Chapter 1
Nature of Impact Analysis

Purpose

The papers in this chapter will help you understand software change and the importance of predicting the effects of changes to software before they are actually made. The papers also address where to apply impact analysis in the software-maintenance process. The chapter explains how you might determine if impact analysis is effective and presents a framework for comparing various types of impact analysis. The last paper describes a real-world problem that impact analysis could help solve.

The Papers

Software change is a fundamental ingredient of software maintenance. Impact analysis is key in analyzing change or potential change and in identifying the software objects the change might affect.

Even with the best-laid plans, software changes. Developers must understand *how* change occurs to effectively predict the impacts of that change.

One way to understand how change occurs is to understand how programs evolve. Over the last two decades, Manny Lehman and Laszlo Belady have proposed and refined "laws" of program evolution ([Belady 1976], [Lehman 1980] [Lehman 1985], [Lehman 1991], [Lehman 1994]). Although the statistical validity of these laws has been challenged [Yuen 1981], [Lawrence 1982], we believe they are helpful, if not absolute, comments on the nature of change.

Belady and Lehman studied several years of OS/360 development and change data. From this data, they proposed five laws that characterize the dynamics of program evolution:

1. *Change is continual.* "A program undergoes continuing change or becomes less useful. The change process continues until it becomes cost-effective to replace the program with a re-created version." That is, software systems change to meet the demands of users and environments until it is no longer cost-effective to change them.
2. *Complexity increases.* "As an evolving program is changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce the complexity." In other words, as the program is changed, the complexity of its structure is likely to increase unless proactively controlled.
3. *There is dynamic dependency.* This is what Belady and Lehman call the "fundamental" law of program evolution: "Program evolution is subject to a dynamic that makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and invariances." In other words, program evolution is composed of several attributes that are dynamically related such that the change in one alters others (through feedback) and tends toward a goal.
4. *Organizational stability is preserved.* "The global activity rate in a project supporting an evolving program is statistically invariant." That is, the rate of work involved in changing an evolving program tends to remain constant despite resource changes.
5. *Familiarity is conserved.* "The release content (changes, additions, deletions) of the successive releases of an evolving program is statistically invariant." In other words, the content of successive software releases tends to remain the same in terms of changes.

The three papers in this chapter add to the understanding of these five laws. In the first, "A Process Model for Software Maintenance," Robert Moreton argues convincingly for an effective software-maintenance process in light of the increasing size and complexity of software systems. A key part of software maintenance is impact analysis, in which the effects of a change request are elaborated for estimates. Moreton clearly explains where and when impact analysis should be employed in the software-maintenance process.

Although impact-analysis technology is rapidly maturing, quantitatively assessing software-change impacts is (as of early 1995) more an art than a science. People who have considerable domain, system, and program experience or knowledge can give reasonable assessments of software-change impacts in relatively small scale situations. However, impact-analysis practices vary widely, even among experienced software professionals.

We need ways to measure the potential impacts of the change itself and the system to be changed. Software stability, traceability, complexity, and size are all measures that influence the impact assessment.

In the next paper, "Impact Analysis – Towards A Framework for Comparison," we introduce a definition of impact analysis that clearly delineates the basis for comparing impact-analysis capabilities. We present a framework model of impact analysis and put forth the ideas underlying impact analysis according to the definition. We also classify several impact-analysis approaches according to this definition. The framework is based on applications of impact analysis; parts are used to perform impact analysis and parts are used to measure the effectiveness of the impact-analysis techniques. We also model the effectiveness of impact analysis to illustrate the situations (both good and bad) that are likely to be found in impact sets.

Robert Arnold concludes the chapter with a paper that describes a real-world application for impact analysis. In "Millennium Now: Solutions for Century Data Change Impact," he describes the nature of the impact-analysis problem through the date-change problem that is likely to plague software organizations at the turn of the century. He suggests steps to take in planning for the transition as well as the tools and techniques necessary to accomplish the effort. Taking a software-assets approach, Arnold suggests that early assessment is the key to addressing the impacts to an organization's software portfolio.

References

[Belady 1976] L. A. Belady and M. M. Lehman, "A Model of Large Program Development," *IBM Systems J.*, Vol. 15, No. 3, 1976, pp. 225-252.

[Lawrence 1982] M. Lawrence, "An Examination of Evolution Dynamics," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1982, pp. 188-196.

[Lehman 1980] M. M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," *Proc. IEEE*, special issue on software eng., IEEE Press, New York, Sept. 1980, pp. 1060-1076.

[Lehman 1985] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*, Academic Press, London, 1985.

[Lehman 1991] M. M. Lehman, "Software Engineering, the Software Process and Their Support," *IEE Software Eng. J.*, special issue on software environments and factories, Sept. 1991, pp. 243-258.

[Lehman 1994] M. M. Lehman, "Software Evolution," *Enc. Software Eng.*, 1994, pp. 1202-1208.

[Yuen 1981] C. K. S. Yuen, "A Phenomenology of Program Maintenance and Evolution," PhD dissertation, Dept. of Computing, Imperial College of Science and Technology, Univ. of London, London, 1981.

A process model for software maintenance

ROBERT MORETON

Director of Studies, Birmingham Polytechnic, UK

Abstract: This paper draws on work undertaken for the Butler Cox Productivity Enhancement Programme (PEP) to describe a process model which will provide a basis for overcoming the problems of cost and complexity associated with software maintenance. PEP is a continuous program that is open to organizations wishing to measure and improve systems development and productivity.

The paper argues that for maintenance work to be effective, it is vital to control the input to the process – the procedure by which change requests are notified and managed in the first place. The procedure of change management is followed by impact analysis, system release planning, change design, implementation, testing and system release/integration. These steps, which occur sequentially, are supported by a further activity that continues concurrently – progress monitoring.

The conclusion of the paper is that a coordinated program, effective across the whole maintenance process and designed to control changes to the system, will become more and more critical as the complexity of the system increases. Formal procedures are essential to ensure that software is not degraded and to provide an audit facility. At the same time there are several automated change and control packages now available that could help to reduce administrative overheads and increase control over system changes.

Introduction

Software maintenance is an expensive and complicated business. Research undertaken by Moreton (1988) for the Butler Cox Productivity Enhancement Programme (PEP), indicates that maintenance is generally undermanaged. As a result system changes are not always properly controlled and there is insufficient recognition of the benefits to be derived from the newer methods and tools. Recent articles by Glass (1988, 1989) highlight the need to formalize the maintenance process, to manage the software as a product and to link quality with maintenance processes. The purpose of this paper is to describe a process model for software maintenance which will provide a basis for rectifying the perceived deficiencies and exploiting the potential benefits.

For maintenance work to be effective, it is vital to control the input to the process – the procedure by which change requests are notified and managed in the first place. This procedure of change management is the first of several steps in the maintenance process. Change management is followed by impact analysis, system release planning, change design, implementation, testing and system release/integration. These steps, which occur sequentially, are supported by a further activity that continues concurrently – progress monitoring. The whole process is illustrated in Figure 1 (adapted from Arthur, 1988).

In a survey of 24 commercial organizations belonging to the Butler Cox PEP, respondents claimed to have a clearly defined procedure in place that corresponds to the first step, change management. Certainly, every respondent in the survey records all user requests and operational problems, but respondents admitted to some failings as well. Periodic formal audits, for instance, are in place in fewer than half of the survey respondents' businesses (see Figure 2). In order to achieve improvements in the maintainance environment, the steps in the process need to be carefully coordinated, not simply monitored individually.

Formalize the maintenance process

To appreciate the importance of formalizing the steps in the maintenance process, it helps to understand more precisely what they are.

