Understanding the Economics of Refactoring

Rob Leitch
MacDonald, Dettwiler and Associates, Ltd.
13800 Commerce Parkway
Richmond, BC, V6V 2J3, Canada
1 (604) 231 2184
rleitch@mda.ca

Eleni Stroulia
Department of Computing Science
221 Athabasca Hall, University of Alberta
Edmonton, AB, T6G 2E8, Canada
1 (780) 492 3520
stroulia@cs.ualberta.ca

ABSTRACT

In this paper we discuss a novel method for estimating the expected maintenance savings given a refactoring plan. This work is motivated by the increased adoption of refactoring practices as part of new agile methodologies and the lack of any prescriptive theory on when to refactor.

1. INTRODUCTION AND MOTIVATION

Estimating the cost of future maintenance activities on a working application is an important research question. If such an estimate were possible, the guesswork would be eliminated from the decision of whether to maintain or replace existing software. There is some evidence in the literature [1] [4] that perfective maintenance accounts for the majority of the overall maintenance effort in a project. Perfective maintenance activities aim to improve the quality attributes of the software, such as its performance or its maintainability. Therefore the problem of "maintenance cost prediction" can be recast as "perfective maintenance cost prediction".

A long-standing method in support of perfective maintenance is local source code transformation, more recently re-discovered as "Refactoring" [3]. Although there are many tools developed to support code transformations there is no general agreement on what transformations are beneficial and when these changes should be applied. For example, the refactoring catalog contains "symmetrical" refactorings, i.e., opposite transformations such as "extract method" and "inline method". In addition, there are alternative refactorings applicable to similar low-level designs, such as "extract subclass" and "extract interface". It is up to developers to decide which type of refactoring to apply in anticipation of future development. Furthermore, currently there is only informal advice on when to refactor. Fowler [3] suggests that refactoring may not be beneficial when there is a deadline coming up or when the software is of such poor quality that it would be easier to re-develop it from scratch. This advice implies an estimate of the cost of refactoring vs. the cost of redevelopment. However, no such cost-estimate model exists. In spite of the lack of strong prescribed methodology, most popular agile methods advocate refactoring as a regular practice in the software lifecycle. This practice is becoming widely adopted as the method of choice for improving the extendibility and maintainability of software.

In our recent work, we have been investigating several aspects of refactoring. These aspects include understanding the impact of long-term code transformations on the quality of software design and the nature of developing a cost-benefit model estimating the tradeoff between the up-front cost of refactoring and the expected downstream maintenance savings. Specifically, we are interested in predicting the Return on investment (ROI) for a planned refactoring activity.

> ROI = (Maintenance Savings from Proposed Refactoring) / (Development Cost of Planned Refactoring). (1)

If the ROI is greater than or equal to one, then the planned refactoring will be cost effective.

2. ESTIMATING THE REFACTORING ROI

To calculate the Refactoring ROI according to formula (1) above, we need to estimate

1. the development cost of the planned refactoring activity, and
2. the anticipated maintenance cost of each of the two software versions (i.e., before and after refactoring).

We adopt COCOMO [2] to calculate the refactoring-plan development cost, and we propose a novel method for predicting the maintenance effort for the original and restructured designs.

A fairly common approach to this problem has been to try and relate design metrics to observed maintenance costs through regression analysis. However, while metrics can be used to identify outlier design components and to comparatively evaluate alternative designs, there are currently no suitable predictive models of 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 evidence that the perfective effort category accounts for the majority of maintenance cost [1] [4]. 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 no general agreement regarding which metrics can predict system fault density.

