90	0.78	N/A	1.00
SITE	1.22	1.09	1.00	0.93	0.86	0.80	1.22
SCED	1.43	1.14	1.00	1.00	1.00	N/A	1.00
						Product:	0.99

Table 27 – NLIS Assumed COCOMOII.2000 Effort Adjustment Factors

Param.	Very Low	Low	Nominal	High	Very High	Extra High	SCORE
PREC	6.20	4.96	3.72	2.48	1.24	0.00	3.72
FLEX	5.07	4.05	3.04	2.03	1.01	0.00	2.03
RESL	7.07	5.65	4.24	2.83	1.41	0.00	4.24
TEAM	5.48	4.38	3.29	2.19	1.10	0.00	3.29
EPML	7.80	6.24	4.68	3.12	1.56	0.00	4.68
						Sum:	17.96
						Exponent	1.09

Table 28 – NLIS Assumed COCOMOII.2000 Model Scale Factors

Note that the Re-use Model parameters for NLIS are assumed to be the same as for the trial case study presented earlier. We have assigned the Effort Adjustment Factors and Model Scale Factors in Table 27 and Table 28 based on a review of the NLIS project documentation – these assumptions are slightly different than for the trial case study (which was an academic system). We felt that since NLIS is a commercial system we should attempt to pass meaningful numbers to the model rather than assuming the default values.

4.5.2 Effort Prediction Model Results

The results of the COCOMOII.2000 analysis, using the same technique as for the trial case study, are summarized in Table 29. The analysis predicts that the refactoring will result in a benefit of 0.14 person-months per maintenance event, meaning that the refactoring becomes cost effective after 28 maintenance activities.

Model Parameters	Cost Calc.	Benefit Calc. (Before)	(After)
Code Size (KSLOC)	2.466	2.466	2.392
DM (0-100)	11.9	5.0	5.0
CM (0-100)	14.4	5.0	5.0
IM (0-100+)	100.0	72.3	70.2
Equivalent KSLOC	1.294	0.869	0.824
Effort (Person-months)	3.86	2.50	2.36
Net Benefit/Maint. Activity		0.14	
Break-Even Point (# Activities)		27.50	

Table 29 – Effort Prediction Model Calculation Results

4.6 Case Study Conclusions

The proposed refactoring is predicted to decrease the overall code size (by 3%) and increase the number of procedures in the system (by 2.6%). In addition, the density of the overall dependency path graph for the system is predicted to decrease by 0.4%.

The refactoring appears to slightly reduce the peak code re-test impact in the impact probability distribution. Compared with the original distribution, the refactored distribution appears shifted to the left.

Procedure-level dependency analysis predicts that the mean regression testing impact (in terms of affected SLOC) of a random maintenance activity will decrease by 5.8% as a result of the proposed design restructuring.

Cost estimation modeling using COCOMOII.2000 suggests that the restructuring will be cost effective if 28 or more maintenance events occur after the refactoring investment.

The NLIS case study results are less pronounced than the results of the trial case study, although the two studies show very similar trends in terms of the parameters listed above. One possible explanation is that the effect of source-code level refactoring is less influential as system size increases. This hypothesis should be tested in future work. It is possible that the results are less pronounced simply because there were relatively fewer refactoring opportunities identified for NLIS than for the Traffic Light Simulation code. An experiment to test the effect of system size on refactoring benefits should take this factor into account.

Based on the case study results, we conclude that this sort of analysis is practical for commercial-grade systems and yields insight into the impact of source code refactoring.

5 Project Conclusions and Future Work

5.1 Achievements

In this project we have defined a new way to attack the problem of predicting the downstream maintenance benefits of software design changes. In addition, we established through a literature survey that existing knowledge is not sufficient to construct a general purpose predictive model of maintenance effort based solely on software design metrics. Instead of a design metrics approach, we have proposed a methodology that uses dependency analysis to evaluate competing designs and perform a cost-benefit analysis of legacy system refactoring.

The proposed methodology makes predictions of system regression testing impact quantified through code dependency analysis, followed by maintenance effort predictions using the COCOMOII.2000 model. A major assumption of this methodology is that maintenance activities are random with respect to the design location, thereby de-coupling the maintenance effort prediction problem from design metrics. In other words, we assume that maintenance will happen and we focus on the cost of this maintenance rather than the originating cause of the maintenance.

In order to demonstrate the methodology we performed two case studies: one trial study of a small Java system (740 SLOC) built in an academic environment, and a second study of a larger industrial Java application (2466 SLOC). These case studies generated concrete results regarding the effect of source code refactoring on system data and control dependencies, as well as the effect on code size and number of procedures. Specifically, the case study results provide support to the following conclusions about refactoring:

- Refactoring tends to reduce the average re-test impact associated with a random maintenance activity within the design.
- Refactoring tends to reduce the amount of code in the system.
- Refactoring tends to increase the number of procedures in the system.
- Refactoring tends to reduce the density of dependency paths within the overall system dependency graph.
- Refactoring tends to reduce the peak re-test impact within the impact probability distribution.
- The regression testing benefits associated with refactoring may decrease as system size increases (this is a hypothesis only).
- Class-level dependency analysis appears to be more conservative than procedure-level dependency analysis regarding the benefits of refactoring.

In addition, the methodology generated quantitative predictions of Return on Investment (ROI) for the proposed design restructuring, providing some basis for project teams to assess whether proposed restructuring activities are worth the up-front effort.

At the very least, this project has contributed some data to the software engineering literature on the measurement of low-level design transformations.

5.2 Future Work

In order to make stronger claims about this proposed methodology and the maintenance impact of refactoring, we feel the following work should be undertaken:

- Further validation of ROI calculations performed using this methodology using maintenance data from industry. Such a project might involve measuring maintenance activities on parallel streams of a system (one stream using a refactored design and another using the original design, with both streams undergoing the same maintenance regime).
- Further investigation and validation of the approach regarding the use of the COCOMOII.2000 re-use model for maintenance effort prediction.
- The identification or development of automated tool support for this methodology. The manual analysis performed in the case studies becomes very difficult and labour-intensive for large systems.
- The application of this methodology to multiple larger systems in order to evaluate the relationship between the predicted benefits of refactoring and system code size.

6 References

[1] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[2] Ladan Tahvildari, Assessing the Impact of Using Design-pattern-based Systems, M.A.Sc. Thesis, Department of Electrical Engineering, University of Waterloo, 1999.

[3] Ladan Tahvildari, Kostas Kontogiannis, John Mylopolous, "Requirements-Driven Software Re-engineering", In Proceedings of the 8th IEEE Working Conference on Reverse Engineering (WCRE 2001), Stuttgart, Germany, pp. 71-80, October 2001.

[4] L. Briand, J. Wüst, John W. Daly, V. Porter, "Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems", Journal of Systems and Software, 51(2000) p 245-273.

[5] L. Briand, J. Wüst, "The Impact of Design Properties on Development Cost in Object-Oriented Systems", IEEE Transactions on Software Engineering, November 2001.

[6] L. Briand, B. Freimut, F. Vollei, "Assessing the Cost-Effectiveness of Inspections by Combining Project Data and Expert Opinion", International Software Engineering Research Network (ISERN), 1999, ISERN-99-14 Version 2, published in the Proceedings of IEEE ISSRE 2000, San Jose, USA.

[7] B. Kitchenham, L. Pickard, and S.L. Pfleeger, Case studies for method and tool evaluation., IEEE Software, vol 12, no. 4, p 52-62, 1995.

[8] Sullivan, K.J., P. Chalasani, S. Jha and V. Sazawal, “Software Design as an Investment Activity: A Real Options Perspective,” in Real Options and Business Srategy: Applications to Decision Making, L. Trigeorgis, consulting editor, Risk Books, 1999. (Previously Sullivan et al., “Software Design Decisions as Real Options,” Technical Report 97-14, University of Virginia Department of Computer Science, Charlottesville, Virginia, USA, 1997.)