Change management

Change management is the critical first step in the maintenance process. A formal procedure for change management is essential for two reasons: it provides a common communication channel between maintenance staff, users, project managers and operations staff, and it provides a directory of changes to the system, for status reporting, project management, auditing and quality control. The basic tool of the change-management procedure is a formal change-

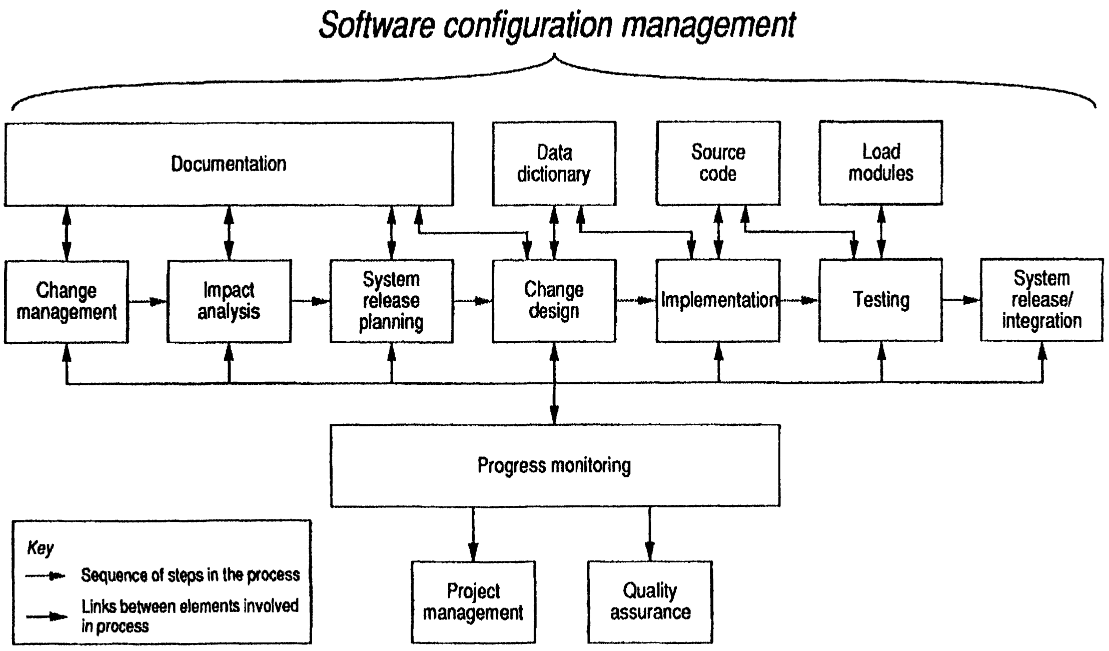


Figure 1. Define the steps in the maintenance process

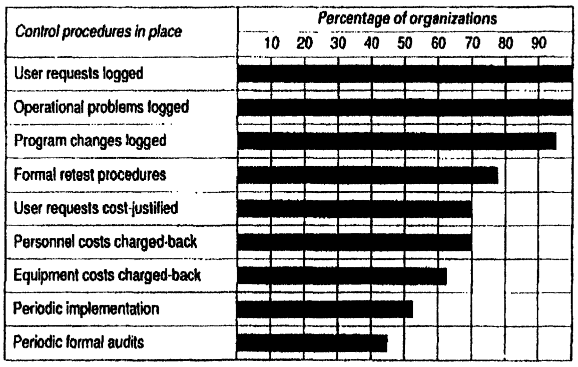


Figure 2. Most organisations have control procedures in place

request document that forms the basis of a contract between the user and the maintainer.

An important element of change management is version control (or software configuration control). It means tracking different versions of programs, releases of software and generations of hardware, and it plays a major role in ensuring the quality of delivered systems. Version control also ensures that software is not degraded by uncontrolled or unapproved changes, and provides an essential audit facility.

Impact analysis

The purpose of impact analysis is to determine the scope of change requests as a basis for accurate resource planning and scheduling, and to confirm the cost/benefit justification. Impact analysis can be broken down into four stages. The first stage is determining the scope of the change request, by verifying the information contained within it, converting it into a systems requirement, and tracing the impact (via documented records) of the change on related systems and programs. In the second stage, resourcing estimates are developed, based on considerations such as size (in estimated lines of code) and software complexity. Code analysers that measure the quality of existing code can be helpful at this stage. The third stage is analysing the costs and benefits of the change request, in the same way as for a new application. In the fourth stage, the maintenance project manager advises the users of the implications of the change request, in business rather than in technical terms, for them to decide whether to authorize proceeding with the changes.

There are three benefits of impact analysis: improved accuracy of resourcing estimates and, hence, better scheduling; a reduction in the amount of corrective maintenance, because of fewer introduced errors; improved software quality.

System release planning

In this step, the system release schedule is planned. Although well established amongst software suppliers, system release planning is not widely practised by the data processing departments of the organizations surveyed, reflecting a difference in the extent to which formal maintenance contracting is established.

A system release batches together a succession of change requests into a smaller number of discrete revisions. System releases can take place according to a timetable that is planned in advance. The timetable planning gives users the chance to set priorities for their change requests, and makes testing activities easier to schedule. The problem with system releases comes, of course, when corrective maintenance is required urgently.

Software is available to help monitor system releases. The software records the changes incorporated in, and the date of, each release, and provides information for project control, auditing and management.

Change design and implementation

The common thread in the work in these two steps is that they are undertaken to satisfy an often short-term user requirement.

Corrective maintenance, in particular, will be undertaken in a limited time and will be concerned primarily with fault repair (with little regard for careful design and integration changes). Emergency repairs must subsequently be linked to the formal software-maintenance process and be treated as a new change request. This will ensure that the repairs are correctly implemented and that the design documentation is updated.

Adaptive maintenance will functionally enhance an existing system. The design and implementation process is similar but more restricted than the design and implementation of new application systems. The major difference is that the design implications of enhancements must be taken into account in the subsequent program and module implementation. Failure to design the change at each level can result in an increasingly complex, unreliable and unmaintainable system. This leads to higher maintenance costs and reduces the life of the system.

Perfective maintenance is concerned with improving the quality of existing systems. The effort is applied to software that is the most expensive to operate and maintain. The design tasks undertaken will range from complete redesign and rewrite to partial restructuring. The process combines the characteristics of the other two types of maintenance.

Testing

The purpose of maintenance testing is to ensure that the software complies with both the change request and the original requirement specification. It forms a major part of a successful quality-assurance plan. In principle, maintenance testing is much like development testing except that, because the scope of maintenance testing is potentially narrower, fewer test cases have to be prepared, validated and filed in the test-case library.

The maintenance test cases should be created as a direct result of the first stage in the impact analysis. They should be sequenced according to the principle of incremental testing so that defects in the change-request specification and design can be identified early on. Walk-throughs and inspections should be implemented routinely as a formal element in the process.

The test-case library itself builds up over time. At first, it contains only the test cases prepared for and validated during original development. It grows as test cases for successive maintenance tests are added to it. A file of this sort is called a regression testing file. A few tools are available from suppliers (such as IBM and Digital) to help with regression testing. Although limited in what they can do (in terms of features such as automatic revalidation, for instance), they are able to provide administrative support.

System release/integration

This step consists of releasing the revised programs into live operation. The implications for maintenance staff are significant because it is their responsibility to ensure that any revised versions are completely integrated with other parts of the system, which may never have been revised or which may have been revised at different times.

Progress monitoring

Progress monitoring takes place concurrently with the other seven steps in the maintenance process. The sort of data that should be collected during progress monitoring includes the time taken per step, the effort involved and the scope of the change. Collecting and filing data of this sort so that performance can be monitored, both over time and between systems, is consistent with the disciplined formal approach to software development and maintenance implied by the terms software engineering. Improving software maintenance productivity is difficult if there is no record of where problems and successes have occurred in the past.

Coordinate the steps in the maintenance process

There is no panacea for solving the problems of maintenance. It is essential, however, to consider not only how each individual step in the process works, but also how the various steps fit together.

The software house, Peterborough Software (UK) Ltd, provides an example of how companies can successfully coordinate the steps in the software maintenance process. The problems that it faces are unusually demanding. The company maintains a range of payroll software packages. The packages run on a variety of computers, under the control of different operating systems, both within the UK and overseas. Altogether, Peterborough Software has around 400 customers. The software coding differs from country to country, to take account of local statutory regulations, such as taxation. Thus, several releases of the same package are current at a time, and all have to be supported in the field. The regulations change frequently and without warning, and maintenance changes therefore have to be implemented swiftly and accurately. The difficulties faced by Peterborough Software are further compounded when customers create nonstandard versions of the software by failing to apply maintenance modifications that are issued to them, or applying them in the wrong sequence.

How does Peterborough Software arrange its maintenance procedures against this background of complexity? The answer lies in disciplined adherence to procedural steps similar to the ones we have described here, and in the use of a computer-based program monitoring system known as the Problem Monitoring System (PMS).