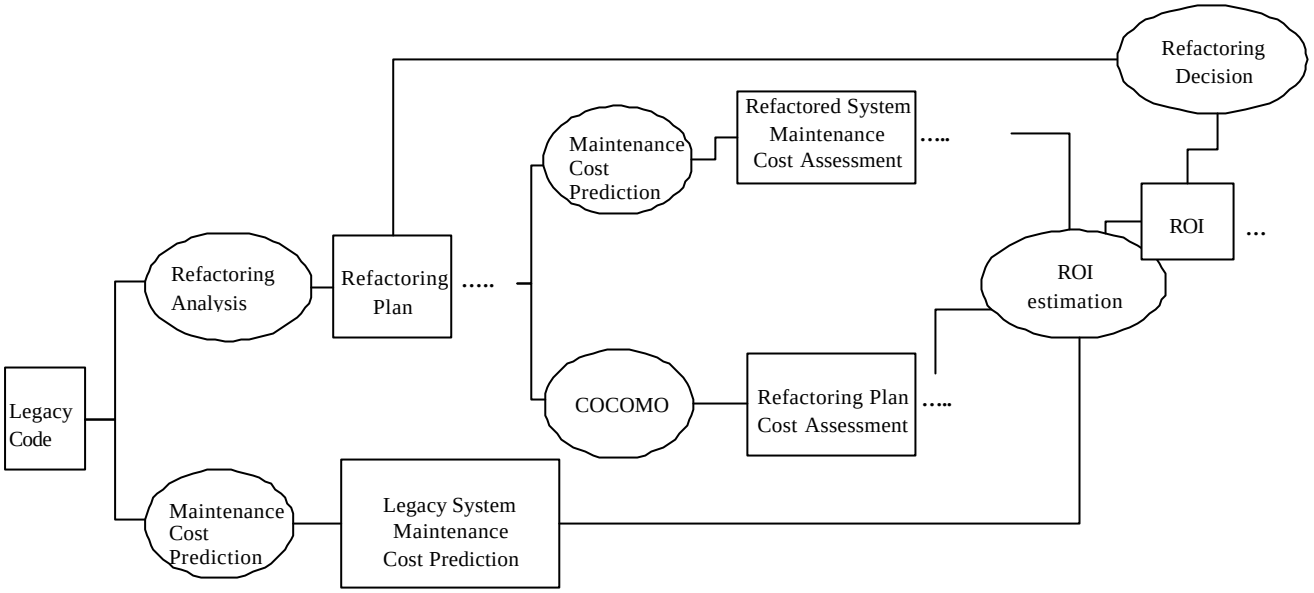


Figure 1: Informed Refactoring Decision Making, using Refactoring ROI estimates.

In our work we have been experimenting with an alternative strategy for predicting maintenance cost. This strategy is based on the following assumptions.

The anticipated future maintenance cost of a given software system is the sum of the costs of each individual future maintenance request.

Maintenance activities occur randomly in the software system, as modifications necessitated by new requirements on the software system. Therefore the probability that a maintenance request will strike an individual module is directly proportional to the size of the module relative to the size of the overall system.

A substantial part of the cost of any single modification is the cost of the regression testing necessitated after the modification is completed. We assume that the regression-testing savings brought about by refactoring can be substantial enough to bring the ROI fraction above 1. With these assumptions we can restate formula (1) as follows:

The regression-testing cost of a particular modification is directly proportional to the amount of code that has to be examined as a result of the change. This in turn can be estimated based on the dependencies of the modified module with the rest of the system.

ROI = (Regression-Testing Savings from Proposed Refactoring) / (Development Cost of Planned Refactoring). (2)

The ROI estimation process implied by these assumptions, as well as the informed refactoring decision-making process it enables, are depicted in Figure 1. Given a legacy system, its expected regression-testing cost is first calculated based on the occurrence of a random maintenance activity. Next, a number of alternative refactoring plans can be formulated and their respective development costs estimated using COCOMO. The predicted regression-testing costs of the proposed new designs are then calculated. At this point, the ROI of each alternative refactoring plan can be computed. These estimates are then used to decide whether refactoring the system is beneficial, and what sort of refactoring plan should be implemented.

3. EXPLORATORY CASE STUDY

Let us now illustrate our ROI estimation method with an exploratory case study using a simple Java system. The trial system was created in a student environment as part of a graduate course in Object-oriented (OO) 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 four-way intersection, including a graphical simulation of the intersection operation. We refer to this system as “TrafficApp”.

For the purposes 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 [3]. Using this definition, the trial case study program contains 740 SLOC, broken down into 6 classes and 29 procedures. Table 1 shows the distribution of code and procedures within TrafficApp.

