# Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation

James R. Cordy

*School of Computing, Queen's University*
*Kingston, Ontario, Canada K7L 3N6*
cordy@cs.queensu.ca

## Abstract

*Recent years have seen many significant advances in program comprehension and software maintenance automation technology. In spite of the enormous potential savings in software maintenance costs, for the most part adoption of these ideas in industry remains at the experimental prototype stage. In this paper I explore some of the practical reasons for industrial resistance to adoption of software maintenance automation. Based on the experience of six years of software maintenance automation services to the financial industry involving more than 4.5 Gloc of code at Legasys Corporation, I discuss some of the social, technical and business realities that lie at the root of this resistance, outline various Legasys attempts overcome these barriers, and suggest some approaches to software maintenance automation that may lead to higher levels of industrial acceptance in the future.*

## 1. Introduction

As evidenced by the prolific proceedings of ICSM, IWPC, WCRE and other program comprehension and software maintenance research venues, recent years have seen many advances in program comprehension and software maintenance automation technology. New techniques in impact analysis [1,2,3,4,5,6,7], architecture analysis and repair [8,9,10,11], clone detection and removal [12,13,14,15,16,17,18], program slicing and aspect analysis [19,20,21] and software refactoring [22,23] have the potential to significantly impact industrial software maintenance practice.

Experiments in the practical application of these techniques by Holt et al [24,25], Müller et al [26,27,28], Klint et al [29,30,31], Sneed [32,33] and others have shown that they are practical and useful in industry. Moreover, the success of companies such as Legasys Corporation, Formal Systems, Semantic Designs and others using design recovery and other formal analysis techniques to attack the Year 2000 ("Y2K") problem have demonstrated that large productivity gains and cost reductions can be obtained when software maintenance is assisted by program comprehension. The largest client using the techniques of Legasys Corporation, for example, reported a 40-fold increase in total productivity (including identification, conversion and test time) over in-house conversion when using design recovery and analysis to identify and systematically reprogram Y2K risks.

Nevertheless, even after this huge example demonstrating the potential and capability of program comprehension techniques to increase software maintenance productivity and decrease costs virtually worldwide, companies have been slow to realize and adopt them in practice. Almost all of the program comprehension-based software maintenance automation companies that were successful in the Y2K problem have been unable to draw enough ongoing business to continue, and the handful that are still around are for the most part struggling.

This paper is aimed at analyzing and reminding us of some of the causes of this lack of adoption, and at identifying some of the practical barriers that lie in the way. Technical barriers, such as parsing problems [34] have already been well covered in the literature. In this paper I will instead concentrate on an analysis of the social, cultural, economic and technological issues such as business risk, budgets and management structure.

The analysis is based on six years of service to a particular industry, the Canadian financial industry, at Legasys Corporation. The observations I make do not necessarily translate well to other segments of the software industry, but I am certain that they apply broadly in the financial and retail segments. Nevertheless, as they say in the United States, "your mileage may vary".

I apologize that this is not by any means a technical paper, although I will present some technical results from Legasys to demonstrate how some of the barriers can be addressed. The observations and opinions expressed here are my own, and even my former colleagues at Legasys may not agree with my interpretations and conclusions about our experience. Also, for reasons of confidentiality, I will not be associating the sources of our observations with particular clients or companies, and I will have to avoid concrete examples.

This is also not a formal anthropological, cultural or technology adoption study, and it shouldn't be taken as such. I have simply taken the liberty of using this keynote to pass on some lessons of personal experience that I speak of often, but don't normally have the opportunity to publish. I hope that you find the observations interesting and useful, and I hope that

*This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).*

awareness of them may help make the results of our research more readily acceptable to the industrial community.

The rest of this paper is structured into three parts. In the first part we begin by setting the scene, describing the Legasys environment and the characteristics of the clients and financial software systems on which these observations are based. Part 2 then outlines four realities, risk, technical, social and financial, that influence the adoption of software maintenance automation in the financial software industry and gives examples of how each of these realities can affect the adoption of particular program comprehension technologies. Finally, we summarize what may be learned and suggest some ways of approaching program comprehension and software maintenance automation research that may help increase industrial adoption of our methods in future.

## 2. Background

Legasys Corporation was founded in 1995 with the explicit mission of "practical industrial application of formal methods in software maintenance automation". While it was not founded with the intention of attacking the Y2K problem, within a year of its founding the majority of its effort was directed at Y2K, and over the six years of its active existence, Legasys was involved in the analysis and reprogramming of over 3.5 Gloc of Y2K conversions and another 1.5 Gloc of custom analysis for other maintenance activities such as platform and language migrations, software system mergers, code security audits and other large scale software maintenance tasks.

Legasys' technology was deeply based in design recovery and source transformation techniques. The main service provided by Legasys was an almost entirely automated Y2K analysis and conversion technology called LS/2000 [35], and a generalization of it to arbitrary impact analysis and migration tasks called LS/AMT (Figure 1).

The vast majority of Legasys clients were in the financial industry, including some of the largest Canadian national banks and insurance companies. Other clients included large retail chains, various government departments in Canada and the United States, airlines, health organizations and others. Virtually all of the client software systems we processed were involved in financial data processing in one form or another, and that is the context of the observations presented in this paper.

## 3. The Financial Industry

According to the Gartner Group, financial data processing is by far the largest segment of the software industry. Gartner estimates that there are over 200 Gloc of COBOL code in 9.5 million applications being actively maintained at present. This is about 60% of the world's total code base and represents an investment of over five trillion U.S. dollars. Moreover, contrary to the usual intuition, this number is actually growing rather than shrinking, due to increased economic activity and new banking initiatives such as debit cards, smart cards and so on. Gartner says the number of lines of COBOL code is presently growing by 5 Gloc per year.