[9] B. Boehm et al. , Software Cost Estimation with COCOMO II, Prentice Hall PTR, 2000.

[10] Danilo Caivano, Filippo Lanubile, Giuseppe Visaggio, "Software Renewal Process Comprehension using Dynamic Effort Estimation", IEEE ICSM 2001, Florence Italy.

[11] Macario Polo, Mario Piattini, Francisco Ruiz, "Using code metrics to predict maintenance of legacy programs: a case study", IEEE ICSM 2001, Florence, Italy.

[12] Rainer Gerlich and Ulrich Denskat, "A Cost Estimation Model for Software Maintenance", Proceedings of ESCOM 1994, Ivrea, Italy.

[13] Basili, V., Briand, L., Condon, S., Kim, Y., Melo, W. y Valett, J.D., "Understanding and Predicting the Process of Software Maintenance Releases", Proceedings of the International Conference on Software Engineering, IEEE Computer Society, Los Alamitos, CA (USA), 1996, pp. 464-474.

[14] Fenton, N. E. and Ohlsson, N. , "Quantitative Analysis of Faults and Failures in a Complex Software System", IEEE Transactions on Software Engineering, vol. 26, no. 8, August 2000, pp 797-814.

[15] Coleman, D., Ash, D., Lowther, B., and Oman, P., "Using Metrics to Evaluate Software System Maintainability", IEEE Computer, Vol. 27, No. 8, August 1994, pp. 44-49.

[16] L. Briand, J. Wuest, Empirical Studies of Quality Models in Object-Oriented Systems, to be published in Advances in Computers, Academic Press, updated Feb. 18, 2002 (from L. Briand Website)

[17] Lionel C. Briand, Walcelio Melo, Juergen Wuest ; Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects; International Software Engineering Research Network (ISERN), 2000; ISERN-00-06 Version 2. Accepted for publication in IEEE Transactions on Software Engineering.

[18] F. Abreu, W. Melo, “Evaluating the Impact of Object-Oriented Design on Software Quality”, Proceedings of Metrics 1996.

[19] S. Chidamber, D. Darcy, C. Kemerer, “Managerial use of Metrics for Object-Oriented Software: An Exploratory Analysis”, IEEE Transactions on Software Engineering, 24 (8), 629-639, 1998.

[20] S.R. Chidamber, C.F. Kemerer, “A Metrics Suite for Object Oriented Design”, IEEE Transactions on Software Engineering, 20 (6), 476-493, 1994.

[21] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller, and François Lustman, “A Change Impact Model for Changeability Assessment in Object-Oriented Systems”, Science of Computer Programming, Elsevier Science Publishers, 2001 (To appear).

[22] Ladan Tahvildari, Kostas Kontogiannis, "On the Role of Design Patterns in Quality-Driven Re-engineering", In Proceedings of the 6th IEEE European

Conference on Software Maintenance and Reengineering (CSMR), Budapest, Hungary, pp. 230-240, March 2002.

[23] B. Boehm and K.J. Sullivan, invited paper, "Software Economics: A Roadmap," 22nd International Conference on Software Engineering, June, 2000.

[24] J. F. Ramil and M. M. Lehman, "Cost Estimation and Evolvability Monitoring for Software Evolution Processes", WESS 2000 Workshop on Empirical Studies of Software Maintenance, San Jose, California, October 14 2000.

[25] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability", Journal of Systems and Software, vol. 23 issue 2, pp111-122, 1993.

[26] IEEE Std. 610.12-1990, ‘ Glossary of Software Engineering Terminology,” in Software Engineering Standards Collection, IEEE CS Press, Los Alamitos, CA, Order No. 1048-06T, 1993.

[27] F. G. Wilkie and B. Hylands, "Measuring Complexity in C++ Application Software", Software Practice and Experience Vol. 28, no. 5, 1998, pp.513-546.

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Longman, Inc., 1995.

[29] J. P. Loyall and S. A. Mathisen, "Using Dependence Analysis to Support the Software Maintenance Process", Proceedings of the Conference on Software Maintenance, IEEE CS Press, Los Alamitos, CA, 1993, pp. 282-291.

7 Appendix – Case Study Data

7.1 Before Refactoring

7.1.1 Procedure Characteristics

Proc. No.	Class	Method	Size (SLOC)
1	STL_Agent	STL_Agent	55
2	STL_Agent	run	97
3	STL_Agent	main	7
4	STL_Agent	setupProperties	18
5	STL_Agent	startLogger	5
6	STL_Agent	getRuntimeDir	1
7	STL_Agent	getConfigFileFullPath	1
8	STL_Agent	renameFile	5
9	STL_Agent	mergeVectors	6
10	STL_Agent	setupScafFiles	7
11	STL_Agent	initialize	8
12	STL_Database	STL_Database	26
13	STL_Database	create	3
14	STL_Database	getEndUser	3
15	STL_Database	getPartyList	8
16	STL_Database	getPreviousDays	22
17	STL_Database	getRetroactiveBusinessDaysForDate	25
18	STL_Database	getTransactionsForParty	15
19	STL_Database	isBusinessDay	9
20	STL_Database	updateSettlementStatus	43
21	STL_Database	getSageMapSupplierCode	8
22	STL_Database	getSageSrSupplierCode	8
23	STL_Database	close	1
24	STL_Database	updateEnduserEFT	16
25	STL_Database	isPartyInSync	16
26	STL_Database	synchronizeEndUser	42
27	STL_Database	assignRankingToEftStatus	11
28	STL_Database	getSystemTableParameter	11
29	STL_BusinessManager	STL_BusinessManager	6
30	STL_BusinessManager	buildPartySettlementList	3
31	STL_BusinessManager	buildTransactionSettlementList	25
32	STL_BusinessManager	buildPartyList	17
33	STL_BusinessManager	synchronizeAllEndusers	10
34	STL_BusinessManager	buildAUDDISPartyList	24
35	STL_BusinessManager	firstTransactionDelay	30
36	STL_BusinessManager	mergePartiesByEFTSTATUS	16
37	REP_ManagerTransactions	REP_ManagerTransactions	70
38	REP_ManagerTransactions	setupProperties	27
39	REP_ManagerTransactions	getManagerDetails	17
40	REP_ManagerTransactions	getMapVatRate	13
41	REP_ManagerTransactions	getAdministratorEmail	10
42	REP_ManagerTransactions	generateCSVReport	41
43	REP_ManagerTransactions	getResults	21
44	REP_ManagerTransactions	createLine	2

45	REP_ManagerTransactions	displayNegative	3
46	REP_ManagerTransactions	getTransactionType	15
47	REP_ManagerTransactions	getServiceType	106
48	REP_ManagerTransactions	dump	6
49	REP_ManagerTransactions	getNonVatableAmount	8
50	REP_ManagerTransactions	getVatableAmount	8
51	REP_ManagerTransactions	getVAT	3
52	REP_ManagerTransactions	getVATRate	34
53	REP_ManagerTransactions	isMapTransaction	3
54	REP_ManagerTransactions	getAmount	1
55	REP_ManagerTransactions	sendMailMessage	22
56	REP_ManagerTransactions	startLogger	4
57	REP_ManagerTransactions	setupScafFiles	7
58	REP_ManagerTransactions	main	9
59	STL_DatabaseInterface	STL_DatabaseInterface	4
60	STL_DatabaseInterface	close	0
61	STL_DatabaseInterface	getEndUser	0
62	STL_DatabaseInterface	getPartyList	0
63	STL_DatabaseInterface	getRetroactiveBusinessDaysForDate	0
64	STL_DatabaseInterface	getTransactionsForParty	0
65	STL_DatabaseInterface	isBusinessDay	0
66	STL_DatabaseInterface	getPreviousDays	0
67	STL_DatabaseInterface	updateSettlementStatus	0
68	STL_DatabaseInterface	getSageMapSupplierCode	0
69	STL_DatabaseInterface	getSageSrSupplierCode	0
70	STL_DatabaseInterface	updateEnduserEFT	0
71	STL_DatabaseInterface	isPartyInSync	0
72	STL_DatabaseInterface	synchronizeEndUser	0
73	STL_DatabaseInterface	assignRankingToEftStatus	0
74	STL_DatabaseInterface	getSystemTableParameter	0
75	STL_EndUser	STL_EndUser	23
76	STL_EndUser	settlementAmount	1
77	STL_EndUser	totalFees	1
78	STL_EndUser	totalVat	1
79	STL_EndUser	creditCode	1
80	STL_EndUser	debitCode	1
81	STL_EndUser	eftStatus	1
82	STL_EndUser	eftStatusDate	1
83	STL_EndUser	testMode	3
84	STL_EndUser	invoiceEmail	1
85	STL_EndUser	invoicedByEmail	3
86	STL_EndUser	setEFTSTATUS	2
87	STL_EndUser	postCredit	6
88	STL_EndUser	postDebit	6
89	STL_EndUserFactory	STL_EndUserFactory	0
90	STL_EndUserFactory	create	19
91	STL_Party	STL_Party	33
92	STL_Party	creditCode	0
93	STL_Party	debitCode	0
94	STL_Party	settlementAmount	0
95	STL_Party	totalFees	0
96	STL_Party	totalVat	0
97	STL_Party	addTransaction	4