3.1. Refactoring Plan

A source code walkthrough was performed on TrafficApp to identify candidate refactoring opportunities according to the criteria defined in [3]. The result of this walkthrough is a list of suggested refactorings presented in Table 4, along with the projected source code impact for each affected procedure. The data in Table 4

indicates the amount of code to be added or deleted for each procedure. Note that the recommended restructuring will add new procedures to the system. Table 1 shows the predicted impact of the restructuring at the class level, including changes in code size and the number of procedures.

Table 1: Code and procedures in TrafficApp.

	Size (SLOC)		
Class	**Before**	**After**	**Change**
Class 1	411	343	-17%
Class 2	164	81	-51%
Class 3	95	108	14%
Class 4	23	23	0%
Class 5	23	23	0%
Class 6	24	24	0%
TOTAL	740	602	-19%
	No. of Proc.		
Class	**Before**	**After**	**Change**
Class 1	4	11	175%
Class 2	7	8	14%
Class 3	6	7	17%
Class 4	5	5	0%
Class 5	2	2	0%
Class 6	5	5	0%
TOTAL	29	38	31%
	Avg. Proc. Size (SLOC)		
Class	**Before**	**After**	**Change**
Class 1	103	31	-70%
Class 2	23	10	-57%
Class 3	16	15	-3%
Class 4	5	5	0%
Class 5	12	12	0%
Class 6	5	5	0%
TOTAL	26	16	-38%

3.2 Impact on the Dependency Structure

Two sets of data and control dependency graphs were constructed for TrafficApp. One set of graphs represents the system state before refactoring (based on a manual code inspection). The second set of graphs represents the predicted state of TrafficApp after refactoring. The plot of Figure 2 illustrates the difference between these sets of graphs, showing changes in the dependency structure of the system resulting from the proposed restructuring.

3.3. Mean Re-test Impact

Table 5 shows the calculation of the mean re-test impact for TrafficApp before and after restructuring based on the overall dependency graphs and the source code distribution in the system. The mean re-test impact before refactoring is 408 SLOC, while the predicted mean re-test impact after refactoring is 216 SLOC.

3.4. Effort Calculations

Table 2 summarizes the example calculations performed using the COCOMOII.2000 model to predict maintenance costs before and after refactoring as well as the cost of the restructuring. As well, this table presents the COCOMO re-use model parameters assumed for TrafficApp. The net result is a predicted savings of 0.225 person-months per maintenance activity as a result of the proposed restructuring. This compares with a restructuring cost of 1.18 person-months.

Table 2: COCOMOII.2000 cost predictions for TrafficApp.

Parameter	Refact. Cost	Maint. Cost Before	Maint. Cost After	Maint. Savings
Size (KSLOC)	0.740	0.740	0.602	-
EAF	1.000	1.000	1.000	-
Scale Factor	18.970	18.970	18.970	-
Exponent	1.100	1.100	1.100	-
SU	30.000	30.000	30.000	-
AA	4.000	4.000	4.000	-
UNFM	0.400	0.400	0.400	-
DM	13.800	5.000	5.000	-
CM	29.700	5.000	5.000	-
IM	100.000	55.100	35.900	-
Equiv. KSLOC	0.437	0.213	0.131	-
Effort (p-months)	1.180	0.538	0.313	0.225

3.5. ROI Calculation

From Table 2, we can see that the ROI will be greater than one if there are greater than or equal to six maintenance activities after the design restructuring. This is determined by dividing the refactoring cost by the maintenance savings per activity (=1.18/0.225=5.2).

3.6. Results Discussion

Table 3 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 lower than for the original design.

Table 3: Dependency graphs before and after refactoring.

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

In Figure 3, 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 3, the code size reference is the original, unchanged version of TrafficApp.

From the above analysis, 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 (using the Extract Method and Move Method transformations defined in [3]).

Table 4: Proposed refactoring plan and design impact for TrafficApp.