In order to understand the realities of the financial software environment, it is necessary to understand some basic facts about the business of financial systems. Financial software systems such as those in a large bank are typically huge, involving hundreds of separate but tightly coupled software applications, each of which is between 100,000 and 1 million or more lines of source code. The total lines of code for a large bank or retail organization may run upwards of 500 Mloc, and may be maintained by a stable of 500 to 1,000 programmers.

Applications may be run independently in overnight, weekly or monthly batch runs, or they may be run continuously in interactive environments such as retail banking or point of sale systems. Most applications are run on large mainframe

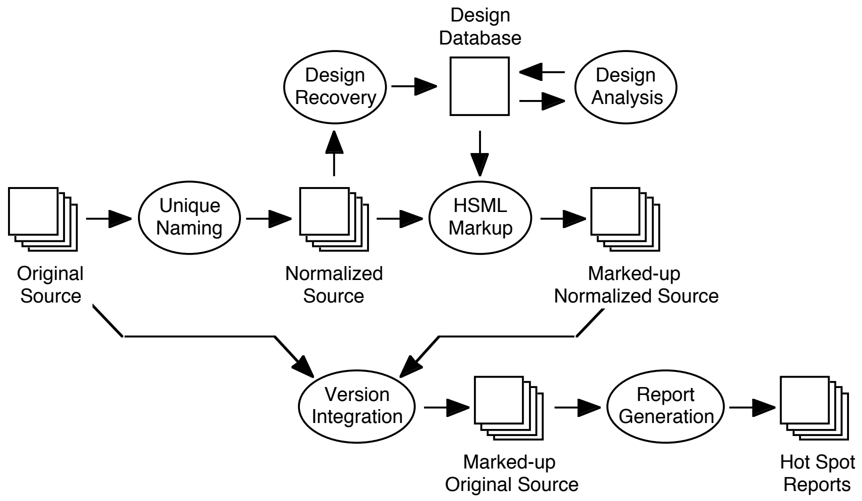

**Figure** 1. LS/AMT Process Architecture.

computers or on mid-sized business computers such as IBM's AS/400 series. Interactive applications typically interact using separate front ends that run on PC's, "green screen" text terminals or web interfaces in a wide-area networked environment.

The quality requirements for financial software systems are extremely high. Consequences of a small software error can run into the many millions of dollars in a single day, and even a small amount of downtime will cost many millions in missed transactions, lost customers and wasted employee hours. For this reason, the vast majority of the software maintenance effort in the industry is spent in systematic testing and validation, and banking systems are among the most thoroughly tested systems in the world.

The financial industry is also extremely competitive. Given that every bank offers essentially the same set of services, each is constantly looking for new services and conveniences that can be offered to customers to distinguish their bank from the others. The result is a continuous stream of new financial products and services such as different kinds of cards, travel points and so on. In order to remain competitive, every bank must react to every other bank's new services by offering something similar - this causes a continuous pressure to rapidly change and update the affected software systems.

Another source of pressure for change is government policy. All financial software systems are affected by changes in laws governing various kinds of taxation of investment, employees, transactions, and so on. But government policy also continually forces other changes such as changes in exchange and currency policy and denominations (e.g., the Euro) and most recently, more detailed recording of transactions and money flow for security reasons. The result of the combination of these two forces, competition and government, means that financial software systems are undergoing rapid, continuous and significant enhancement in an environment where quality constraints are very high.

## 4. Kinds of Reality

In the context of the environment described above, we identify four separate realities that place barriers in the way of adoption of program comprehension and software maintenance automation techniques - the reality of business risk, the reality of technology, the reality of social environment and the reality of economics, In the following sections we examine each of these realities in turn, observe how it can affect the adoption of software maintenance automation, and give concrete examples of program comprehension tasks that may be affected by each.

### 4.1 The Reality of Business Risk

The stringent quality environment and enormous consequences of software failure mean that the primary consideration when planning and executing software maintenance tasks is not cost, but rather risk. The risk factor is so important that Legasys clients reported that more than 70% of their total software maintenance effort is spent in extensive testing, including regression test runs of actual transaction data often over several days to insure that no existing behavior has been accidentally changed by a maintenance step.

In general, the cost of a designing and making a software change itself is small compared to the cost of quality control (testing), and miniscule compared to the potential cost of error. In this environment, every software change is considered in the context of the risk it poses that the software system may be broken by the change. This leads to a very conservative maintenance strategy - since the risk of introducing an error is roughly proportional to the number or extent of the changes made, systems are changed as little as possible when maintained.

When viewed in this way, older “legacy” code, which typically has been running successfully for many years, has been tuned to be free of error by actual experience, and exhibits an extremely high observed quality, is considered to be a very valuable asset. (At Legasys we said that perhaps legacy code should more properly be referred to as “heritage” code.) It is therefore not surprising that financial organizations exhibit a very strong resistance to any large scale software change, no matter what the potential payoff - it is simply too risky.

The result of this reality can be seen in the way that these organizations chose to address the Y2K problem. While it was clear from the beginning that the most desirable thing to do was to expand two digit year fields to four digits, the risk of doing so was simply too high. Impact analysis conducted at Legasys, for example, showed that in a million line mortgage application, more than 15% of the total source lines would require some kind of change. The potential risk of changing over 150,000 lines of code for this change is of course simply too high - the change was complex and chances it could be done perfectly were simply not good enough. As a result, virtually all financial institutions, including all Legasys clients, chose to use a “windowing” solution instead. While not a perfect or completely permanent solution, windowing involves changing only those lines that actually compare or do arithmetic with dates, and not any others. Impact analysis showed that this change affected fewer than 0.3% of the source lines, or only about 3,000 lines in a million line system, clearly a much lower risk.