98	STL_Party	removeTransaction	5
99	STL_Party	addTransactionToList	1
100	STL_Party	removeTransactionFromList	1
101	STL_Party	getTransaction	6
102	STL_Party	getTransactions	1
103	STL_Party	getNumberOfTransactions	1
104	STL_Party	electronicSettlementFrequency	1
105	STL_Party	getSettlementGracePeriod	1
106	STL_Party	getBacsPayeeName	1
107	STL_Party	getPartyName	1
108	STL_Party	sortCode	1
109	STL_Party	accountNumber	1
110	STL_Party	referenceNumber	1
111	STL_Party	weeklySettlementDay	1
112	STL_Party	paymentMethod	1
113	STL_Party	getPartyId	1
114	STL_Party	toString	5
115	STL_Party	getPartyType	1
116	STL_Party	setPartyType	1
117	STL_Transaction	STL_Transaction	48
118	STL_Transaction	toString	12
119	STL_Transaction	domainTransactionId	1
120	STL_Transaction	transactionType	1
121	STL_Transaction	action	1
122	STL_Transaction	customerId	1
123	STL_Transaction	acctSysCustId	1
124	STL_Transaction	serviceRequestId	1
125	STL_Transaction	serviceRequestType	1
126	STL_Transaction	transactionDate	1
127	STL_Transaction	paymentMethod	1
128	STL_Transaction	currency	1
129	STL_Transaction	dataProviderFee	1
130	STL_Transaction	dataProviderVat	1
131	STL_Transaction	channelCharge	1
132	STL_Transaction	channelVat	1
133	STL_Transaction	hubCharge	1
134	STL_Transaction	hubVat	1
135	STL_Transaction	total	1
136	STL_Transaction	settlementDate	1
137	STL_Transaction	transactionMode	1
138	STL_Transaction	llc1	1
139	STL_Transaction	con29	1
140	STL_Transaction	cr21	1
141	STL_Transaction	setSettlementDate	1
142	STL_Transaction	sageSRSupplierCode	1
143	STL_Transaction	managerId	1
144	UTL_Mail	UTL_Mail	15
145	UTL_Mail	setFrom	1
146	UTL_Mail	setSubject	1
147	UTL_Mail	setMessageBody	1
148	UTL_Mail	setRecipients	1
149	UTL_Mail	addAttachment	2
150	UTL_Mail	reset	6

151	UTL_Mail	send	25
152	STL_TransactionFactory	STL_TransactionFactory	1
153	STL_TransactionFactory	create	37
154	STL_TransactionException	STL_TransactionException	1
155	UTL_DBConnection	UTL_DBConnection	15
156	UTL_DBConnection	getConnection	1
157	UTL_DBConnection	rollback	2
158	UTL_DBConnection	commit	2
159	UTL_DBConnection	close	2
160	UTL_DBConnection	executeQuery	9
161	UTL_DBConnection	setValues	25
162	UTL_DBConnection	executeUpdate	8
163	UTL_DBConnection	executeIdQuery	8
164	UTL_DBConnection	getLatestSequenceValue	9
165	UTL_DBConnection	zValue	4
166	STL_Reports	STL_Reports	4
167	STL_Reports	generateBacsTransactionFile	60
168	STL_Reports	generateSageTransactionFile	38
169	STL_Reports	createBacsSummary	31
170	STL_Reports	userHasNonManagedTransactions	9
171	STL_Reports	getUsersManagers	11
172	STL_Reports	createInvoices	21
173	STL_Reports	createInvoiceStatements	22
174	STL_Reports	generateReport	58
175	STL_Reports	mailReport	16
176	STL_Reports	statementTransactionType	16
177	STL_Reports	statementParty	8
178	STL_Reports	statementAction	6
179	STL_Reports	statementServiceRequestType	91
180	STL_Reports	statementDate	2
181	STL_Reports	statementPaymentMethod	8
182	STL_Reports	statementCurrency	6
183	STL_Reports	statementTransactionMode	6
184	STL_Reports	statementServiceDeliveryMechanism	10
185	STL_Reports	getSupplierCode	6
186	STL_Reports	transactionAlreadyCreated	12
187	RPT_PrintDate	RPT_PrintDate	2
188	RPT_PrintDate	main	21
189	STL_Day	STL_Day	11
190	STL_Day	getDay	1
191	STL_Day	getDayOfWeek	1
192	STL_Day	isBusinessDay	1
193	STL_PartyException	STL_PartyException	1
194	STL_Util	STL_Util	5
195	STL_Util	doubleToString	6
196	STL_Util	justifyString	12
197	STL_Util	dateToString	3
198	STL_Util	stringToDate	5
199	STL_Util	buildFilenameString	2
200	STL_Util	prepJob	3
201	STL_Util	prepAttachment	3
202	STL_Util	ftpFile	9
203	UTL_CSVFormattedFile	UTL_CSVFormattedFile	3

204	UTL_CSVFormattedFile	addLine	6
205	UTL_FCFFormattedFile	UTL_FCFFormattedFile	7
206	UTL_FCFFormattedFile	addLine	10
207	UTL_Property	UTL_Property	17
208	UTL_Property	getPropertyValue	2
209	UTL_Property	setPropertyValue	5
210	UTL_Property	writeToDisk	13
211	UTL_Property	tokenize	6
212	UTL_Property	main	15
213	UTL_FormattedFile	UTL_FormattedFile	2
214	UTL_FormattedFile	addLine	0
215	UTL_FormattedFile	generateFile	9
216	UTL_Logger	UTL_Logger	18
217	UTL_Logger	setLogFile	1
218	UTL_Logger	setOutput	1
219	UTL_Logger	label	2
220	UTL_Logger	severityToInt	4
221	UTL_Logger	log	11
222	UTL_Logger	output	3
223	UTL_Logger	main	5
224	UTL_DBConstant	UTL_DBConstant	53
225	STL_Constants	STL_Constants	123
226	UTL_Constant	UTL_Constant	38
227	REP_Constants	REP_Constants	143
		TOTAL:	2466