The maintenance procedure is carried out by two divisions within Peterborough Software. One is the Customer Support Division, which effectively looks after management impact analysis and system release planning. The other is the Development Division, which is responsible for coding, testing and quality assurance.

Change requests received by the Customer Support Division come from three sources: customers, whose requests take the form of enhancements (called facility requests), queries and error reports; impending legislative changes; the market. To survive, Peterborough Software has to compete by offering products that are constantly being improved. Maintenance arising from customers is both adaptive and corrective in nature; from the other two sources, it is mostly adaptive and perfective.

Customers are the most important source of change requests – the Customer Support Division receives up to 400 telephone enquiries a day, for instance enquiries are routed to application-support groups organized by software product and by the kind of equipment it runs on. Within the application-support groups, consultants familiar with the way the software can be used, and with the way it works, form the first line of response. They are able to resolve most of the enquiries on the spot, but 20 per cent have to be passed to the Development Division for resolution. It is here that the PMS comes into its own. It logs problem reports at every stage of response and resolution, using customer references and event codes. When a coding change is made, for instance, the programmer records the details on the PMS. These are immediately available to others, so duplication is avoided. The PMS helps to coordinate adaptive and corrective maintenance work. It monitors maintenance progress and produces management statistics.

The Development Division is organized into groups that specialize in analysis, coding and quality assurance. Tested software is batched for release. Different forms of release reflect the level of support that Peterborough Software provides. For instance, versions for release which are necessitated by government legislation get full support. Any earlier versions still left in the field beyond a certain date no longer enjoy full support.

The Peterborough Software example is a model for the maintenance of any large application system, but particularly for multisite, multiversion implementation with large numbers of users. The principal lessons are as follows:

- Recognition of the cost and of the importance of the post-release phases of the system life cycle, and the consequent planning (for example, replacement, migration and technical design) for the maintenance effort
- The rigour applied to pre-release testing and post-release version identification and control
- The formal contractual basis that clearly specifies the responsibilities of supplier and customer
- Recognition of the relative importance of problems that occur in practice at the operational level (including those deriving from imperfect documentation or training), and at the code maintenance level and of the need to provide adequate support staff at both levels.

A coordinated programme, effective across the whole maintenance process, and designed to control changes to the system, will become more and more critical as the complexity of the systems increases. Formal procedures are essential to ensure that software is not degraded and to provide an audit facility. The experience of Peterborough Software may serve as a model. At the same time, there are several automated change-and-configuration-control packages currently being introduced to the market that could help to reduce administrative overheads and increase control over system changes.

References

Arthur, L.J. (1988) *Software Evolution,* Wiley, New York

Glass, R.L. (1988) Champions of the living software *Systems Development,* August.

Glass, R.L. (1989) Linking quality and maintenance *Systems Development,* July.

Moreton, R. (1988) Managing software maintenance *Butler Cox Productivity Enhancement Programme,* Paper 8.

Biographical notes

Robert Moreton is a principal lecturer at Birmingham Polytechnic. As Director of Studies within the Department of Computing, he is responsible for the academic quality of a range of professional and degree courses in computing and information technology. He is also an associate consultant with Butler-Cox, where he specializes in systems development methods and project management. In the past eight years, he has worked on both consultancy and research projects for the company. He has contributed to Butler-Cox reports on cost-effective systems development and maintenance, managing software maintenance and trends in information technology. He was also responsible for a study of the market for system-building tools in Europe. A common theme of his work has been tracking and evaluating developments in information technology and forecasting the potential impact of these developments.

He has a masters degree in computer science from Brunel University and is a member of the British Computer Society.

Address for correspondence: Department of Computing, Feeney Building, Birmingham Polytechnic, Perry Barr, Birmingham B42 2SU.

Impact Analysis - Towards A Framework for Comparison

Robert S. Arnold

Software Evolution Technology
12613 Rock Ridge Rd.
Herndon, VA 22070
703-450-6791
r.arnold@compmail.com

Shawn A. Bohner

MITRE Corporation
7525 Colshire Dr.
McLean, VA 22102
703-883-7354
bohner@mitre.org

ABSTRACT

The term "impact analysis" is used with many meanings. We define a three-part framework for characterizing and comparing diverse impact analysis approaches. The parts correspond to how an approach is used to accomplish impact analysis, how an approach does impact analysis internally, and the effectiveness of the impact analysis approach. To illustrate the framework's application, we classify five impact analysis approaches according to it.

1. Introduction

1.1. Purpose

Many activities are termed "impact analysis," yet it is difficult to relate them. Impact analysis (IA) approaches should be characterized so that IA approaches can be understood, compared, and assessed. This paper presents a framework for doing this.

The framework aids comparing IA approaches, assessing the strengths and weaknesses of individual IA approaches, and unifying the widely varying IA technology within a single conceptual framework. We present the framework, justify it, and use it to compare five IA approaches.[1]

1.2. How the Reader Can Use These Results

If the reader is interested in understanding, evaluating, or using IA technology, reading this paper will be helpful. By understanding the parts of the IA framework, the reader will see several features of IA that could be in a given IA approach, but may not be. This will help the reader assess the potential value of an IA approach. The reader can use the parts of the framework to critique claims of IA made by software tool vendors or by researchers.

Vendors and researchers may use the framework to redefine their work in terms comparable to other approaches and tools. This will help them track IA technology improvements.

1.3. New Results

This paper has several new results. First, the framework for understanding and classifying IA approaches is new. We are unaware of any other such paper in the IA literature. The comparison of the five IA approaches systems using the framework is new. We have not seen different types of IA approaches compared according to a common framework.

1.4. Paper Structure

Section 2 discusses why an IA classification framework is needed and the issues such a framework should address. Section 3 presents the framework. Section 4 applies the framework to compare five IA approaches and tools. Section 5 relates our work to others'.

2. The Need for an Evaluation Framework

2.1. Definition

Impact analysis (IA) is the activity of identifying what to modify to accomplish a change, or of identifying the potential consequences of a change. Examples of IA are:

- using cross reference listings to see what other parts of a program contain references to a given variable or procedure,
- using program slicing to determine the program subset that can affect the value of a given variable [Gallagher1991],
- browsing a program by opening and closing related files,
- using traceability relationships to identify changing artifacts,
- using configuration management systems to track and find changes, and
- consulting designs and specifications to determine the scope of a change.

IA precedes, or is used in conjunction with, change. It

[1] By "approach" we mean tools, semi-automatic procedures, and manual procedures.

provides input to performing the change. Normally, nothing changes except our understanding of what may be involved with the change.[2]

2.2. No Consensus Definition

IA has been practiced in various forms for years, yet there is no consensus definition. For example, IA does not appear in the IEEE Glossary of Software Engineering Terminology [IEEE1983]. [RADC1986] defined IA as "an examination of an impact to determine its parts or elements." (They defined an impact as the "effect or result of making a change to a system or its software.") [Pfleeger91] defined IA as "the evaluation of the many risks associated with the change, including estimates of the effects on resources, effort, and the schedule." (p. 433)

2.3. Related Terms

There are other IA-related terms. An **impact** (noun) is a part determined to be affected, and therefore worthy of inspection. **Traceability** is the ability to determine what parts are related to what other parts according to specific relationships. A **side effect** is an "error or other undesirable behavior that occurs as a result of a modification " [Freedman1981]. **Stability** is "...the resistance to the potential ripple effect which a program would have when it is modified" ([Yau1980], p. 28). **Ripple effect** is the "effect caused by making a small change to a system which affects many other parts of a system." [Stevens1974]

2.4. Problems with Impact Analysis Divergence

The lack of a common view of IA, and the proliferation of related terms, has led to several problems:

- It is hard to decide what is meant by IA. People rarely give explicit definitions.
- There is a lack of dimensions for comparing one IA approach with another.
- It is hard to know if enough information is available for significant comparison.
- It is hard to discern when different work on IA is related.
- It is hard to discern what work contributes to IA and what does not, according to a basic framework for assessing the technology.

This paper presents a conceptual framework for resolving these problems.

[2]Some IA approaches, for specialized applications, have the option of actually performing a change once impacts are found. We consider this an added feature and not part of the basic impact analysis definition.