When viewed from a risk perspective, legacy software improvement or renovation strategies can seem quite naive. For example, why would an organization take the enormous risk of attempting to “update” to a new programming language, or to “improve” the architecture of their software system when it is being smoothly evolved to adapt to new demands in small, manageable increments as it is?

A rather surprising side effect of the emphasis on risk management is the role of clones in the software system. As part of the design analysis used to tackle the Y2K problem at Legasys, we routinely ran across clones and near clones of many kinds, including large COPY books (include files), data records, program modules and code sections. Indeed, we estimated that by reducing clones the total code base of some of our clients could be reduced by a factor of two or more. However, when viewed from a risk perspective, this would very likely be a bad idea!

Why is that? Well, first we must understand the role of clones in the financial software industry. In a sense, cloning is the way in which designs are reused in these systems. Data processing programs and records across an organization often have very similar purposes; there are only a limited number of different kinds of financial tasks, and the data structures and programs to carry out these tasks are therefore very similar. So a standard practice when authoring a new application or program module is to clone an existing one - thus reusing its design, and a copy of its code.

Another source of cloning is the practice of custom views of a data record. Financial systems are typically based around a relatively small number of large central record structures. Many applications and modules will use the central descriptions of these records as COPY books. However, as with any application, financial solutions can often be more easily coded by changing the way in which the data is viewed or structured. One way to handle this adaptation is to change the central, single copy of the data description to allow for the new view. However, if we constrain ourselves to such a single copy, every time we add a view or make a change for a new program or maintenance step, we are risking the possibility that the change, however small, may affect one of the many other programs and applications that use the record. At the very least we have to test them all over again just in case - and as we shall see, testing is our major cost.

So instead, it is common practice in the industry to use custom clones of the main record data description fore each application or even module. They all still share the design of the data structure, but each is free to change its own view in any way it likes, without fear of affecting any other. But suppose, you say, that a fix is made to one of them because a bug was discovered - wouldn't we want the fix to be in all the versions of the data description? The surprising answer is no - as a matter of fact, we exactly do NOT want to do that - because the risk that one of the other, already properly working, modules or applications may depend on the old description is too high! Instead, from a risk perspective, it makes sense only to change the one in the module that failed - and not in any other. If the analogous problem has not appeared in the other module, then it most likely doesn't affect it. And if it ever does, then the lowest risk solution will be to address that problem separately at that time.

Cloned record descriptions also have the advantage that they increase the degrees of freedom in implementing and maintaining each new application or module - each is free to change or refine its view of the data in any way it sees fit, including ways to assist in addressing observed problems. This can help speed maintenance, and without fear of accidentally affecting other unrelated modules or applications. Similarly, cloned applications and program modules increase the degrees of freedom and decrease maintenance risk for program logic.

Because the culture of these software systems is so deeply based in such cloning, clone detection may not even be a very interesting service. For the most part, programmers are quite aware of the clones. Because the systems are based in a small number of large central records, typically programmers can easily identify which central structure is associated with each clone simply by looking at the code. Similar observations can be made about cloned programs and program sections.

The one time that clone detection does have a central role to play is on those rare occasions when a fundamental change is made to the central record itself. In such a situation, identification of all clones of the record is essential, and programs comprehension techniques have a very strong role to play if they can provide the analysis quickly when needed.

## 4.2. The Technological Reality

In the financial software world, COBOL is king. The Gartner Group estimates that the total number of lines of COBOL in the world is over 200 Gloc, and that this number is presently still growing at about 5 Gloc per year. There are good reasons for this - the COBOL language was designed for data processing, and no other language serves the purpose as well. Conversion to other "modern" languages actually makes little sense in the data processing environment. Modern languages such as Java handle text-oriented data processing records very poorly, because the object-oriented model simply does not match the nested records and text storage overlays that are the bread and butter of data processing tasks. Modern languages also provide no native scaled decimal arithmetic, which is widely used in data processing both to avoid non-decimal roundoff and for easy conversion to and from text fields.

Perhaps the most compelling reason to stay in COBOL is that the risk of language conversion is so enormous that it is simply not reasonable to try. Conversion cannot be done instantly for a typical financial organization's code base of over a hundred million lines of code. It would take a large number of person-years to do such a conversion, and in the interim the system would have to be continually updated in response to the business pressures of competition, government and other external forces. By the time a conversion was completed, it would be so far out of date with the current system as to be unusable. Even if it were usable, the chances that an entire new system of such a size would perform without error are essentially zero, even with very extensive testing.

Thus, as many of you already know, any program comprehension services we provide must be aimed at COBOL, RPG and the other legacy languages if they are to serve the majority of the industry. And providing language conversion services should concentrate on those conversions that make sense, such as RPG and PL/1 to COBOL (a very popular business move). One conversion that does however make sense is the conversion of interactive systems implemented using CICS and other old interaction techniques to modern interaction techniques such as Java and Visual Basic. This leads to the need for aspect identification, something program comprehension research can help with.

One of the most common phrases we heard in the business software world is "tools don't help". The resistance to new software "tools" to assist in software maintenance is very large, and based on a long and unhappy experience with many inadequate and premature CASE tools of the past. It's clear how this attitude has come about - even our most recent tools present results in forms that are inaccessible to most COBOL programmers.

Graphs and UML diagrams may be full of meaning, but unless their meaning can be presented in source terms, they will not be understood by the financial programming community. Slices and flow graphs may be a powerful representation of potential impact, but they do not reduce the source to be looked at significantly enough to enhance productivity.

In the world of financial systems, only the source is real. The reason is clear - pressures require that changes and enhancements be made very quickly, and to quote a COBOL programmer, “you can't make a fix in the abstract”. Our experience is that the programmers in these shops are very deeply familiar with the source code of their systems. They understand the source code and its business meaning very deeply. The algorithms and structures being manipulated are not complex and have a high level of similarity (and cloning, as we have noted).