7.1.2 Re-test Impact Calculation

Proc. No.	Size (SLOC)	% of Total	Re-test Impact (SLOC)	% of Total	Contrib. to Mean
1	55	2.2%	1985	80.5%	44.3
2	97	3.9%	1985	80.5%	78.1
3	7	0.3%	25	1.0%	0.1
4	18	0.7%	1985	80.5%	14.5
5	5	0.2%	31	1.3%	0.1
6	1	0.0%	1	0.0%	0.0
7	1	0.0%	1985	80.5%	0.8
8	5	0.2%	5	0.2%	0.0
9	6	0.2%	6	0.2%	0.0
10	7	0.3%	77	3.1%	0.2
11	8	0.3%	1985	80.5%	6.4
12	26	1.1%	1985	80.5%	20.9
13	3	0.1%	1985	80.5%	2.4
14	3	0.1%	1985	80.5%	2.4
15	8	0.3%	1985	80.5%	6.4
16	22	0.9%	1978	80.2%	17.6
17	25	1.0%	1985	80.5%	20.1
18	15	0.6%	1978	80.2%	12.0
19	9	0.4%	1985	80.5%	7.2
20	43	1.7%	1985	80.5%	34.6
21	8	0.3%	1985	80.5%	6.4
22	8	0.3%	1985	80.5%	6.4
23	1	0.0%	105	4.3%	0.0

Proc. No.	Size (SLOC)	% of Total	Re-test Impact (SLOC)	% of Total	Contrib. to Mean
24	16	0.6%	381	15.5%	2.5
25	16	0.6%	16	0.6%	0.1
26	42	1.7%	1985	80.5%	33.8
27	11	0.4%	1985	80.5%	8.9
28	11	0.4%	1985	80.5%	8.9
29	6	0.2%	1985	80.5%	4.8
30	3	0.1%	1985	80.5%	2.4
31	25	1.0%	1985	80.5%	20.1
32	17	0.7%	1985	80.5%	13.7
33	10	0.4%	1985	80.5%	8.0
34	24	1.0%	1978	80.2%	19.3
35	30	1.2%	1978	80.2%	24.1
36	16	0.6%	1985	80.5%	12.9
37	70	2.8%	1978	80.2%	56.1
38	27	1.1%	1978	80.2%	21.7
39	17	0.7%	1978	80.2%	13.6
40	13	0.5%	1978	80.2%	10.4
41	10	0.4%	1978	80.2%	8.0
42	41	1.7%	1978	80.2%	32.9
43	21	0.9%	1978	80.2%	16.8
44	2	0.1%	1978	80.2%	1.6
45	3	0.1%	55	2.2%	0.1
46	15	0.6%	1978	80.2%	12.0
47	106	4.3%	1978	80.2%	85.0
48	6	0.2%	85	3.4%	0.2
49	8	0.3%	9	0.4%	0.0
50	8	0.3%	1978	80.2%	6.4
51	3	0.1%	4	0.2%	0.0
52	34	1.4%	1978	80.2%	27.3
53	3	0.1%	1978	80.2%	2.4
54	1	0.0%	1	0.0%	0.0
55	22	0.9%	1978	80.2%	17.6
56	4	0.2%	102	4.1%	0.2
57	7	0.3%	86	3.5%	0.2
58	9	0.4%	1978	80.2%	7.2
59	4	0.2%	1982	80.4%	3.2
60	0	0.0%	0	0.0%	0.0
61	0	0.0%	0	0.0%	0.0
62	0	0.0%	0	0.0%	0.0
63	0	0.0%	0	0.0%	0.0
64	0	0.0%	0	0.0%	0.0
65	0	0.0%	0	0.0%	0.0
66	0	0.0%	0	0.0%	0.0
67	0	0.0%	0	0.0%	0.0
68	0	0.0%	0	0.0%	0.0
69	0	0.0%	0	0.0%	0.0
70	0	0.0%	0	0.0%	0.0
71	0	0.0%	0	0.0%	0.0
72	0	0.0%	0	0.0%	0.0
73	0	0.0%	0	0.0%	0.0
74	0	0.0%	0	0.0%	0.0

Proc. No.	Size (SLOC)	% of Total	Re-test Impact (SLOC)	% of Total	Contrib. to Mean
75	23	0.9%	1985	80.5%	18.5
76	1	0.0%	1985	80.5%	0.8
77	1	0.0%	1985	80.5%	0.8
78	1	0.0%	1985	80.5%	0.8
79	1	0.0%	1	0.0%	0.0
80	1	0.0%	1	0.0%	0.0
81	1	0.0%	1985	80.5%	0.8
82	1	0.0%	1985	80.5%	0.8
83	3	0.1%	1978	80.2%	2.4
84	1	0.0%	1985	80.5%	0.8
85	3	0.1%	1985	80.5%	2.4
86	2	0.1%	264	10.7%	0.2
87	6	0.2%	1985	80.5%	4.8
88	6	0.2%	1985	80.5%	4.8
89	0	0.0%	1985	80.5%	0.0
90	19	0.8%	1985	80.5%	15.3
91	33	1.3%	1978	80.2%	26.5
92	0	0.0%	1985	80.5%	0.0
93	0	0.0%	1985	80.5%	0.0
94	0	0.0%	1985	80.5%	0.0
95	0	0.0%	1985	80.5%	0.0
96	0	0.0%	1985	80.5%	0.0
97	4	0.2%	1985	80.5%	3.2
98	5	0.2%	1978	80.2%	4.0
99	1	0.0%	1985	80.5%	0.8
100	1	0.0%	1978	80.2%	0.8
101	6	0.2%	1985	80.5%	4.8
102	1	0.0%	1985	80.5%	0.8
103	1	0.0%	1985	80.5%	0.8
104	1	0.0%	1985	80.5%	0.8
105	1	0.0%	1985	80.5%	0.8
106	1	0.0%	1985	80.5%	0.8
107	1	0.0%	1985	80.5%	0.8
108	1	0.0%	1985	80.5%	0.8
109	1	0.0%	1985	80.5%	0.8
110	1	0.0%	1985	80.5%	0.8
111	1	0.0%	1	0.0%	0.0
112	1	0.0%	1985	80.5%	0.8
113	1	0.0%	1985	80.5%	0.8
114	5	0.2%	5	0.2%	0.0
115	1	0.0%	1985	80.5%	0.8
116	1	0.0%	1979	80.3%	0.8
117	48	1.9%	1978	80.2%	38.5
118	12	0.5%	12	0.5%	0.1
119	1	0.0%	1985	80.5%	0.8
120	1	0.0%	1985	80.5%	0.8
121	1	0.0%	1985	80.5%	0.8
122	1	0.0%	1	0.0%	0.0
123	1	0.0%	1985	80.5%	0.8
124	1	0.0%	1985	80.5%	0.8
125	1	0.0%	1985	80.5%	0.8

Proc. No.	Size (SLOC)	% of Total	Re-test Impact (SLOC)	% of Total	Contrib. to Mean
126	1	0.0%	1985	80.5%	0.8
127	1	0.0%	1985	80.5%	0.8
128	1	0.0%	13	0.5%	0.0
129	1	0.0%	1985	80.5%	0.8
130	1	0.0%	1985	80.5%	0.8
131	1	0.0%	1985	80.5%	0.8
132	1	0.0%	1985	80.5%	0.8
133	1	0.0%	1985	80.5%	0.8
134	1	0.0%	1985	80.5%	0.8
135	1	0.0%	1	0.0%	0.0
136	1	0.0%	1	0.0%	0.0
137	1	0.0%	1985	80.5%	0.8
138	1	0.0%	1985	80.5%	0.8
139	1	0.0%	1985	80.5%	0.8
140	1	0.0%	1985	80.5%	0.8
141	1	0.0%	1985	80.5%	0.8
142	1	0.0%	1985	80.5%	0.8
143	1	0.0%	1985	80.5%	0.8
144	15	0.6%	1985	80.5%	12.1
145	1	0.0%	1985	80.5%	0.8
146	1	0.0%	1985	80.5%	0.8
147	1	0.0%	1985	80.5%	0.8
148	1	0.0%	1985	80.5%	0.8
149	2	0.1%	1985	80.5%	1.6
150	6	0.2%	1985	80.5%	4.8
151	25	1.0%	318	12.9%	3.2
152	1	0.0%	1979	80.3%	0.8
153	37	1.5%	1978	80.2%	29.7
154	1	0.0%	1	0.0%	0.0
155	15	0.6%	1985	80.5%	12.1
156	1	0.0%	1985	80.5%	0.8
157	2	0.1%	2	0.1%	0.0
158	2	0.1%	383	15.5%	0.3
159	2	0.1%	157	6.4%	0.1
160	9	0.4%	1985	80.5%	7.2
161	25	1.0%	813	33.0%	8.2
162	8	0.3%	414	16.8%	1.3
163	8	0.3%	1978	80.2%	6.4
164	9	0.4%	1978	80.2%	7.2
165	4	0.2%	4	0.2%	0.0
166	4	0.2%	1989	80.7%	3.2
167	60	2.4%	1985	80.5%	48.3
168	38	1.5%	1985	80.5%	30.6
169	31	1.3%	1985	80.5%	25.0
170	9	0.4%	1985	80.5%	7.2
171	11	0.4%	1985	80.5%	8.9
172	21	0.9%	1985	80.5%	16.9
173	22	0.9%	1985	80.5%	17.7
174	58	2.4%	1985	80.5%	46.7
175	16	0.6%	1985	80.5%	12.9
176	16	0.6%	1985	80.5%	12.9