3. The Impact Analysis Framework

In this section we present the framework intuitively. First we summarize the major parts of the framework. Next we discuss each part in more detail. Then we summarize the collected features of the framework as a way to compare IA approaches.

3.1. Overview

Figure 3-1 outlines how to use the framework. The framework can be used to guide understanding of an IA approach, to compare or evaluate IA approaches, or to structure analyses of IA approaches. The framework provides several points for assessing an IA approach. This

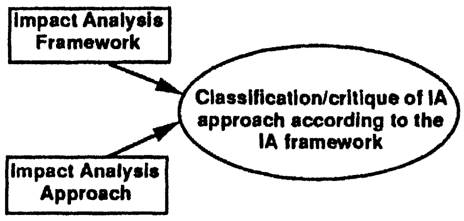

Impact Analysis Framework
Classification/critique of IA approach according to the IA framework
Impact Analysis Approach

Figure 3-1. How to Use the Impact Analysis Framework

will result in a critique or assessment of the IA approach according to the framework.

Figure 3-2 summarizes the three parts of the IA framework: IA Application, IA Parts, and IA Effectiveness. IA Application examines how the IA approach is used to accomplish IA. It looks at the features offered by the IA approach interface. IA Parts examines the nature of the internal parts and methods used to actually perform the IA. IA Effectiveness examines properties of the resulting search for impacts, especially how well they accomplish the goals of IA.[3]

The following sections, describing each part of the framework, are structured as follows: First, the purpose of the framework part is given. Then a diagram is given to frame its context. The parts of the diagram are discussed. Finally, a table is given that summarizes the framework elements resulting from this part.

[3]It is common, in evaluations of tools and technology, to create evaluation criteria for technology-specific and technology-generic factors. The latter include reliability of the vendor, user-friendliness of the tool interface, level of customer service, etc. In discussing this framework, we just focus on the impact analysis-specific items. Non-functional criteria are not discussed in this paper, but often do form a part of a technology evaluation.

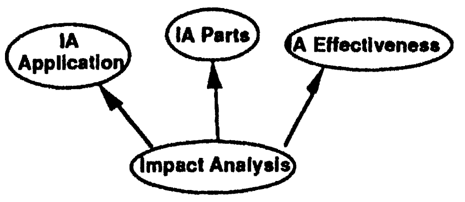


Figure 3-2. Parts of the Impact Analysis Framework

3.2. IA Application

IA Application examines how the approach is actually used to perform IA. To accomplish IA, we must have a proposed change, something to be changed, and a way to estimate what must be done to do the change.

Figure 3-3 pictures a generic IA process. A change is conceived in the real world, then reduced to a change specification.[4] The change specification uses objects and relationships familiar to the change specifier. These objects and relationships are drawn from the artifact object model. The change specification and knowledge of the item to be changed are used to specify what is initially impacted to the IA approach.

The IA approach then determines what else may be affected. These results are then translated, if necessary, back into real world terms. The results are then used to plan, to scope, or to accomplish the change.

The IA approach may also provide other features, such as

- explanations of why items are estimated to be impacted,
- measures of the IA itself,
- animation illustrating how the impacts ripple,
- access to change histories,
- suggested change strategies,
- actually performing the change,
- ways to test the change, and
- graphical views of impacts.

An example of IA is when a programmer is given an engineering change report and asked "what is involved" to do the change. The programmer should provide a difficulty assessment (how hard it will be for the programmer to do the change), a level of effort estimate (how long the change should take) and perhaps a risk assessment (how complicated the change will be to perform). IA—here browsing program code and forming a strategy for a accomplishing the change—helps the programmer to answer all three points.

The key elements of IA application are given in Table 3-1.

3.3. IA Parts

This part of the framework concerns the functional parts of the IA approach—what the approach does, and how it does it, and the duties of the agents or tools involved.

Figure 3-4 illustrates the elements of IA Parts. To express a specific change, the IA approach has its own model of objects and relationships at its interface. The input, expressed in terms of the interface object model, is translated into the IA approach's internal object model.

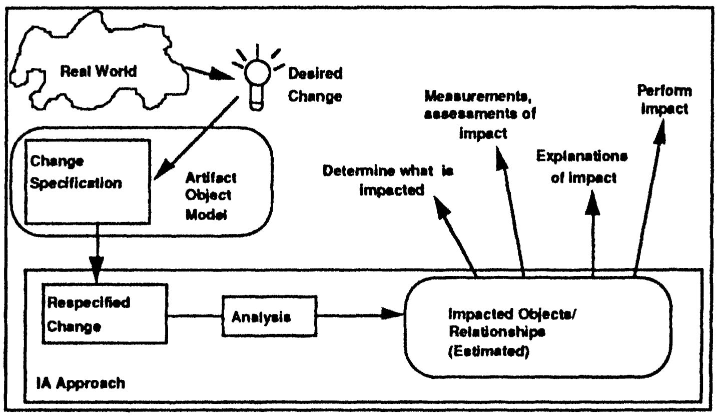


Figure 3-3. IA Application: Performing from the User's Viewpoint

The internal object model defines the objects and relationships (or dependencies) the approach uses to accomplish IA.

The internal object model is normally stored in a repository of some kind. The repository has its own features for loading, browsing, and modifying objects and relationships. The repository is loaded by decomposing the artifact into objects and relationships conforming to the internal object model.

The impact model defines the rules or embedded assumptions reflecting the semantics about what affects what. It defines the classes of objects and relationships used by the IA approach, and ways (rules, algorithms) for determining when a change to one object will affect another object. These may be embedded in the internal

[4] Often many intermediate steps are done before a change specification is reached. We focus here just on the key conceptual elements of the technical impact analysis process.

object model or the impact calculation algorithms. Sometimes they may appear as a separate rules base.

The tracing/impact approach implements the impact model. The tracing/impact approach defines how objects and dependencies are represented, how impact rules are captured (e.g., programmed), and the specific search algorithms used to find impacted objects and relationships.

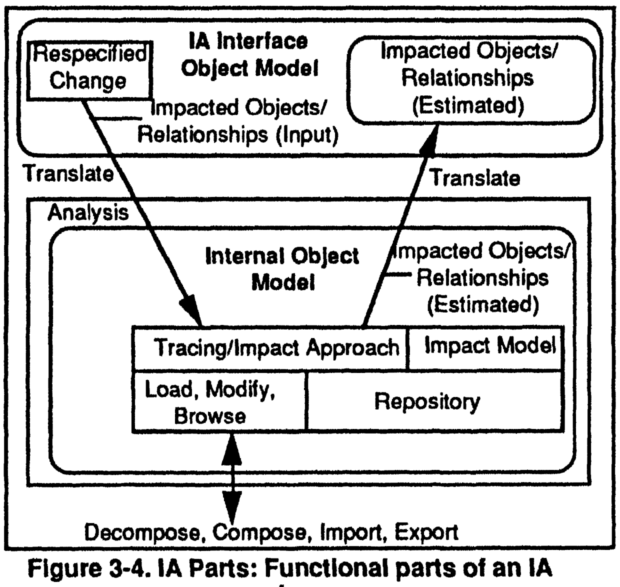
Once the results of IA are obtained at the internal object model level, these must be translated back into the interface object model, then further interpreted to determine what parts of the original artifacts are impacted. For some artifacts (e.g., programs), often the directly impacted artifact objects are supplied by the impact analysis approach. For other artifact sets (e.g., requirements), significant manual work is needed to accomplish determine what is impacted.

Each of these parts has many variations that, for brevity, we do not discuss here. Table 3-2 summarizes the elements of "IA Parts."