If we are to assist in software maintenance in a way that makes sense in this technical environment, our inferences and impact analyses must be presented in terms of source. And if they are to demonstrate a real advantage over hand analysis by a programmer with deep knowledge of the source, then they must present results that are immediate, accurate and tightly focussed to the parts of source that are relevant to the task at hand.

This “source imperative” was addressed at Legasys using two technologies: unique naming and hot spot markup. Unique naming [35] is a method for attaching design recovered software graphs and the original source code in a way that allows any analysis carried out on the one to be immediately reflected in the other. The idea is to mark up every entity declaration and reference in the source code with a unique identifier, or "UID", which can be used universally across an entire system to refer to the unique entity referred to (Figure 2(a)). This UID is then used to refer to the entity in the design recovered graph as well (Figure 2(b)). UID's are then used to serve as a set of “keys” implicitly linking each entity in the design database to its declaration and all of its references in the original source files.

This idea has several advantages. First, since it involves keeping separate source and graph representations that are implicitly linked by UID's, there is no need for any source information in the graph. This means smaller, more efficient software graphs without AST's, syntactic elements or other clutter. Second, it allows for easily “pushing” results of a graph analysis back to source - when a graph analysis yields a new relationship between two entities (UID's), the source involved can be highlighted simply by marking references in the source with the same UID's. Similarly, if an inference is made from source analysis, it can be reflected into the design graph simply by asserting a new relationship between the UID's of the entity references involved in the uniquely named source.

At Legasys the idea was formalized in HSML [36], the Hot Spot Markup Language, a notation designed for expressing the markup of source based on design graph inferences. HSML expressions used syntactic categories to express scope limitations of source markup based on design graph queries. Unique naming provides the link between the two.

```
        01 DATE.
           05 YY     PIC 99.
           05 MM     PIC 99.
           05 DD     PIC 99.
        77 YEAR      PIC 99.
           MOVE YY TO YEAR.

 01 [DATE - A `A.CBL` # DATE-REC]. [
    05 [YY DATE - A `A.CBL` # YY] PIC 99.
    05 [MM DATE - A `A.CBL` # MM] PIC 99.
    05 [DD DATE - A `A.CBL` # DD] PIC 99.]
 77 [YEAR - A `A.CBL` # YEAR] PIC 99.
    MOVE [YY DATE - A `A.CBL` # YY] TO
         [YEAR - A `A.CBL` # YEAR].
```

(a) Unique naming in source.

*In this simple example, the identifiers DATE-REC, YY, MM, DD and YEAR in the original code (above) are almost certainly not unique over the entire application. Unique renaming (below) labels each declared identifier with a unique name (UID) encoding the entire context, program name and source file of the declared entity, for example YY DATE – A `A.CBL` for YY. Each resolved reference to the same entity is labelled with the same UID, so that the declaration and every reference to an identifier is labelled with its unique name.*

```
Field (YY DATE - A `A.CBL`, DATE - A `A.CBL`)
Field (MM DATE - A `A.CBL`, DATE - A `A.CBL`)
Field (DD DATE - A `A.CBL`, DATE - A `A.CBL`)
Move (YY DATE - A `A.CBL`, YEAR - A `A.CBL`)
```

(b) Unique naming in the design graph.

*Design recovery analyzes the uniquely renamed source to yield the design graph for the system. The edgelist of the design graph uses the unique names (UID’s) of the related entities to refer to them. Because all references to an entity in both the design graph and the source use the same unique name, any inferences about the entity in the graph may be easily attached to source references, and any new analysis about the entity in the source may be easily added as a new edge in the graph.*

**Figure** 2. Unique Naming as a Bridge Between Source and Graph.

Another technical reality is the prevalence of custom local dialects, features, preprocessors and coding tools in the business programming world. This presents a serious barrier to acceptance of our techniques that has already been the subject of a number of papers [34,37,38]. An analysis technique that fails every time it hits a new or different syntax or language variant is doomed to oblivion in these environments.

At Legasys we began addressing this problem using robust parsing techniques somewhat related to Moonen's island parsing [37]. Robust parsing yields some parse for every input, no matter how malformed, by allowing for uninterpreted sections where a parse could not be found. Such techniques are an absolute necessity if program comprehension is to make inroads in the financial community. In most analyses, partial answers are acceptable - any help with a complex impact analysis is useful. But having no answer is completely unacceptable, and programmers will rapidly drop any analyzer that fails to yield answers due to parse errors.

## 4.3. The Social Reality

The management structure and consequent social environment of software maintenance groups in the financial industry can have an enormous influence on the adoption of program comprehension technology. A large financial institution typically will have some hundreds of millions of lines of source code maintained by some hundreds of programmers. The source code is organized into some hundreds of applications, each on the order of a few hundred thousand to a million lines of code. Not surprisingly, the maintainers are organized into teams that reflect the application structure of the software systems. Each team manages one or more closely related applications under the direction of a technical manager.

In general, each technical manager has the majority responsibility for decisions concerning the applications maintained by his or her team. Group managers or technical executives are responsible for a set of teams, and general managers or vice presidents are responsible for sets of group managers. The important point here is how decisions are made concerning new technology adoption.

Let us propose to the company that they adopt our program comprehension technology. Usually we begin by demonstrating our capabilities to the general managers or vice presidents. We show how our inferences can reduce maintenance tasks, automate maintenance steps, and so on. If we've done our homework, we show our results as source, so that they know that what we have is real (recall that only source is real).