Proc. No.	Size (SLOC)	% of Total	Re-test Impact (SLOC)	% of Total	Contrib. to Mean
177	8	0.3%	1985	80.5%	6.4
178	6	0.2%	1985	80.5%	4.8
179	91	3.7%	1985	80.5%	73.3
180	2	0.1%	1985	80.5%	1.6
181	8	0.3%	1985	80.5%	6.4
182	6	0.2%	6	0.2%	0.0
183	6	0.2%	1985	80.5%	4.8
184	10	0.4%	1985	80.5%	8.0
185	6	0.2%	1985	80.5%	4.8
186	12	0.5%	1985	80.5%	9.7
187	2	0.1%	23	0.9%	0.0
188	21	0.9%	21	0.9%	0.2
189	11	0.4%	1992	80.8%	8.9
190	1	0.0%	1979	80.3%	0.8
191	1	0.0%	1	0.0%	0.0
192	1	0.0%	1979	80.3%	0.8
193	1	0.0%	1	0.0%	0.0
194	5	0.2%	1983	80.4%	4.0
195	6	0.2%	1985	80.5%	4.8
196	12	0.5%	1985	80.5%	9.7
197	3	0.1%	1985	80.5%	2.4
198	5	0.2%	1990	80.7%	4.0
199	2	0.1%	1985	80.5%	1.6
200	3	0.1%	1978	80.2%	2.4
201	3	0.1%	3	0.1%	0.0
202	9	0.4%	27	1.1%	0.1
203	3	0.1%	1985	80.5%	2.4
204	6	0.2%	202	8.2%	0.5
205	7	0.3%	1985	80.5%	5.6
206	10	0.4%	21	0.9%	0.1
207	17	0.7%	1985	80.5%	13.7
208	2	0.1%	1985	80.5%	1.6
209	5	0.2%	1983	80.4%	4.0
210	13	0.5%	13	0.5%	0.1
211	6	0.2%	1978	80.2%	4.8
212	15	0.6%	1978	80.2%	12.0
213	2	0.1%	11	0.4%	0.0
214	0	0.0%	0	0.0%	0.0
215	9	0.4%	292	11.8%	1.1
216	18	0.7%	2012	81.6%	14.7
217	1	0.0%	101	4.1%	0.0
218	1	0.0%	2013	81.6%	0.8
219	2	0.1%	616	25.0%	0.5
220	4	0.2%	618	25.1%	1.0
221	11	0.4%	618	25.1%	2.8
222	3	0.1%	614	24.9%	0.7
223	5	0.2%	24	1.0%	0.0
224	53	2.1%	2031	82.4%	43.7
225	123	5.0%	2104	85.3%	104.9
226	38	1.5%	2016	81.8%	31.1
227	143	5.8%	2121	86.0%	123.0

Proc. No.	Size (SLOC)	% of Total	Re-test Impact (SLOC)	% of Total	Contrib. to Mean
				TOTAL:	1782

7.2 After Refactoring

7.2.1 Procedure Characteristics

Proc. No.	Class	Method	Size (SLOC)
1	STL_Agent	STL_Agent	55
2	STL_Agent	run	68
3	STL_Agent	main	7
4	STL_Agent	setupProperties	18
5	STL_Agent	startLogger	5
6	STL_Agent	getRuntimeDir	1
7	STL_Agent	getConfigFileFullPath	1
8	STL_Agent	renameFile	5
9	STL_Agent	mergeVectors	6
10	STL_Agent	setupScafFiles	7
11	STL_Agent	initialize	8
12	STL_Database	STL_Database	26
13	STL_Database	create	3
14	STL_Database	getEndUser	3
15	STL_Database	getPartyList	8
16	STL_Database	getPreviousDays	22
17	STL_Database	getRetroactiveBusinessDaysForDate	25
18	STL_Database	getTransactionsForParty	15
19	STL_Database	isBusinessDay	9
20	STL_Database	updateSettlementStatus	29
21	STL_Database	getSageMapSupplierCode	8
22	STL_Database	getSageSrSupplierCode	0
23	STL_Database	close	1
24	STL_Database	updateEnduserEFT	11
25	STL_Database	isPartyInSync	13
26	STL_Database	synchronizeEndUser	39
27	STL_Database	assignRankingToEftStatus	11
28	STL_Database	getSystemTableParameter	9
29	STL_BusinessManager	STL_BusinessManager	6
30	STL_BusinessManager	buildPartySettlementList	3
31	STL_BusinessManager	buildTransactionSettlementList	25
32	STL_BusinessManager	buildPartyList	17
33	STL_BusinessManager	synchronizeAllEndusers	10
34	STL_BusinessManager	buildAUDDISPartyList	24
35	STL_BusinessManager	firstTransactionDelay	30
36	STL_BusinessManager	mergePartiesByEFTSTATUS	16
37	REP_ManagerTransactions	REP_ManagerTransactions	70
38	REP_ManagerTransactions	setupProperties	27
39	REP_ManagerTransactions	getManagerDetails	17
40	REP_ManagerTransactions	getMapVatRate	13
41	REP_ManagerTransactions	Deleted	0
42	REP_ManagerTransactions	generateCSVReport	41
43	REP_ManagerTransactions	getResults	21
44	REP_ManagerTransactions	createLine	2
45	REP_ManagerTransactions	displayNegative	3
46	REP_ManagerTransactions	getTransactionType	15
47	REP_ManagerTransactions	getServiceType	106

Proc. No.	Class	Method	Size (SLOC)
48	REP_ManagerTransactions	dump	6
49	REP_ManagerTransactions	getNonVatableAmount	8
50	REP_ManagerTransactions	getVatableAmount	8
51	REP_ManagerTransactions	getVAT	3
52	REP_ManagerTransactions	getVATRate	34
53	REP_ManagerTransactions	isMapTransaction	3
54	REP_ManagerTransactions	getAmount	1
55	REP_ManagerTransactions	sendMailMessage	22
56	REP_ManagerTransactions	startLogger	4
57	REP_ManagerTransactions	setupScafFiles	7
58	REP_ManagerTransactions	main	9
59	STL_DatabaseInterface	STL_DatabaseInterface	4
60	STL_DatabaseInterface	close	0
61	STL_DatabaseInterface	getEndUser	0
62	STL_DatabaseInterface	getPartyList	0
63	STL_DatabaseInterface	getRetroactiveBusinessDaysForDate	0
64	STL_DatabaseInterface	getTransactionsForParty	0
65	STL_DatabaseInterface	isBusinessDay	0
66	STL_DatabaseInterface	getPreviousDays	0
67	STL_DatabaseInterface	updateSettlementStatus	0
68	STL_DatabaseInterface	getSageMapSupplierCode	0
69	STL_DatabaseInterface	getSageSrSupplierCode	0
70	STL_DatabaseInterface	updateEnduserEFT	0
71	STL_DatabaseInterface	isPartyInSync	0
72	STL_DatabaseInterface	synchronizeEndUser	0
73	STL_DatabaseInterface	assignRankingToEftStatus	0
74	STL_DatabaseInterface	getSystemTableParameter	0
75	STL_EndUser	STL_EndUser	23
76	STL_EndUser	settlementAmount	1
77	STL_EndUser	totalFees	1
78	STL_EndUser	totalVat	1
79	STL_EndUser	creditCode	1
80	STL_EndUser	debitCode	1
81	STL_EndUser	eftStatus	1
82	STL_EndUser	eftStatusDate	1
83	STL_EndUser	testMode	3
84	STL_EndUser	invoiceEmail	1
85	STL_EndUser	Deleted	0
86	STL_EndUser	setEFTSTATUS	2
87	STL_EndUser	postCredit	6
88	STL_EndUser	Deleted	0
89	STL_EndUserFactory	STL_EndUserFactory	0
90	STL_EndUserFactory	create	19
91	STL_Party	STL_Party	33
92	STL_Party	creditCode	0
93	STL_Party	debitCode	0
94	STL_Party	settlementAmount	0
95	STL_Party	totalFees	0
96	STL_Party	totalVat	0
97	STL_Party	addTransaction	4
98	STL_Party	removeTransaction	5
99	STL_Party	addTransactionToList	1