Proc. No.	Code Problems	Refactoring	Add.	Del.	Proc. No.	Code Problems	Refactoring	Add.	Del.
1	Long Method, Duplicated Code, Feature Envy	Extract Method	24	225	33	N/A (new proc.)	Extract Method	27	0
2	Duplicated Code	Extract Method	4	28	34	N/A (new proc.)	Extract Method	81	0
10	Switch Statement, Duplicated Code, Feature Envy	Move Method	4	49	35	N/A (new proc.)	Extract Method	17	0
11	Long Method, Switch Statement, Duplicated Code	Extract Method	4	56	36	N/A (new proc.)	Extract Method	9	0
30	N/A (new proc.)	Extract Method	4	0	37	N/A (new proc.)	Move Method	13	0
31	N/A (new proc.)	Extract Method	9	0	38	N/A (new proc.)	Extract Method	14	0
32	N/A (new proc.)	Extract Method	10	0	-	-	-	-	-
		SUBTOTAL:	59	358			SUBTOTAL:	161	0
							TOTAL:	220	358

Table 5: Mean re-test impact before and after restructuring.

		BEFORE REFACTORING					AFTER REFACTORING			
Class No.	Proc. No.	Size (SLOC)	Test Impact (SLOC)	Prob.	Mean (SLOC)	Proc. No.	Size (SLOC)	Test Impact (SLOC)	Prob.	Mean (SLOC)
Class 1	1	306	677	41.4%	279.9	1	105	539	17.4%	94.0
	2	80	106	10.8%	11.5	2	56	91	9.3%	8.5
	3	18	84	2.4%	2.0	3	18	32	3.0%	1.0
	4	7	7	0.9%	0.1	4	7	7	1.2%	0.1
	-	-	-	-	-	30	4	116	0.7%	0.8
	-	-	-	-	-	31	9	121	1.5%	1.8
	-	-	-	-	-	32	10	122	1.7%	2.0
	-	-	-	-	-	33	27	139	4.5%	6.2
	-	-	-	-	-	34	81	193	13.5%	26.0
	-	-	-	-	-	35	17	129	2.8%	3.6
	-	-	-	-	-	36	9	91	1.5%	1.4
Class 2	5	27	530	3.6%	19.3	5	27	228	4.5%	10.2
	6	3	203	0.4%	0.8	6	3	141	0.5%	0.7
	7	1	201	0.1%	0.3	7	1	139	0.2%	0.2
	8	3	120	0.4%	0.5	8	3	64	0.5%	0.3
	9	15	99	2.0%	2.0	9	15	61	2.5%	1.5
	10	49	133	6.6%	8.8	10	4	18	0.7%	0.1
	11	66	120	8.9%	10.7	11	14	82	2.3%	1.9
	-	-	-	-	-	38	14	61	2.3%	1.4
Class 3	12	29	677	3.9%	26.5	12	29	539	4.8%	26.0
	13	7	447	0.9%	4.2	13	7	222	1.2%	2.6
	14	18	562	2.4%	13.7	14	18	253	3.0%	7.6
	15	18	99	2.4%	2.4	15	18	61	3.0%	1.8
	16	1	188	0.1%	0.3	16	1	97	0.2%	0.2
	17	22	106	3.0%	3.2	17	22	54	3.7%	2.0
	-	-	-	-	-	37	13	49	2.2%	1.1
Class 4	18	9	357	1.2%	4.3	18	9	156	1.5%	2.3
	19	3	605	0.4%	2.5	19	3	310	0.5%	1.5
	20	3	605	0.4%	2.5	20	3	310	0.5%	1.5
	21	1	1	0.1%	0.0	21	1	1	0.2%	0.0
	22	7	614	0.9%	5.8	22	7	319	1.2%	3.7
Class 5	23	3	336	0.4%	1.4	23	3	135	0.5%	0.7
	24	20	20	2.7%	0.5	24	20	20	3.3%	0.7
Class 6	25	7	321	0.9%	3.0	25	7	120	1.2%	1.4
	26	5	111	0.7%	0.8	26	5	96	0.8%	0.8
	27	1	401	0.1%	0.5	27	1	176	0.2%	0.3
	28	1	1	0.1%	0.0	28	1	1	0.2%	0.0
	29	10	11	1.4%	0.1	29	10	11	1.7%	0.2
		MEAN RE-TEST IMPACT:			408		MEAN RE-TEST IMPACT:			216

Figure 2: The changes in the dependency structure of TrafficApp before and after the proposed restructuring.