What happens when we leave the meeting? Well, the vice presidents ask the group managers about our stuff, the group managers ask their technical managers, and the technical managers consult with their programmers. The latter two are puzzled: first, they know they are doing a great job, and they are wondering why the boss is interested in this idea. They try out our technique. From their point of view it is magic - in goes the old code, out comes the new code, or the UML, or whatever. They are dubious because they don't understand the automatic process, and at the same time they feel threatened - if it works, the company may not need all of them any more!

The result is that the technical managers report to their group managers that it can't help, or that they can already do this by hand, or some other negative response. The group manager reports to the vice president that the technical staff don't see any role for the technology, and whoosh! - we're out the door.

If we are to be successful in having our technologies adopted in these environments, we are going to have to deal with this social reality. At Legasys, our approach to this was to present all our results as advice, leaving all programming decisions in the hands of the maintainer. The idea was for our techniques to assist the programmer, not replace him or her. So even where our technology was doing an automated reprogramming, we did not present the result as a finished program. Rather, we presented a concise report consisting of the set of suggested changes to the source code (remember it must be in terms of source code!) for the programmer to examine and choose to use or not. In order to make the decisions efficient, the report provided enough context around the suggested changes to allow the programmer to make decisions without referring to the entire source in most cases, and a web interface assisted in hand changes where warranted.

The important point here is that while the technology is doing the same thing, all control is left in the hands of the programmer. There is no threat because from the programmer's view he or she is still doing all the maintenance on the source, it is just that a very insightful assistant is helping by doing some of the leg work for them. They understand what is going on because they can see what is suggested to be done, check it out for themselves, and reject things they don't trust. This philosophy of assist, don't replace, is the only one that can succeed in the social and management environment of these organizations.

This way of doing things also has another advantage. Even if our reprogramming technology isn't able to completely automate, it can still be useful - if there are things our system does not know how to resolve, it can present the unchanged code sections to the human programmer with the suggestion that something needs to be done, but the technique can't automate it for him or her. This not only reinforces the sense of control and feeling of value for the programmer, but it makes a symbiosis more powerful than either the technique alone or the programmer alone. The program comprehension and automation does what it does best - deep and complex semantic searching and automated reprogramming for common cases using templates. The human does what he or she does best - apply business and system knowledge to check the automation, and resolve any cases that the automation can't handle.

At Legasys we exploited this model in addressing the Y2K problem. Our LS/2000 system would exploit human interaction in two stages. Following unique naming and design recovery, LS/2000 used an exhaustive trace of reference chains in the design graph to identify and classify into the 45 or so different date formats each data field representing a date in an application. In the process, some fields whose date status or format was ambiguous or beyond the analysis capabilities of the system were usually identified. In this first interaction stage, the programmer was presented with a source-linked web interface in which the system asked for advice to resolve each of these ambiguities. Alternative interpretations were presented, along with links to the source code contexts (using UID's) and related items. Given the programmer's business and application knowledge, these ambiguities were typically easy to resolve and a half-million line application could usually be resolved in about an hour.

Following date resolution, the LS/2000 system completed the process using patterns to identify and reprogram Y2K sensitive uses of the identified date fields in comparisons, arithmetic, and so on. Once the process was completed, the results were again presented to the programmer, this time as a hot spot report showing the original and reprogrammed source code for each change (Figure 3). Acceptance or rejection of each change was again left to the programmer, and those very few sections requiring change that the system could not automatically reprogram could be handled by hand.

Another social barrier to automation is the question of source ownership. As we have already noted, programmers in financial software environments are intimately familiar with the source code of their systems. Their ability to continue to understand it is rooted deeply in its recognizability - it has a familiar look and feel. People recognize familiar code sections in much the same way that they recognize faces, using synesthetic memory. Teams become attached to their application and its source code as a familiar old friend.

Recognizability of the source therefore becomes an important issue. Even if our automated maintenance systems do a wonderful job of renovating or updating an application, if the source code comes back reformatted, even just by changing the indentation or comment placement, the recognizability and hence the deep understanding is disturbed. It just doesn't look like their old friend any more, and they want their old code back.

The importance of this issue to acceptance of automation is far greater than one might think. At Legasys we found it to be a significant barrier to acceptance of our work. In the end we dealt with this problem using a source code factoring technology [39], in which details of source code format were factored out early in the analysis, and then restored after reprogramming, making the text-wise most minimal change to the original source code. In this way, the source code retains its old familiar face, with maybe a couple of warts removed.

Perhaps the most obvious of the social barriers, but also the easiest to overlook, is the issue of simplicity. A programmer cannot accept a solution or a technology if they can't understand it. This implies that whatever solutions are offered by our techniques, they must be straightforward enough to be understood by the programmers who will use them. We've already pointed out that source code is the only real medium of understanding for programmers of the financial world (and thus graphs may not be a good choice for presenting analysis results to this community). But additionally, any source code introduced must be simple enough to be widely understood, and must fit into the standard practice and style of the community.

It's easy to make the mistake of insulting a programming staff by shooting “over their heads”. An example of this can be drawn from the Y2K experience. Once it had been decided, for risk reasons, that a “windowing” solution to the Y2K problem was to be adopted in the reprogramming, at Legasys we proposed a one-line solution involving a mathematically perfect reprogramming using modular arithmetic and COBOL built-in functions to make a permanent and (mathematically) beautiful solution. The Legasys staff were very proud of this solution.