Proc. No.	Class	Method	Size (SLOC)
100	STL_Party	removeTransactionFromList	1
101	STL_Party	getTransaction	6
102	STL_Party	getTransactions	1
103	STL_Party	getNumberOfTransactions	1
104	STL_Party	electronicSettlementFrequency	1
105	STL_Party	getSettlementGracePeriod	1
106	STL_Party	getBacsPayeeName	1
107	STL_Party	getPartyName	1
108	STL_Party	sortCode	1
109	STL_Party	accountNumber	1
110	STL_Party	referenceNumber	1
111	STL_Party	weeklySettlementDay	1
112	STL_Party	paymentMethod	1
113	STL_Party	getPartyId	1
114	STL_Party	toString	5
115	STL_Party	getPartyType	1
116	STL_Party	setPartyType	1
117	STL_Transaction	STL_Transaction	48
118	STL_Transaction	toString	12
119	STL_Transaction	domainTransactionId	1
120	STL_Transaction	transactionType	1
121	STL_Transaction	action	1
122	STL_Transaction	customerId	1
123	STL_Transaction	acctSysCustId	1
124	STL_Transaction	serviceRequestId	1
125	STL_Transaction	serviceRequestType	1
126	STL_Transaction	transactionDate	1
127	STL_Transaction	paymentMethod	1
128	STL_Transaction	currency	1
129	STL_Transaction	dataProviderFee	1
130	STL_Transaction	dataProviderVat	1
131	STL_Transaction	channelCharge	1
132	STL_Transaction	channelVat	1
133	STL_Transaction	hubCharge	1
134	STL_Transaction	hubVat	1
135	STL_Transaction	total	1
136	STL_Transaction	settlementDate	1
137	STL_Transaction	transactionMode	1
138	STL_Transaction	llc1	1
139	STL_Transaction	con29	1
140	STL_Transaction	cr21	1
141	STL_Transaction	setSettlementDate	1
142	STL_Transaction	sageSRSupplierCode	1
143	STL_Transaction	managerId	1
144	UTL_Mail	UTL_Mail	15
145	UTL_Mail	setFrom	1
146	UTL_Mail	setSubject	1
147	UTL_Mail	setMessageBody	1
148	UTL_Mail	setRecipients	1
149	UTL_Mail	addAttachment	2
150	UTL_Mail	reset	6
151	UTL_Mail	send	25

Proc. No.	Class	Method	Size (SLOC)
152	STL_TransactionFactory	STL_TransactionFactory	1
153	STL_TransactionFactory	create	37
154	STL_TransactionException	STL_TransactionException	1
155	UTL_DBConnection	UTL_DBConnection	15
156	UTL_DBConnection	getConnection	1
157	UTL_DBConnection	rollback	2
158	UTL_DBConnection	commit	2
159	UTL_DBConnection	close	2
160	UTL_DBConnection	executeQuery	9
161	UTL_DBConnection	setValues	25
162	UTL_DBConnection	executeUpdate	8
163	UTL_DBConnection	executeIdQuery	8
164	UTL_DBConnection	getLatestSequenceValue	9
165	UTL_DBConnection	zValue	4
166	STL_Reports	STL_Reports	4
167	STL_Reports	generateBacsTransactionFile	22
168	STL_Reports	generateSageTransactionFile	17
169	STL_Reports	createBacsSummary	31
170	STL_Reports	userHasNonManagedTransactions	9
171	STL_Reports	getUsersManagers	11
172	STL_Reports	createInvoices	23
173	STL_Reports	Deleted	0
174	STL_Reports	generateReport	40
175	STL_Reports	mailReport	16
176	STL_Transaction	statementTransactionType	16
177	STL_Transaction	statementParty	8
178	STL_Transaction	statementAction	6
179	STL_Transaction	statementServiceRequestType	91
180	STL_Transaction	statementDate	2
181	STL_Transaction	statementPaymentMethod	8
182	STL_Transaction	statementCurrency	6
183	STL_Transaction	statementTransactionMode	6
184	STL_Transaction	statementServiceDeliveryMechanism	10
185	STL_Reports	getSupplierCode	6
186	STL_Reports	Deleted	0
187	RPT_PrintDate	RPT_PrintDate	2
188	RPT_PrintDate	main	21
189	STL_Day	STL_Day	11
190	STL_Day	getDay	1
191	STL_Day	getDayOfWeek	1
192	STL_Day	isBusinessDay	1
193	STL_PartyException	STL_PartyException	1
194	STL_Util	STL_Util	5
195	STL_Util	doubleToString	6
196	STL_Util	justifyString	12
197	STL_Util	dateToString	3
198	STL_Util	stringToDate	5
199	STL_Util	buildFilenameString	2
200	STL_Util	prepJob	3
201	STL_Util	prepAttachment	3
202	STL_Util	ftpFile	9
203	UTL_CSVFormattedFile	UTL_CSVFormattedFile	3

Proc. No.	Class	Method	Size (SLOC)
204	UTL_CSVFormattedFile	addLine	6
205	UTL_FCFFormattedFile	UTL_FCFFormattedFile	7
206	UTL_FCFFormattedFile	addLine	10
207	UTL_Property	UTL_Property	17
208	UTL_Property	getPropertyValue	2
209	UTL_Property	setPropertyValue	5
210	UTL_Property	writeToDisk	13
211	UTL_Property	tokenize	6
212	UTL_Property	main	15
213	UTL_FormattedFile	UTL_FormattedFile	2
214	UTL_FormattedFile	addLine	0
215	UTL_FormattedFile	generateFile	9
216	UTL_Logger	UTL_Logger	18
217	UTL_Logger	setLogFile	1
218	UTL_Logger	setOutput	1
219	UTL_Logger	label	2
220	UTL_Logger	severityToInt	4
221	UTL_Logger	log	11
222	UTL_Logger	output	3
223	UTL_Logger	main	5
224	UTL_DBConstant	UTL_DBConstant	53
225	STL_Constants	STL_Constants	123
226	UTL_Constant	UTL_Constant	38
227	REP_Constants	REP_Constants	143
228	STL_Agent	proc228	9
229	STL_Agent	proc229	7
230	STL_Agent	proc230	2
231	STL_Database	proc231	8
232	STL_Database	proc232	7
233	STL_Reports	proc233	11
234	STL_EndUser	proc234	20
235	STL_EndUser	proc235	12
236	STL_EndUser	proc236	20
237	STL_Transaction	proc237	22
		TOTAL:	2392