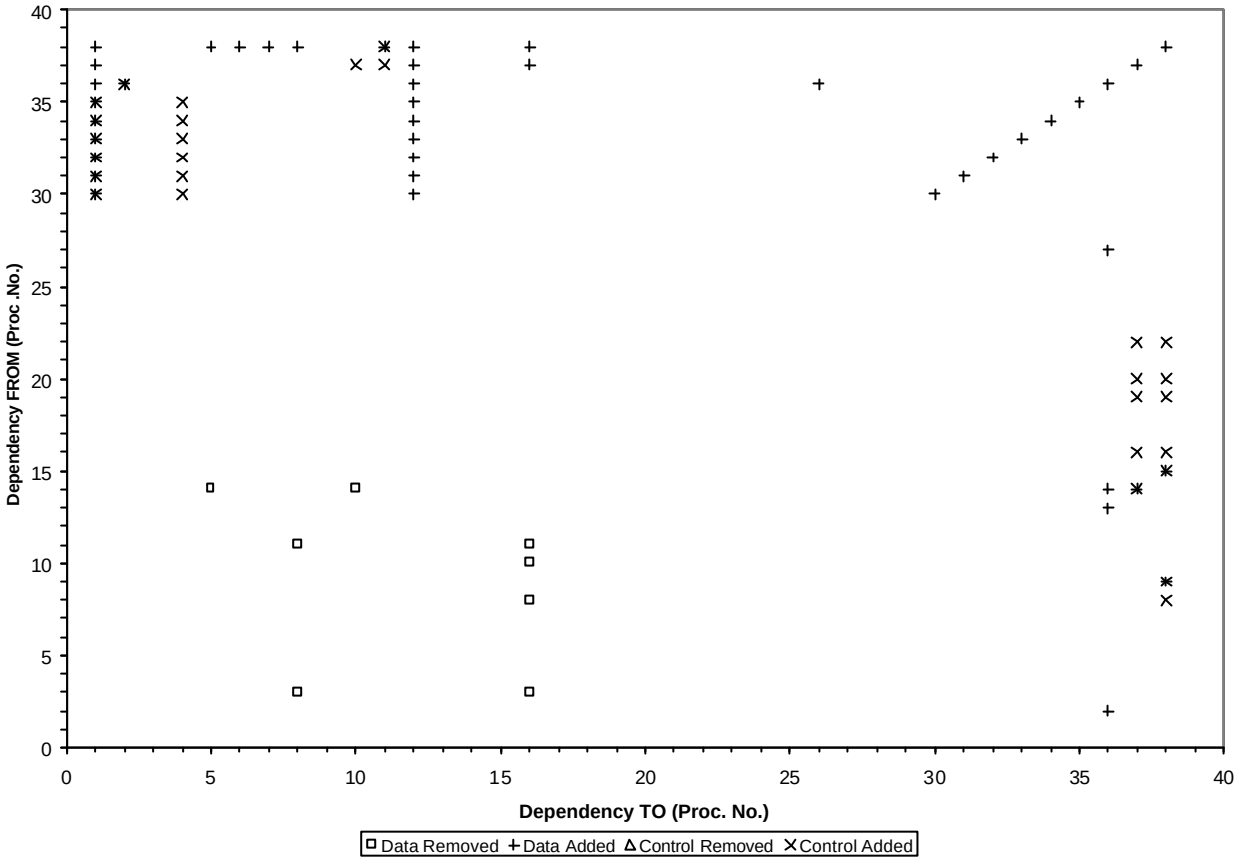
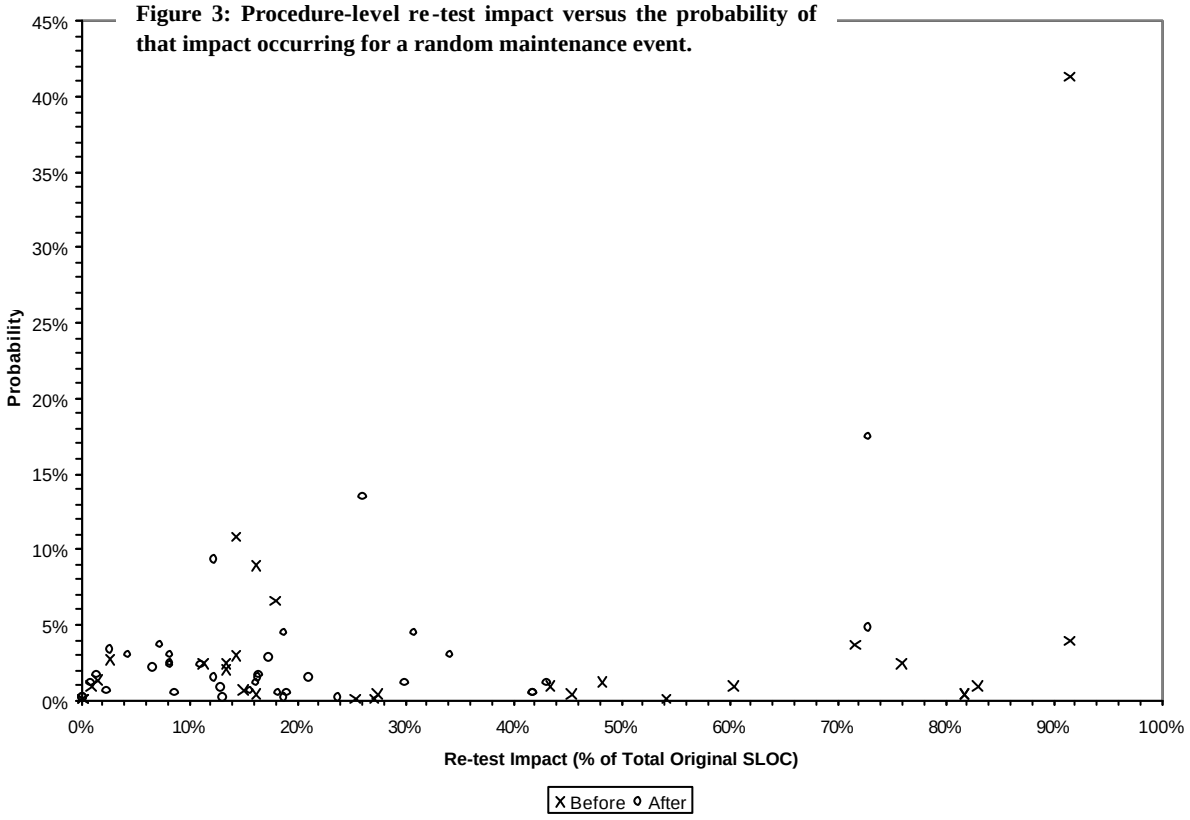