Table 3-1. Framework Elements for IA Application

Element	Explanation	Rating Scale
Artifact Object Model (Domain)	What are the types of objects and relationships captured from the application domain?	Program objects and/or relationships; Predefined domain objects and/or relationships; User specifiable domain objects and/or relationships; None; Unknown
Decomposition	Can the item to be analyzed be automatically decomposed and stored within the IA approach/tool?	Yes, syntax with complete semantics; Yes, syntax with some semantics; Yes, syntax only; No; Unknown
Change specification	How is the change specified for the IA approach?	Yes, with detailed analysis; Yes, with some analysis; No, not applied; Unknown
Results specification	How are the results of IA expressed?	Report; Browsing; Database view; None; Unknown
Interpretation	How much effort by the user is needed to interpret the results (i.e., derive true impacts from IA)?	Significant; Some; None; Unknown
Other features	What other features are available to the user?	<The specific feature>. E.g.: explanations, metrics, impact animation, options to perform the change, access to change histories, suggested change strategies, ways to test the change; None; Unknown

An example illustrating IA Parts is incremental program recompilation. The programmer makes a change to software and the compiler must determine the minimal parts that must be recompiled, and in what order. The change here is specified implicitly: the compiler detects which parts of the code have been modified. The change is then translated into the compiler's compilation graph, which captures compilation dependencies between compilable code units. The compilation graph was produced through earlier compilations of the code. The compiler than applies its impact algorithm to determine what program units must be recompiled and in what order. (Often this is not visible to the programmer, or is just taken for granted.) Though not part of IA, the compiler then often goes ahead and recompiles the affected program units. What the programmer sees is a program that has been recompiled.



Respecified Change
IA Interface Object Model
Impacted Objects/ Relationships (Estimated)
Impacted Objects/ Relationships (Input)
Translate
Translate
Analysis
Internal Object Model
Impacted Objects/ Relationships (Estimated)
Tracing/Impact Approach
Impact Model
Load, Modify, Browse
Repository
Decompose, Compose, Import, Export

Figure 3-4. IA Parts: Functional parts of an IA approach.

3.4. IA Effectiveness

This part of the framework concerns how well the IA approach accomplishes IA. Once IA is done, how accurate is it?

3.4.1. Definitions

To discuss effectiveness, we introduce the concepts pictured in Figure 3-5.

At the artifact object model level, IA is assumed to take place on a bounding set of objects, which we shall call the *System*. These objects are drawn from an encompassing model called the *Universe*.

At the interface object model level, the analogs of the System and the Universe are the *System#* and *Universe#*. (We append a "#" to sets that are at the interface object model level.) The *starting impact set* (SIS#) is the set of objects that are thought to be initially affected by a change. The *estimated impact set* (EIS#) is the set of objects estimated to be affected by the IA approach. The SIS# and EIS# are assumed to share the same object model, namely the interface object model. The *impact paths* are the search paths tying objects in the SIS# to objects in the EIS#.

The *actual impact set* (AIS) is the set of objects (in the artifact object model) actually modified as the result of performing the change. The AIS# is the image of the AIS in terms of objects and relationships in the interface object model.

The AIS is normally not unique, since a change can be implemented in several ways. (Nevertheless, in our discussion we will mention "the" AIS, meaning an AIS resulting from a particular impact analysis.) In our examples, we will also assume that the AIS reflects a correct implementation of a change.

It is also possible to characterize the SIS, EIS, and AIS in terms of the internal object model. For simplicity in discussing the framework, we shall discuss them at the level of the interface object model and above.

3.4.2. Effectiveness Concepts and Measures

We consider four areas that should be examined to determine the effectiveness of an IA approach.

3.4.2.1. SIS# and EIS#

This area looks at the relationship of the SIS# with EIS#. By definition, the EIS# always contains the SIS#. Yet the relative size of the EIS# influences the work to be done later in checking the objects that the IA approach estimates to be affected.

Table 3-3 discusses the possibilities. For each case, a picture of the relationship, with defining conditions, is given. Then a metric with a measurement is given to detect when the case occurs. A point description (i.e., in looking at a single measurement) of the implications of the case is described. Finally, the "Desired Trends" indicates the desired, expected, and goal measurement tends that would be wanted over several applications of the IA

Table 3-2. Framework Elements for IA Parts

Element	Explanation	Rating Scale
Interface Object Model	What objects and relationships can be expressed at the approach's interface?	Strings; Program objects; Predefined document objects; User specifiable document objects; Unknown
Internal Object Model	What objects and relationships does the approach use to accomplish IA?	Document oriented; Object based; Graph structure; None; Unknown
Impact Model	How are dependencies modeled? Where does the approach take these dependencies into account? How closely does the impact model mirror dependencies of the artifact's model of dependencies?	Data flow; Control flow; String matching; Object dependency; None; Unknown
Tracing/Impact Approach	How does the approach accomplish IA through tracing affected objects and relationships? What algorithms or procedures are used?	Decomposition; Pattern matching; Heuristic search; Stochastic search; Not explicit; None; Unknown
Decomposition	What objects and relationships are captured from the artifact and stored as objects and relationships within the repository?	Compiler; Database entity; Object filter; None; Unknown
Repository	What repository is used to store objects and relationships?	Relational Database Management System (RDBMS); File system; None; Unknown
Load, modify, browse	What features does the repository have for loading objects and relationships into it, modifying them, and browsing them?	Load; Modify; Browse; All three available; None; Unknown

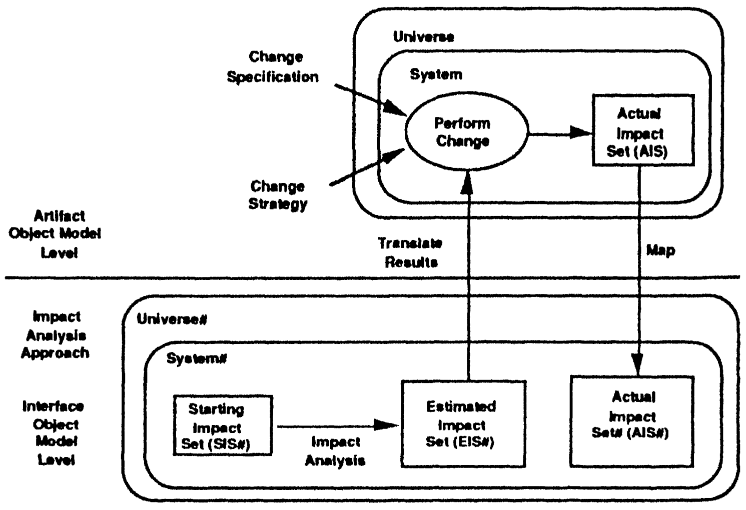


Figure 3-5. Key sets of objects in IA effectiveness.

approach. (The percentages in this column are suggested starting points. They may be tuned as desired.) The percentages in the "Expected" and "Goal" trends sum to 100% across the boxes, rather than within the boxes, because of mutually exclusive cases.

3.4.2.2. EIS# and System#

This area looks at the relationship of the EIS# with the System#. In general, we do not want the IA approach to estimate that everything is affected—that is, the EIS# is the same as the System#—unless that is indeed the case. The "distance" of the EIS# from the System# is a way to gauge the sharpness of the IA. Table 3-4 illustrates some cases and interpretations.

3.4.2.3. EIS# and AIS#

The relationship between the EIS# and the AIS# is also meaningful. We want the AIS# to be contained regularly in the EIS#, and preferably very close or exactly the same. This would give us more confidence that IA approach results in estimated impacts that can be more carefully relied on to give the true scope of a change.

We do not want the AIS# greater than EIS#, since this means that the EIS# is less a reliable indicator of the true scope of a change. Table 3-5 illustrates some cases and

Table 3-3. SIS# and EIS# Possibilities

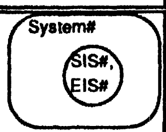
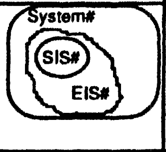
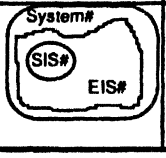
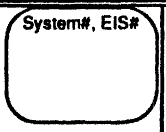
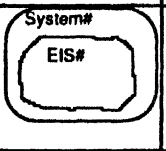
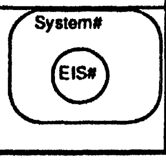
Case/Picture	Defining Conditions	Measurement When Present	Point Interpretation	Desired Trend
1 	EIS# = SIS#; EIS#, SIS# $\subseteq$ System#	|SIS#| / |EIS#| = 1	Best. Estimated impact restricted to SIS#.	Desired: SIS# = EIS# always. Expected: SIS# = EIS# are equal in 5% of a random sample of impact analyses Goal: SIS# = EIS# in 20% of a random sample
2 	|EIS#| > |SIS#|; SIS# $\subset$ EIS#; EIS#, SIS# $\subseteq$ System#	K < |SIS#| / |EIS#| < 1, for some user-selected K such that 0 < K < 1	Expected. The estimated impacts are just a little more than the SIS#. "Little" is relative. We suggest K = .7 here.	Desired: 1 > |SIS#| / |EIS#| ≥ .70 never. Expected: 1 > |SIS#| / |EIS#| ≥ .70 in 40% of a random sample of impact analyses. Goal: 1 > |SIS#| / |EIS#| ≥ .70 in 60% of a random sample of impact analyses.
3 	|EIS#| >> |SIS#|, SIS# $\subset$ EIS#; EIS#, SIS# $\subseteq$ System#	|SIS#| / |EIS#| < K, where K is as in the preceding row	Not good. Big jump from SIS# to EIS# means a lot of things to check in the EIS#.	Desired: |SIS#| / |EIS#| < .70 never. Expected: |SIS#| / |EIS#| < .70 in 55% of a random sample of impact analyses. Goal: |SIS#| / |EIS#| < .70 in 10% of a random sample of impact analyses.