7.2.2 Re-test Impact Calculation

Proc. No.	Size (SLOC)	% of Total	Rel. to Old SLOC	Re-test Impact (SLOC)	% of Total	Contrib. to Mean
1	55	2.3%	2.2%	1899	79.4%	43.7
2	68	2.8%	2.8%	1899	79.4%	54.0
3	7	0.3%	0.3%	1899	79.4%	5.6
4	18	0.8%	0.7%	1899	79.4%	14.3
5	5	0.2%	0.2%	1899	79.4%	4.0
6	1	0.0%	0.0%	1	0.0%	0.0
7	1	0.0%	0.0%	1899	79.4%	0.8
8	5	0.2%	0.2%	5	0.2%	0.0
9	6	0.3%	0.2%	6	0.3%	0.0
10	7	0.3%	0.3%	77	3.2%	0.2
11	8	0.3%	0.3%	1899	79.4%	6.4

Proc. No.	Size (SLOC)	% of Total	Rel. to Old SLOC	Re-test Impact (SLOC)	% of Total	Contrib. to Mean
12	26	1.1%	1.1%	1899	79.4%	20.6
13	3	0.1%	0.1%	1899	79.4%	2.4
14	3	0.1%	0.1%	1899	79.4%	2.4
15	8	0.3%	0.3%	1899	79.4%	6.4
16	22	0.9%	0.9%	1899	79.4%	17.5
17	25	1.0%	1.0%	1899	79.4%	19.8
18	15	0.6%	0.6%	1892	79.1%	11.9
19	9	0.4%	0.4%	1899	79.4%	7.1
20	29	1.2%	1.2%	1899	79.4%	23.0
21	8	0.3%	0.3%	1899	79.4%	6.4
22	0	0.0%	0.0%	0	0.0%	0.0
23	1	0.0%	0.0%	76	3.2%	0.0
24	11	0.5%	0.4%	308	12.9%	1.4
25	13	0.5%	0.5%	20	0.8%	0.1
26	39	1.6%	1.6%	250	10.5%	4.1
27	11	0.5%	0.4%	250	10.5%	1.1
28	9	0.4%	0.4%	1892	79.1%	7.1
29	6	0.3%	0.2%	1899	79.4%	4.8
30	3	0.1%	0.1%	1899	79.4%	2.4
31	25	1.0%	1.0%	1899	79.4%	19.8
32	17	0.7%	0.7%	1899	79.4%	13.5
33	10	0.4%	0.4%	1899	79.4%	7.9
34	24	1.0%	1.0%	1899	79.4%	19.1
35	30	1.3%	1.2%	1899	79.4%	23.8
36	16	0.7%	0.6%	1899	79.4%	12.7
37	70	2.9%	2.8%	1892	79.1%	55.4
38	27	1.1%	1.1%	1892	79.1%	21.4
39	17	0.7%	0.7%	1892	79.1%	13.4
40	13	0.5%	0.5%	1892	79.1%	10.3
41	0	0.0%	0.0%	0	0.0%	0.0
42	41	1.7%	1.7%	1892	79.1%	32.4
43	21	0.9%	0.9%	1892	79.1%	16.6
44	2	0.1%	0.1%	1892	79.1%	1.6
45	3	0.1%	0.1%	55	2.3%	0.1
46	15	0.6%	0.6%	1892	79.1%	11.9
47	106	4.4%	4.3%	1892	79.1%	83.8
48	6	0.3%	0.2%	85	3.6%	0.2
49	8	0.3%	0.3%	9	0.4%	0.0
50	8	0.3%	0.3%	1892	79.1%	6.3
51	3	0.1%	0.1%	4	0.2%	0.0
52	34	1.4%	1.4%	1892	79.1%	26.9
53	3	0.1%	0.1%	1892	79.1%	2.4
54	1	0.0%	0.0%	1	0.0%	0.0
55	22	0.9%	0.9%	1892	79.1%	17.4
56	4	0.2%	0.2%	1892	79.1%	3.2
57	7	0.3%	0.3%	86	3.6%	0.3
58	9	0.4%	0.4%	1892	79.1%	7.1
59	4	0.2%	0.2%	1896	79.3%	3.2
60	0	0.0%	0.0%	0	0.0%	0.0
61	0	0.0%	0.0%	0	0.0%	0.0
62	0	0.0%	0.0%	0	0.0%	0.0

Proc. No.	Size (SLOC)	% of Total	Rel. to Old SLOC	Re-test Impact (SLOC)	% of Total	Contrib. to Mean
63	0	0.0%	0.0%	0	0.0%	0.0
64	0	0.0%	0.0%	0	0.0%	0.0
65	0	0.0%	0.0%	0	0.0%	0.0
66	0	0.0%	0.0%	0	0.0%	0.0
67	0	0.0%	0.0%	0	0.0%	0.0
68	0	0.0%	0.0%	0	0.0%	0.0
69	0	0.0%	0.0%	0	0.0%	0.0
70	0	0.0%	0.0%	0	0.0%	0.0
71	0	0.0%	0.0%	0	0.0%	0.0
72	0	0.0%	0.0%	0	0.0%	0.0
73	0	0.0%	0.0%	0	0.0%	0.0
74	0	0.0%	0.0%	0	0.0%	0.0
75	23	1.0%	0.9%	1899	79.4%	18.3
76	1	0.0%	0.0%	1899	79.4%	0.8
77	1	0.0%	0.0%	1899	79.4%	0.8
78	1	0.0%	0.0%	1899	79.4%	0.8
79	1	0.0%	0.0%	1	0.0%	0.0
80	1	0.0%	0.0%	1	0.0%	0.0
81	1	0.0%	0.0%	1899	79.4%	0.8
82	1	0.0%	0.0%	1899	79.4%	0.8
83	3	0.1%	0.1%	1899	79.4%	2.4
84	1	0.0%	0.0%	1899	79.4%	0.8
85	0	0.0%	0.0%	0	0.0%	0.0
86	2	0.1%	0.1%	221	9.2%	0.2
87	6	0.3%	0.2%	1899	79.4%	4.8
88	0	0.0%	0.0%	1892	79.1%	0.0
89	0	0.0%	0.0%	1899	79.4%	0.0
90	19	0.8%	0.8%	1899	79.4%	15.1
91	33	1.4%	1.3%	1892	79.1%	26.1
92	0	0.0%	0.0%	1899	79.4%	0.0
93	0	0.0%	0.0%	1899	79.4%	0.0
94	0	0.0%	0.0%	1899	79.4%	0.0
95	0	0.0%	0.0%	1899	79.4%	0.0
96	0	0.0%	0.0%	1899	79.4%	0.0
97	4	0.2%	0.2%	1899	79.4%	3.2
98	5	0.2%	0.2%	1892	79.1%	4.0
99	1	0.0%	0.0%	1899	79.4%	0.8
100	1	0.0%	0.0%	1892	79.1%	0.8
101	6	0.3%	0.2%	1899	79.4%	4.8
102	1	0.0%	0.0%	1899	79.4%	0.8
103	1	0.0%	0.0%	1899	79.4%	0.8
104	1	0.0%	0.0%	1899	79.4%	0.8
105	1	0.0%	0.0%	1899	79.4%	0.8
106	1	0.0%	0.0%	1899	79.4%	0.8
107	1	0.0%	0.0%	1899	79.4%	0.8
108	1	0.0%	0.0%	306	12.8%	0.1
109	1	0.0%	0.0%	293	12.2%	0.1
110	1	0.0%	0.0%	136	5.7%	0.1
111	1	0.0%	0.0%	1	0.0%	0.0
112	1	0.0%	0.0%	1899	79.4%	0.8
113	1	0.0%	0.0%	1899	79.4%	0.8