Figure 3: Procedure-level re-test impact versus the probability of that impact occurring for a random maintenance event.



The refactoring appears to reduce the peaks along both axes of the impact probability distribution of Figure 3. 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% due to 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.

4. DISCUSSION

This work is at a very early stage, however we have applied our refactoring ROI method to 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 estimated 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 can change the system dependency structure in a beneficial way, allowing recovery of the initial refactoring investment over a number of maintenance activities simply on the basis of regression-testing savings.

This early experience has generated several interesting “research leads” that we would like to pursue. For example, it would be interesting to explore other metrics to estimate maintenance benefits in addition to examining regression-testing costs. Furthermore, we are currently estimating regression-testing cost based on procedure-level dependencies; would other more or less precise metrics be better predictors? Could the refactoring decision-making process be influenced by previous refactorings applied to the system, i.e., can some refactorings preclude other refactorings from occurring in the future?

5. REFERENCES

[1] 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.

[2] Boehm, B., Horowitz, E., Madachy, R., Reifer, D., Clark, B.D., Steece, B., Brown, A.W., Chulani, S., and Abts, C., Software Cost Estimation with COCOMO II, Prentice Hall PTR, 2000.

[3] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D., Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[4] Polo, M., Piattini, M., Ruiz, F. Using code metrics to predict maintenance of legacy programs: a case study, IEEE ICSM 2001, Florence, Italy.