Table 3-4. EIS# and System# Possibilities

Case/Picture	Defining Condition	Measurement When Present	Point Interpretation	Desired Trend
1 	EIS# = System#	|EIS#| / |System# | = 1	Default, not so helpful for impact analysis. But may indicate a system with extreme ripple effect.	Desired: EIS# = System# should never occur. Expected: EIS# = System# in 30% of a random sample of impact analyses. Goal: EIS# = System# in 5% of a random sample of impact analyses.
2 	|System# | > |EIS#|; EIS# $\subset$ System#	J < |EIS#| / |System#| < 1, for some user-selected J such that 0 < J < 1	Better. Change estimated not to affect entire system.	Desired: |EIS#| < |System#| always. Expected: |EIS#| < |System#| in 50% of a random sample of impact analyses Goal: |EIS#| < |System#| in 70% of a random sample of impact analyses.
3 	|System# | >> |EIS#|; EIS# $\subset$ System#	|EIS#| / |System#| < J, where J is as in the preceding row	Even better. Change estimate is restricted to a relatively small subset of the system.	Desired: |EIS#| << |System#| always. Expected: |EIS#| << |System#| in 20% of a random sample of impact analyses. Goal: |EIS#| << |System#| in 25% of a random sample of impact analyses.

paragraphs, the *paragraph* containing the sentences is specified as the SIS# to the IA approach. The IA approach then does IA, resulting in an EIS# of five paragraphs (including the paragraph in the SIS#). Except for the SIS# paragraph, all four other paragraphs must be inspected, meaning 24 sentences must be inspected (6 sentences/paragraph in the figure). In contrast, if the granularity was at the level of sentences, then potentially only the two actually affected sentences (see figure) could have been found. Thus a finer granularity search would allow a potential search savings of 480% (= 24 predicted impacted sentences / (5 actually impacted sentences, including, in this example, the two sentences in the original SIS)).

Table 3-6 illustrates some possibilities with granularity. Similar comments and observations can be made between the relative granularities of the interface object model and the internal object model.

3.4.3. Summary Table

Table 3-7 summarizes the resulting framework elements from IA effectiveness.

4. Classify Systems According To Framework

To illustrate the use of, and provide some justification for, the IA framework, Table 4-1 uses the framework elements to compare five IA approaches. The first is a program slicer represented by the Surgeon's Assistant developed to investigate decomposition slicing as a software maintenance technique [Gallagher1991]. Decomposition here is effectively an impact analysis of the change at the code level. The second is a generic manual cross referencing that detect all references to a given software object (data field, disk file, flag, module, etc.) and assists in understanding relationships between software artifacts.

The third is a traceability system represented by the Automated Life Cycle IA System (ALICIA) [RADC1986]. ALICIA represents one of the first and most comprehensive attempts to address IA with automated traceability mechanisms. The fourth is a Documenting System called Software Document Support (SODOS) environment [Horowitz1986]. It supports the development and maintenance of software documentation. Finally, a commercial tool called the Battlemap Analysis Tool™ (BAT) [McCabe1992] represents a control flow analyzer (among other capabilities). Control flow tools identify calling dependencies, logical decisions (conditions such as IF-THEN-ELSE, LOOPS, CASE statements, etc.), and other control information to examine control flow impacts.

We see two distinct approaches to IA here. The first type (represented by the program slicer, cross referencer, and control flow analyzer) is source code oriented and examines dependencies within the same artifact type. The second type (represented by ALICIA and SODOS) is life-cycle-document oriented and examines dependencies between differing artifact types. Generally speaking, the first type is more mature and provides a finer grained analysis of

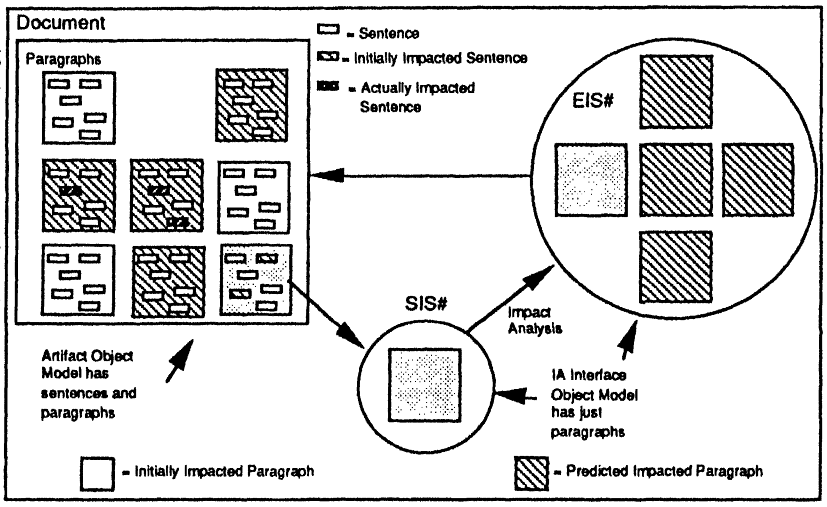

Document
Paragraphs
▭ - Sentence
▭ - Initially Impacted Sentence
▬ - Actually Impacted Sentence
EIS#
SIS#
Impact Analysis
Artifact Object Model has sentences and paragraphs
IA Interface Object Model has just paragraphs
□ - Initially Impacted Paragraph
▨ - Predicted Impacted Paragraph

Figure 3-6. How granularity can influence the search for true impacts after impact analysis.

Table 3-6. Granularity Possibilities			
Case	Artifact Object Model	Interface Object Model	Comments
G1	□	□	Granularities are about equal. Potentially less work needed to translate impacts into the SIS#. Potentially less work needed to translate results from the EIS#.
G2	▤	□	Artifact object model has finer granularity than the interface object model. Potentially more work in discovering true impacts (at artifact object model level) from those predicted in the EIS#.
G3	□	▤	Artifact object model has coarser granularity than the interface object model. Potentially more work is needed to specify fine grained objects in the SIS#.

interpretations.

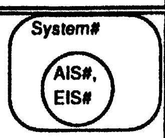
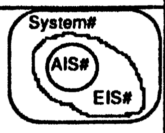
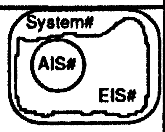
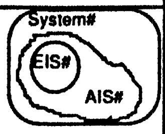
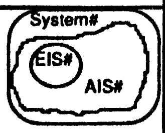
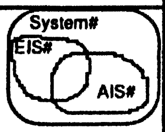
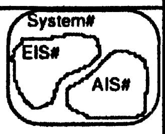
3.4.2.4. Granularity

In practice, the repeated translations from the artifact objects to interface and internal objects, and back, causes several kinds of problems. For example, there is the problem of predicting what artifact objects are actually affected from the EIS# (interface model objects). The translation from the artifact object model to the interface object model and back is usually manual.

Figure 3-6 illustrates this problem. If the artifact object model includes sentences and paragraphs as objects, but the interface object model includes only paragraphs as objects, then we must express a change to a sentence in the artifact model as an impacted paragraph in the interface object model. The resulting EIS# would then include impacted paragraphs. This means work is needed to look inside the impacted paragraphs to find the exact sentences that may be impacted. It also means many more spurious impacts would have to be looked at because the impact bandwidth of a paragraph is often bigger than that for a sentence.