The only problem was that the solution was completely opaque to anyone who was not intimately familiar with the brand-new built-in function capabilities of COBOL (which are in general unknown in the financial community), and who did not hold a degree in algebra. Of course, when this solution was presented with great enthusiasm to our Y2K clients in the banking community we were met with a blank stare. Once they understood what we were trying to program, they simply said, “oh, you mean this” - and showed us a simpler, easier solution that they had been using as a quick hack for some time. While not completely permanent, their simple solution would work for the next seventy years, and was easily adapted after that.

```
Program: XYEGPROG
Line  Program Source Line                                                          HS Src File
----  -------------------                                                          -- --------
26    002600      16  FISCAL-DATE-JULIAN      PIC S9(5) COMP-3.                    <- XXCOPYDJ

52    005300            24  WS-FISCAL-DATE-JULIAN   PIC S9(5) COMP-3.              <- XYEGPROG

63              COPY LS2KROLL.                                                     <= XYEGPROG
64              77  Y2K-FISCAL-DATE-JULIAN     PIC 9(5).                           <= XYEGPROG
65              77  Y2K-WS-FISCAL-DATE-JULIAN  PIC 9(5).                           <= XYEGPROG

232                ADD ROLLDIFF-1-YYNNN FISCAL-DATE-JULIAN GIVING                  <= XYEGPROG
233                    Y2K-FISCAL-DATE-JULIAN                                      <= XYEGPROG
234                ADD ROLLDIFF-1-YYNNN WS-FISCAL-DATE-JULIAN GIVING               <= XYEGPROG
235                    Y2K-WS-FISCAL-DATE-JULIAN                                   <= XYEGPROG
236         *******IF FISCAL-DATE-JULIAN IS NOT GREATER THAN                       <= XYEGPROG
237         ****************WS-FISCAL-DATE-JULIAN                                  <= XYEGPROG
238                IF Y2K-FISCAL-DATE-JULIAN IS NOT GREATER THAN                   <= XYEGPROG
239                         Y2K-WS-FISCAL-DATE-JULIAN                              <= XYEGPROG
240                   PERFORM FISCAL-DATE-LESS.                                       XYEGPROG
```

**Figure** 3. Example LS/2000 Hot Spot Report.

Of course, upon reflection, the simpler solution is a better one for many reasons. Because it is easily understood by every COBOL programmer, it reduces cognitive overhead and enhances recognizability (it has a nice "face"). Because it is consistent with a solution already in use, it is familiar. And because it is simpler, it reduces risk and enhances long term maintainability.

The mistake of assuming that an academic solution is superior or more desirable is an easy one to make, but it is one we should avoid if we want to make serious contributions to industrial practice.

### 4.4. The Economic Reality

In the process of analysis of financial systems at Legasys it was frequently the case that we discovered significant opportunities for improvement of the systems. For example, we have already observed that by removing clones we could frequently reduce the code base by a factor of two or more. Whenever we suggested to a client that it would make sense to take action on such opportunities, the most common answer was: "good idea, but there's no budget for it". This meant that it could not be done, because it was impossible to assign any staff to doing it.

In order to understand this barrier, it is necessary to understand a little about the planning environment of the software divisions of these organizations. As in many organizations, budgets are allocated on an annual basis. Each budget item is considered on the basis of how it will contribute to the bottom line (profit) of the company in the most cost-effective way. While it is well understood how enhancement, correction and testing helps to serve customers, assist marketing and reduce risk, the argument for the economic benefits of preventive maintenance is much more difficult to make.

For example, while re-architecting a system may potentially yield long term benefits, the system is already known to be working well and to be maintainable - because it is already being maintained and exhibiting high quality. Why would we allocate the large budget and divert the person power it would take to undertake a re-architecting of the software? The cost will be high, with no effect on the bottom line for the year, and the project will divert staff from enhancements and fixes that do affect the bottom line. The upshot of this reasoning is that in these environments, there will never be a budget for it.

A possible strategy to overcome this problem was planned at Legasys but never implemented. The idea was to present re-architecting steps in such a way that they can be carried out in an incremental fashion as part of ongoing maintenance. As regular corrective maintenance and enhancement is going on, program comprehension would provide an awareness of the desired improved architecture, in the form of reports or lists of desirable source changes that can be referred to while maintaining. Each time a maintenance step is undertaken, small architectural improvements suggested by the reports are made as a side effect of the work being done, with the hope that the system would gradually evolve towards a better overall architecture.

The challenge is to provide program comprehension insights and architecture improvement suggestions in a way that is accessible, incremental and expressed in source terms. A tall order indeed!

Another economic barrier to adoption is due to a simple fact - in the high quality environment of financial systems we have spoken of, the dominant cost of software maintenance is the cost of testing. According to Legasys' largest clients, testing accounts for over 70% of the total cost and time spent in software maintenance. The result of this is that if you hope to affect the cost of maintenance, you must assist in some way with testing.

At Legasys we used our notion of hot spot reports to provide some such assistance with the testing of Y2K changes. Each hot spot report showed the changes that had been suggested or made for each potential Y2K risk in a concise report for each source module. The hot spots were then used as a testing checklist, and Y2K testing was guided by a strategy of "covering" the hot spots in testing each module. The result was that we were able to very significantly reduce the cost of the Y2K change by reducing the dominant testing costs as well as analysis and reprogramming costs.

## 5. Conclusions and Suggestions for the Future

In this paper we have explored, in one of the largest software communities, a few of the industry realities that place practical barriers in the way of the adoption of program comprehension and software maintenance automation technology. Based on experience with over 4.5 Gloc of financial code processed at Legasys for the Y2K and other maintenance problems, we observed that by adapting to these realities we can help to improve the chances of adoption of our technologies and increase levels of acceptance.

The observations of this paper indicate several concrete steps that we can take as a community to help speed the industrial adoption of program comprehension techniques:

- We can concentrate on services rather than tools
- We can think in terms of assistants rather than robots
- We can couch the results of our analyses and suggestions in the familiar terms of concrete source code
- We can emphasize agile, lightweight, hands-off techniques that provide timely answers as needed
- We can design to keep control of the analysis and maintenance in the hands of the programmers
- We can adapt to take advantage of the programmers' own knowledge and understanding to help our techniques do better
- We can spend more effort in understanding and adapting to the context of our potential users

The key point I hope you will take away from this paper is the importance of spending time understanding the target community. By studying the maintenance culture of each industrial community, by treating their way of doing things with respect, and by working to understand how our techniques can best be fit into their existing working environment, we can both increase chances of adoption and enhance our own success.