Proc. No.	Size (SLOC)	% of Total	Rel. to Old SLOC	Re-test Impact (SLOC)	% of Total	Contrib. to Mean
114	5	0.2%	0.2%	5	0.2%	0.0
115	1	0.0%	0.0%	1899	79.4%	0.8
116	1	0.0%	0.0%	1893	79.1%	0.8
117	48	2.0%	1.9%	1892	79.1%	38.0
118	12	0.5%	0.5%	12	0.5%	0.1
119	1	0.0%	0.0%	1899	79.4%	0.8
120	1	0.0%	0.0%	1899	79.4%	0.8
121	1	0.0%	0.0%	1899	79.4%	0.8
122	1	0.0%	0.0%	1	0.0%	0.0
123	1	0.0%	0.0%	1899	79.4%	0.8
124	1	0.0%	0.0%	1899	79.4%	0.8
125	1	0.0%	0.0%	1899	79.4%	0.8
126	1	0.0%	0.0%	1899	79.4%	0.8
127	1	0.0%	0.0%	1899	79.4%	0.8
128	1	0.0%	0.0%	13	0.5%	0.0
129	1	0.0%	0.0%	1899	79.4%	0.8
130	1	0.0%	0.0%	1899	79.4%	0.8
131	1	0.0%	0.0%	1899	79.4%	0.8
132	1	0.0%	0.0%	1899	79.4%	0.8
133	1	0.0%	0.0%	1899	79.4%	0.8
134	1	0.0%	0.0%	1899	79.4%	0.8
135	1	0.0%	0.0%	1	0.0%	0.0
136	1	0.0%	0.0%	1	0.0%	0.0
137	1	0.0%	0.0%	1899	79.4%	0.8
138	1	0.0%	0.0%	1899	79.4%	0.8
139	1	0.0%	0.0%	1899	79.4%	0.8
140	1	0.0%	0.0%	1899	79.4%	0.8
141	1	0.0%	0.0%	1899	79.4%	0.8
142	1	0.0%	0.0%	1899	79.4%	0.8
143	1	0.0%	0.0%	1899	79.4%	0.8
144	15	0.6%	0.6%	1899	79.4%	11.9
145	1	0.0%	0.0%	1899	79.4%	0.8
146	1	0.0%	0.0%	1899	79.4%	0.8
147	1	0.0%	0.0%	1899	79.4%	0.8
148	1	0.0%	0.0%	1899	79.4%	0.8
149	2	0.1%	0.1%	1899	79.4%	1.6
150	6	0.3%	0.2%	1899	79.4%	4.8
151	25	1.0%	1.0%	262	11.0%	2.7
152	1	0.0%	0.0%	1893	79.1%	0.8
153	37	1.5%	1.5%	1892	79.1%	29.3
154	1	0.0%	0.0%	1	0.0%	0.0
155	15	0.6%	0.6%	1899	79.4%	11.9
156	1	0.0%	0.0%	1899	79.4%	0.8
157	2	0.1%	0.1%	2	0.1%	0.0
158	2	0.1%	0.1%	343	14.3%	0.3
159	2	0.1%	0.1%	128	5.4%	0.1
160	9	0.4%	0.4%	1899	79.4%	7.1
161	25	1.0%	1.0%	680	28.4%	7.1
162	8	0.3%	0.3%	382	16.0%	1.3
163	8	0.3%	0.3%	1892	79.1%	6.3
164	9	0.4%	0.4%	1892	79.1%	7.1

Proc. No.	Size (SLOC)	% of Total	Rel. to Old SLOC	Re-test Impact (SLOC)	% of Total	Contrib. to Mean
165	4	0.2%	0.2%	4	0.2%	0.0
166	4	0.2%	0.2%	1903	79.6%	3.2
167	22	0.9%	0.9%	1899	79.4%	17.5
168	17	0.7%	0.7%	1899	79.4%	13.5
169	31	1.3%	1.3%	1899	79.4%	24.6
170	9	0.4%	0.4%	1899	79.4%	7.1
171	11	0.5%	0.4%	1899	79.4%	8.7
172	23	1.0%	0.9%	1899	79.4%	18.3
173	0	0.0%	0.0%	1892	79.1%	0.0
174	40	1.7%	1.6%	1899	79.4%	31.8
175	16	0.7%	0.6%	1899	79.4%	12.7
176	16	0.7%	0.6%	1899	79.4%	12.7
177	8	0.3%	0.3%	1899	79.4%	6.4
178	6	0.3%	0.2%	1899	79.4%	4.8
179	91	3.8%	3.7%	1899	79.4%	72.2
180	2	0.1%	0.1%	1899	79.4%	1.6
181	8	0.3%	0.3%	1899	79.4%	6.4
182	6	0.3%	0.2%	6	0.3%	0.0
183	6	0.3%	0.2%	1899	79.4%	4.8
184	10	0.4%	0.4%	1899	79.4%	7.9
185	6	0.3%	0.2%	1899	79.4%	4.8
186	0	0.0%	0.0%	0	0.0%	0.0
187	2	0.1%	0.1%	23	1.0%	0.0
188	21	0.9%	0.9%	21	0.9%	0.2
189	11	0.5%	0.4%	1906	79.7%	8.8
190	1	0.0%	0.0%	1900	79.4%	0.8
191	1	0.0%	0.0%	1	0.0%	0.0
192	1	0.0%	0.0%	1900	79.4%	0.8
193	1	0.0%	0.0%	1	0.0%	0.0
194	5	0.2%	0.2%	1897	79.3%	4.0
195	6	0.3%	0.2%	1899	79.4%	4.8
196	12	0.5%	0.5%	1899	79.4%	9.5
197	3	0.1%	0.1%	1899	79.4%	2.4
198	5	0.2%	0.2%	1904	79.6%	4.0
199	2	0.1%	0.1%	1899	79.4%	1.6
200	3	0.1%	0.1%	1892	79.1%	2.4
201	3	0.1%	0.1%	3	0.1%	0.0
202	9	0.4%	0.4%	1901	79.5%	7.2
203	3	0.1%	0.1%	1899	79.4%	2.4
204	6	0.3%	0.2%	173	7.2%	0.4
205	7	0.3%	0.3%	1899	79.4%	5.6
206	10	0.4%	0.4%	21	0.9%	0.1
207	17	0.7%	0.7%	1899	79.4%	13.5
208	2	0.1%	0.1%	1899	79.4%	1.6
209	5	0.2%	0.2%	1897	79.3%	4.0
210	13	0.5%	0.5%	13	0.5%	0.1
211	6	0.3%	0.2%	1892	79.1%	4.7
212	15	0.6%	0.6%	1892	79.1%	11.9
213	2	0.1%	0.1%	11	0.5%	0.0
214	0	0.0%	0.0%	0	0.0%	0.0
215	9	0.4%	0.4%	204	8.5%	0.8

Proc. No.	Size (SLOC)	% of Total	Rel. to Old SLOC	Re-test Impact (SLOC)	% of Total	Contrib. to Mean
216	18	0.8%	0.7%	1926	80.5%	14.5
217	1	0.0%	0.0%	101	4.2%	0.0
218	1	0.0%	0.0%	1927	80.6%	0.8
219	2	0.1%	0.1%	530	22.2%	0.4
220	4	0.2%	0.2%	1913	80.0%	3.2
221	11	0.5%	0.4%	1913	80.0%	8.8
222	3	0.1%	0.1%	528	22.1%	0.7
223	5	0.2%	0.2%	1897	79.3%	4.0
224	53	2.2%	2.1%	1945	81.3%	43.1
225	123	5.1%	5.0%	2018	84.4%	103.8
226	38	1.6%	1.5%	1930	80.7%	30.7
227	143	6.0%	5.8%	2035	85.1%	121.7
228	9	0.4%	0.4%	1899	79.4%	7.1
229	7	0.3%	0.3%	1899	79.4%	5.6
230	2	0.1%	0.1%	1899	79.4%	1.6
231	8	0.3%	0.3%	1899	79.4%	6.4
232	7	0.3%	0.3%	341	14.3%	1.0
233	11	0.5%	0.4%	1892	79.1%	8.7
234	20	0.8%	0.8%	135	5.6%	1.1
235	12	0.5%	0.5%	1911	79.9%	9.6
236	20	0.8%	0.8%	1899	79.4%	15.9
237	22	0.9%	0.9%	1899	79.4%	17.5
					TOTAL:	1678