In this example, the SIS has two sentences. Because the IA interface object model can only express impacts with

Table 3-5. AIS# and EIS# Possibilities

Case/Picture	Defining Conditions	Measurement When Present	Point Interpretation	Desired Trend
1 	$EIS\# = AIS\#$; $EIS\# \subseteq System\#$	$\lvert AIS\# \rvert / \lvert EIS\# \rvert = 1$	Best. Estimated impact set matches AIS#. If this happens regularly, usefulness of IA is substantially increased.	Desired: $\lvert AIS\# \rvert = \lvert EIS\# \rvert$ always. Expected: $\lvert AIS\# \rvert = \lvert EIS\# \rvert$ in 10% of a random sample of impact analyses Goal: $\lvert AIS\# \rvert = \lvert EIS\# \rvert$ in 70% of a random sample of impact analyses
2 	$\lvert EIS\# \rvert > \lvert AIS\# \rvert$; $AIS\# \subset EIS\#$; $EIS\# \subseteq System\#$	$H < \lvert AIS\# \rvert / \lvert EIS\# \rvert < 1$, for some user-selected H such that $0 < H < 1$	"Safe". The EIS# contains the AIS#, and the EIS# is not much bigger than the AIS#.	Desired: $\lvert AIS\# \rvert < \lvert EIS\# \rvert$ never. Expected: $\lvert AIS\# \rvert < \lvert EIS\# \rvert$ in 50% of a random sample of impact analyses Goal: $\lvert AIS\# \rvert < \lvert EIS\# \rvert$ in 20% of a random sample of impact analyses
3 	$\lvert EIS\# \rvert >> \lvert AIS\# \rvert$; $AIS\# \subset EIS\#$; $EIS\# \subseteq System\#$	$\lvert AIS\# \rvert / \lvert EIS\# \rvert < H$, where H is as in the preceding row	Safe, but not so good. Big jump from AIS# to EIS# means a lot of things to check in EIS# before arriving at the AIS#.	Desired: $\lvert AIS\# \rvert << \lvert EIS\# \rvert$ never. Expected: $\lvert AIS\# \rvert << \lvert EIS\# \rvert$ in 40% of a random sample of impact analyses Goal: $\lvert AIS\# \rvert << \lvert EIS\# \rvert$ in 10% of a random sample of impact analyses
4 	$\lvert AIS\# \rvert > \lvert EIS\# \rvert$; $EIS\# \subset AIS\#$; $EIS\# \subseteq System\#$	$M < \lvert EIS\# \rvert / \lvert AIS\# \rvert < 1$, for some user-selected M such that $0 < M < 1$	Expected. IA approximates, and falls short of, what needs to be changed.	Desired: $\lvert EIS\# \rvert < \lvert AIS\# \rvert$ never. Expected: $\lvert EIS\# \rvert < \lvert AIS\# \rvert$ in 60% of a random sample of impact analyses Goal: $\lvert EIS\# \rvert < \lvert AIS\# \rvert$, in 20% of a random sample of impact analyses
5 	$\lvert AIS\# \rvert >> \lvert EIS\# \rvert$; $EIS\# \subset AIS\#$; $EIS\#, SIS\# \subseteq System\#$	$\lvert EIS\# \rvert / \lvert AIS\# \rvert < M$, where M is as in the preceding row	Not so good. Big jump from EIS# to AIS# means extra work to discover AIS#.	Desired: $\lvert EIS\# \rvert << \lvert AIS\# \rvert$ never. Expected: $\lvert EIS\# \rvert << \lvert AIS\# \rvert$ in 30% of a random sample of impact analyses Goal: $\lvert EIS\# \rvert << \lvert AIS\# \rvert$ in 10% of a random sample of impact analyses
6 	$\lvert AIS\# \cap EIS\# \rvert > 0$, $AIS\# \neq EIS\#$; $EIS\# \subseteq System\#$	$\lvert EIS\# \cap AIS\# \rvert > 0$	Not so good. Extra work to check EIS# objects that aren't in AIS#. Extra work to discover objects in AIS# not in EIS#	Desired: $\lvert EIS\# \cap AIS\# \rvert$ near 1 most of a random sample of impact analyses. Expected: $.7 < \lvert EIS\# \cap AIS\# \rvert < 1$, in 60% of a random sample of impact analyses Goal: $.9 < \lvert EIS\# \cap AIS\# \rvert < 1$, in 80% of a random sample of impact analyses
7 	$\lvert AIS\# \cap EIS\# \rvert = 0$; $EIS\# \subseteq System\#$	$\lvert EIS\# \cap AIS\# \rvert = 0$	Not so good. A worse version of case 6.	Desired: $\lvert EIS\# \cap AIS\# \rvert = 0$ never Expected: $\lvert EIS\# \cap AIS\# \rvert = 0$ in 20% of a random sample of impact analyses Goal: $\lvert EIS\# \cap AIS\# \rvert = 0$ in 5% of a random sample of impact analyses

impacts while the second type is less mature, but provides a broader analysis.

Table 3-7. Evaluation Parameters for Logical Performance Effectiveness

Element	Explanation	Desired IA Effectiveness Trends
SIS and EIS	What trend is observed in the relative size of the SIS and EIS when the approach is applied to typical problems? We would like the EIS to be as close as possible to the SIS.	|SIS| / |EIS| = 1, (i.e., SIS = EIS), or nearly 1
EIS and System#	What trend is observed in the relative size of the EIS and System# when the approach is applied to typical problems? We would like the EIS to be much smaller than the System#.	|EIS| / |System#| ≤ N, where N is some small tolerance level
EIS and AIS#	What trend is observed in the relative size of the EIS and AIS# when the approach is applied to typical problems? We would like the EIS to contain the AIS#, and the AIS# to equal to or smaller than the EIS.	|EIS| / |AIS#| = 1, (i.e., EIS = AIS#), or nearly 1
Granular-ity	What is the relative granularity of the artifact object model vs. the interface object model? We would like the granularities to match, if possible.	G1, granularities match. (It is even better if granularities are fine enough too.)

5. Relation to Other Work

This paper presents a variety of concepts relating IA technology. We are aware of other work for characterizing change, but have found no applicable evaluation criteria for IA technology.

In one characterization of change, Madhavji describes a broad perspective on change with respect to people, policies, laws, resources, processes, and results [Madhavji1991]. Madhavji distinguishes changes to described items from changes to the environment that houses these items. The Prism Dependency Structure facility supports describing items and their inter-dependencies as well as identifying possible effects of changes. The Prism Change Structure facility supports classifying, recording, and analyzing change-related data from a qualitative perspective.

Our work differs from Madhavji's in that his work focuses on the change process while ours focuses on IA technology and a characterization of IA applications, parts, and effectiveness criteria.

6. Conclusions

IA has many meanings in industry today and the trend is to use the term for more and more situations. In this paper, we have attempted to bring some order and structure to the discussion of IA technology. We recognize the need for a framework to compare IA approaches. This paper presented a definition of IA and a framework that delineates clearly the basis for comparing IA capabilities. The framework

Table 4-1. Comparison of Impact Analysis Systems with IA Framework

Table 4-1a. Framework Category: IA Application

IA Approach	Artifact Object Model	Decomposition	Change specification	Results specification	Interpretation of EIS# to Determine AIS	Other Features
Program Slicer – Surgeon's Assistant [Gallagher1991]	Programs & entities within them	Yes, with some semantics. Knows about programming language objects, control, & data flow relationships.	Yes, with some analysis. User must specify slice criteria & initially affected program, variables, etc	Browsing. Can browse the sliced pieces of the program.	Some effort. Slice approach may compute the slice for the programmer.	Testing, Browsing
Manual Cross Referencing, based on name id & cross reference listings.	Programs, predefined documents	No. Just passively identifies text strings according to match criteria.	No, not applied. User manually reviews cross reference listings	Report. Cross reference listing is the report	Significant. Much effort needed to locate secondary dependencies.	None.
Traceability System – Automated Life Cycle Impact Analysis System (ALICIA) [RADC1986]	DOD-STD-2167 documents & entities within them	Yes, with little semantics. software life cycle objects (SLOs) stored in a RDBMS with a schema that reflects the DOD-STD-2167 relationships.	Yes, with detailed analysis. Uses method. Requirements for a change are analyzed for tracing SLOs.	Report & Database View. ALICIA provides impact report & navigation through a view.	Significant. SLOs are not elaborated nor explained.	Methodology, Document-oriented database schema
Documenting System – Software Document Support (SODOS) environment [Horowitz1986]	Predefined document set & ASCII text.	Yes, with some semantics. Stores decomposed SLOs in a RDBMS based on an object model & predefined relationships.	Yes, with detailed analysis. Req'ts for change are incorpo-rated in B-spec, then analyzed for tracing to SLOs.	Report, Browsing & Database View. Results of IA in hypermedia graph & view for browsing & report.	Significant. SODOS for document management. SLOs not elaborated nor explained thus the user must interpret results	Query – supports traceability with both predefined & user defined relationship Browsing – naviga-tion/edit support.
Control Flow Analyzer – Battlemap Analysis Tool (BAT) [McCabe1992]	Source code programs	Yes, with little semantics. These are limited to decision logic & calling structures.	No, not applied. Control flow analyzers do not support change specification.	Report, Browsing, & Database View. Results stored as a graph & output in a report or editor.	Significant. Control flow impacts not explained by the tool. Interpretation is left to the user.	Call graphs Complexity Metrics Browsing/ editing Execution path slicer