## Related Work.

Many people have studied adoption and the industrial environment more thoroughly and formally than I have in this rambling collection of experiences. Harry Sneed has for years reported on industrial realities and their implications, including risks [32]. Paul Klint, Mark van den Brand, Leon Moonen and their colleagues at CWI have published experience with industrial COBOL systems similar to those I have reported here [29]. Implications of social and technical culture on software maintenance have been studied by Janice Singer [40] and Timothy Lethbridge [41]. Ric Holt, Hausi Müller, John Mylopoulos and Kostas Kontogiannis have reported experience with many realities of the industrial software development environment at IBM [25], and recently Müller's group has focussed on adoption as a primary goal [42]. My apologies to the many others studying technology adoption and other topics who in my ignorance I have forgotten or may not be aware of.

## Acknowledgments.

I wish to thank my colleagues at Legasys, some of whom are the real source of some of the insights shared here. I particularly note Kevin Schneider, Tom Dean, Andrew Malton and Donald Jardine. The understanding of the financial systems environment I share here comes from many hours spent with programmers, technical managers, group leaders and vice presidents in our client organizations. Of particular note is the Bank of Nova Scotia, who took a chance on Legasys in the early days and taught us most of what I have reported here, and IBM Global Services, whose deep knowledge of the data processing industry helped us understand how to make our solutions more accessible to the business software community.

## References.

[1] K.B. Gallagher and J.R. Lyle, "Using Program Slicing in Software Maintenance", *IEEE Transactions on Software Engineering* 17,8 (August 1991), pp. 751-761.

[2] J. Han, "Supporting Impact Analysis and Change Propagation in Software Engineering Environments", *Proc. STEP'97 - 8th International Workshop on Software Technology and Engineering Practice,* London (July 1997), pp. 172-182.

[3] M.A. Chaumun, H. Kabaili, R.K. Keller and F. Lustman, "A Change Impact Model for Changeability Assessment in Object-Oriented Software Systems", *Proc. CSMR'99 - 3rd Euromicro Working Conference on Software Maintenance and Reengineering,* Amsterdam (March 1999), pp. 130-138.

[4] L. Moonen, "Lightweight Impact Analysis using Island Grammars", *Proc. IWPC 2002 - 10th International Workshop on Program Comprehension,* Paris (June 2002), pp. 219-228.

[5] J. Zhao, ""Change Impact Analysis for Aspect-Oriented Software Evolution", *Proc. 5th Int. Workshop on Principles of Software Evolution,* Orlando, Florida (May 2002), pp. 108-112.

[6] A. Cimitile, A.R. Fasolino , G. Visaggio, "A Software Model for Impact Analysis: A Validation Experiment", *Proc. WCRE 99 - 5th International Conference on Reverse Engineering,* Atlanta, Georgia (October 1999), pp. 212-223.

[7] S. Zhou, H. Zedan and A. Cau, "A Framework For Analyzing The Effect of 'Change' In Legacy Code", *Proc. ICSM'99 - International Conference on Software Maintenance,* Oxford, England (August 1999), pp. 23-32.

[8] V. Tzerpos and R.C. Holt, "The Orphan Adoption Problem in Architecture Maintenance", *WCRE '97 - 4th Working Conf. on Reverse Engineering,* Amsterdam (October 1997), pp. 76-83.

[9] R.C. Holt, "Structural manipulations of software architecture using Tarski relational algebra", *WCRE'98 - 5th Working Conf. on Reverse Engineering,* Honolulu (October 1998), pp. 210-219.

[10] J.B. Tran, M.W. Godfrey, E.H.S. Lee, and R.C. Holt, "Architecture Analysis and Repair of Open Source Software", *Proc. IWPC'00 - 8th International Workshop on Program Comprehension,* Limerick, Ireland (June 2000), pp. 48-59.

[11] H. Fahmy and R.C. Holt. "Software Architecture Transformations", *Proc. ICSM'00 - International Conference on Software Maintenance,* San Jose (October 2000), pp. 88-96.

[12] H.J. Johnson, "Substring Matching for Clone Detection and Change Tracking", *Proc. ICSM'94 -Int. Conference on Software Maintenance,* Victoria, Canada (September 1994), pp. 120-126.

[13] B. S. Baker, "On Finding Duplication and Near-duplication in Large Software Systems", *WCRE'95 - 2nd Working Conf. on Reverse Engineering,* Toronto (July 1995), pp. 86-95.

[14] E.L. Burd and M. Munro, "Investigating the Maintenance Implications of the Replication of Code", *Proc. ICSM'97 - Int. Conference on Software Maintenance,* Bari, Italy (October 1997), pp. 322-331.

[15] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection using Abstract Syntax Trees", *Proc. ICSM'98 - International Conference on Software Maintenance,* Bethesda, Maryland (November 1998), pp. 368-377.

[16] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code", *Proc. ICSM'99 - International Conference on Software Maintenance,* Oxford, England (August 1999), pp. 109-119.

[17] Y. Ueda, T. Kamiya, S. Kusumoto and K. InDue, "Gemini: Maintenance Support Environment Based on Code Clone Analysis", *Proc. METRICS 2002 - 8th International Symposium on Software Metrics,* Ottawa, Canada (June 2002), pp. 67-76.

[18] T. Kamiya ,S. Kusumoto , K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code", *IEEE Transactions on Software Engineering* 28,7 (July 2002), pp. 654-670.

[19] A. De Lucia, “Program slicing: Methods and Applications”, *Proc. SCAM’01 - Int. Workshop on Source Code Analysis and Manipulation,* Florence, Italy (November 2001), pp. 142-149.

[20] M. Harman, L. Hu, R.M. Hierons, C. Fox, S. Danicic, J. Wegener, H. Sthamer and André Baresel, “Evolutionary Testing Supported by Slicing and Transformation”, *Proc. ICSM 2002 - International Conference on Software Maintenance,* Montréal, Canada (October 2002), pp. 285-294.

[21] J. Zhao, “Slicing Aspect-Oriented Software”, *Proc. IWPC’02 - 10th IEEE International Workshop on Programming Comprehension,* Paris (June 2002), pp. 251-260.

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

[23] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin, “Automated support for program refactoring using invariants”, *Proc. ICSM 2002 - International Conference on Software Maintenance,* Montréal, Canada (October 2002), pp. 736-743.

[24] S. Mancoridis and R.C. Holt, “Recovering the Structure of Software Systems Using Tube Graph Interconnection Clustering”, *Proc. ICSM 1996 - Int. Conference on Software Maintenance,* Monterey, California (Nov. 1996), pp. 23-32.

[25] P. Finnigan, R.C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong, “The Software Bookshelf”, *IBM Systems Journal* 36,4 (November 1997), pp. 564-593.

[26] S.R. Tilley, H.A. Müller , M.J. Whitney, and K. Wong, “Domain--retargetable Reverse Engineering”, *Proc. ICSM 1993 - International Conference on Software Maintenance,* Montréal, Canada (September 1993), pp. 130-139.

[27] M.-A.Storey, K. Wong and H.A. Müller, “Rigi - A Visualization Environment for Reverse Engineering”, *Proc. ICSE’97 - IEEE 19th International Conference on Software Engineering,* Boston, Massachusetts (May 1997), pp. 606-607.

[28] M.-A.Storey, K. Wong, D. Hooper, K. Hopkins, and H.A. Müller, “On Designing an Experiment to Evaluate a Reverse Engineering Tool”, *WCRE’96 - 3rd Working Conf. on Reverse Engineering,* Monterey, California (Nov. 1996), pp. 31-41.

[29] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef, “Control Flow Normalization for COBOL/CICS Legacy Systems”, *Proc. CSMR’98 - 2nd Euromicro Conference on Software Maintenance and Reengineering,* Palazzo degli Affari, Italy (March 1998), pp. 11-19.

[30] A. van Deursen, P. Klint and C. Verhoef, “Research Issues in Software Renovation”, *FASE’99 - Fundamental Approaches to Software Engineering,* Amsterdam (March 1999), pp. 1-21.

[31] P. Klint and C. Verhoef, “Enabling the Creation of Knowledge about Software Assets”, *Data and Knowledge Engineering* 41,2-3 (June 2002), pp. 141-158.

[32] H.M. Sneed, “Risks Involved in Reengineering Projects”, *Proc. WCRE’99 - 6th Working Conference on Reverse Engineering,* Atlanta, Georgia (October 1999), pp. 204-211.

[33] H.M. Sneed, “Wrapping Legacy COBOL Programs behind an XML-Interface”, *WCRE’01 - 8th Working Conf. on Reverse Engineering,* Stuttgart, Germany (Oct. 2001), pp. 189-197.

[34] M.G.J. van den Brand , M.P.A. Sellink, and C. Verhoef, “Current Parsing Techniques in Software Renovation Considered Harmful”, *Proc. IWPC’98 - 6th Int. Workshop on Program Comprehension,* Ischia, Italy (June 1998), pp. 108-117.

[35] T.R. Dean, J.R. Cordy, K.A. Schneider and A.J. Malton , "Using Design Recovery Techniques to Transform Legacy Systems", *Proc. ICSM 2001 - Int. Conference on Software Maintenance,* Florence, Italy (November 2001), pp. 622-631.

[36] J.R. Cordy, K.A. Schneider, T.R. Dean and A.J. Malton, "HSML: Design Directed Source Code Hot Spots", *Proc. IWPC 2001 - 9th International Workshop on Program Comprehension*, Toronto, Canada (May 2001), pp. 145-154.

[37] Leon Moonen, “Generating Robust Parsers Using Island Grammars”, *Proc. 8th Working Conference on Reverse Engineering*, Stuttgart (October 2001), pp. 13–22.

[38] A. Cox and C. Clarke, “A Comparative Evaluation of Techniques for Syntactic Level Source Code Analysis”, *Proc. APSEC’00 - IEEE 7th Asia-Pacific Software Engineering Conference,* Singapore (December 2000), pp. 282-291.

[39] A.J. Malton, K.A. Schneider, J.R. Cordy, T.R. Dean, D. Cousineau and J. Reynolds, “Processing Software Source Text in Automated Design Recovery and Transformation”, *Proc. IWPC 2001 - 9th International Workshop on Program Comprehension,* Toronto (May 2001), pp. 127-134.

[40] J. Singer, “Practices of Software Maintenance”, *Proc. ICSM’98 - International Conference on Software Maintenance,* Bethesda, Maryland (November 1998), pp. 139-145.

[41] J. Singer, T. Lethbridge , N. Vinson, and N. Anquetil, “An Examination of Software Engineering Work Practices”, *Proc. CASCON '97 IBM Centre for Advanced Studies Conference,* Toronto (October 1997), pp. 209-223.

[42] H. Müller, “Leveraging Cognitive Support and Modern Platforms for Adoption-Centric Reverse Engineering (ACRE)”, *http://www.acse.cs.uvic.ca/pages/acse_v1_0/acre.html* (2002).