Table 4-1b. Framework Category: IA Parts

IA Approach	Interface Object Model	Internal Object Model	Decomposition	Impact Model	Tracing/ Impact Approach	Repository	Load, Modify, Browse
Program Slicer – Surgeon's Assistant	Program objects such as variables, statements, etc	Data define-use graph & control flow graph	Similar to compiler	Slicing based on data & control flow dependencies	Decomposition slice, program slicing	File system	All three are available
Manual Cross Referencing	Character Strings, i.e., representing variables	None. Just matches characters	None	Matching between character strings	Pattern matching	File system	Not available
Traceability System – ALICIA	DOD-STD-2167 doc's (req'ts, design, code, test, etc.) & input templates	Meta-schema based on DOD-STD-2167 objects	Documents decomposed into relational database	Has user-defined & predefined relationships that have dependency info.	Based on traceability relationships. Heuristic & stochastic impact search algorithms	RDBMS	All three available
Documenting System – SODOS	Documents based on predefined object-based software life cycle	Manages SLOs using object-based model & hypermedia graph. Meta-schema based on predefined document & management objects.	Decomposes documents into object model using templates & filters	Object configuration management & navigation. User-defined & predefined relationships that have dependency information	Based on user-defined & predefined traceability relationships. This enables consistency checking of traceability relationships	RDBMS – Smalltalk-80 object-oriented front-end to RDBMS	All three available
Control Flow Analyzer – BAT	Program objects such as variables, statements, etc.	Control flow graph, calling hierarchies, module structure	Decomposes the code into its control flow elements	Based on control flow dependencies	Not explicit – identifies changes associated with control flow	File system	All three available

Table 4-1c. Framework Category: IA Effectiveness

IA Approach	SIS and EIS	EIS and System#	EIS and AIS#	Granularity
Program Slicer – Surgeon's Assistant	Could not determine from available sources.	Could not determine from available sources.	Could not determine from available sources.	Interface object model is finer-grained. Impacts between programs must be expressed in terms of subprogram entities.
Manual Cross Referencing	Could not determine from available sources.	Could not determine from available sources.	Could not determine from available sources.	Interface object model is finer grained. Impacts between documents must be re-expressed as items that are cross-referenced.
Traceability System – ALICIA	Could not determine from available sources.	Could not determine from available sources.	Could not determine from available sources.	Interface object model is more coarse grained. Impacts between documents must be re-expressed in impact model.
Documenting System – SODOS	Could not determine from available sources.	Could not determine from available sources.	Could not determine from available sources.	Interface object model is more coarse grained. Impacts between documents must be re-expressed in impact model.
Control Flow Analyzer – BAT	Could not determine from available sources.	Could not determine from available sources.	Could not determine from available sources.	Interface object model is finer grained. Impacts between programs must be expressed in terms of subprogram entities.

consists of three parts: IA Application, IA Parts, and IA Effectiveness. To demonstrate the use of this framework, we classified five existing IA technologies according to its criteria.

We believe this work will be helpful to those desiring to investigate the functionality of existing IA approaches. For those hoping to compare approaches, the framework should provide plenty of useful differentiators.

Bibliography

[Freedman1981] Freedman, D. P. and G. M. Weinberg, "A Checklist for Potential Side Effects of a Maintenance Change," In G. Parikh, Techniques of Program and System Maintenance, 1981, page(s) 93 - 100.

[Gallagher1991] Gallagher, K.B. and J.R. Lyle, "Using Program Slicing in Software Maintenance," IEEE Transactions on Software Engineering, Volume 17, Number 8, August 1991, page(s) 751 - 761.

[Horowitz1986] Horowitz, E. and R. Williamson, "SODOS: A Software Document Support Environment - Its Definition," IEEE Transactions on Software Engineering, Volume SE-12, Number 8, August 1986, page(s) 849 - 859.

[IEEE1983] IEEE, "Glossary of Software Engineering Terminology," Std. 729-1983, In IEEE, Software Engineering Standards, Third Edition, New York: IEEE, 1989., 1983.

[Madhavji1991] Madhavji, Nazim H., "The Prism Model of Changes," Proceedings of the International Conference on Software Engineering, IEEE Computer Society Press, 1991, page(s) 166 - 177.

[McCabe1992] McCabe & Associates, Inc., "Battlemap Analysis Tool Reference Manual," McCabe & Associates, Inc., Twin Knolls Professional Park, 5501 Twin Knolls Road, Columbia, MD, December 1992.

[Pfleeger1991] Pfleeger, Shari L., Software Engineering: The Production of Quality Software, New York, Macmillan Publishing Co., 1991.

[RADC1986] Rome Air Development Center, "Automated Life Cycle Impact Analysis System," RADC-TR-86-197, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, Rome, NY, December 1986.

[Stevens1974] Stevens, W., G. Meyers, and L. Constantine, "Structured Design," IBM Systems Journal, Volume 13, Number 2, 1974.

[Yau1978] Yau, S.S., J.S. Collofello, and T.M. MacGregor, "Ripple Effect Analysis of Software Maintenance," Proc. of COMPSAC, Washington, DC: IEEE Computer Society Press, 1978, page(s) 60 - 65.

[Yau1980] Yau, S.S. and J. Collofello, "Some Stability Measures for Software Maintenance," IEEE Transactions on Software Engineering, Volume SE-6, Number 6, November 1980, page(s) 545 - 552.

The Year 2000 Problem: Impact, Strategies, and Tools

Assessing and acting on your "Year 2000 Problems" in 1996 could save your organization costly embarrassments later on

Dr. Robert S. Arnold

February 12, 1996

Software Evolution Technology, Inc.
12613 Rock Ridge Rd.
Herndon, Virginia 22070
USA
703-450-6791
rarnold@sevtec.com

January 1, 2000 is less than four years away. When that happens, will your business stop working? It could if your systems handle dates with two-digit year fields. And if you don't have such systems, it could still happen to you. Especially if the systems or software packages, that you depend on, have the problem. Acting now could help you stay in business when year 2000 is upon you.[1]

THE PROBLEM

The Year 2000 Problem is the problem of assuming an insufficient number of digits in a year field in programs or data. For example, the year 1995 is represented as "95." The year "1989" is represented "89." An so on.

Such a simple assumption. Why all the ruckus?

Say it's January 12, 2000 and your company sends out an invoice that day. Your accounts receivable legacy software represents years with two digits. So January 12, 2000 is represented as 01/12/00 (MMDDYY in your data). But your legacy code thinks this means January 12, *1900*. Yes, it should be year 2000, but your old, two-digit, code does not take this into account.

Now suppose your customer does not pay the bill by March 12, 2000. Your accounts aging software scans the data base to see what accounts receivable are due. It sees the January 12, (19)00 invoice. Since the invoice is over a 100 years old (!), the software could call this a really bad debt and cancel it!

The Year 2000 problem could be a problem well *before* the year 2000 arrives. Suppose it is July 1, 1999 and you decide to grant Sally a pay raise on January 1, 2000. The effective salary raise date, 1/1/00, is input to the code. The software may grant the pay raise *immediately* because 2/1/00 precedes the current date, 7/1/99, making the raise immediately effective! She might even receive back pay dating from 1900!

Similar problems can happen in other applications having dates of effect, such as bank interest rates, insurance policy premium rates, pension plan annuity checks, and shipping rates.

The Year 2000 Problem can affect any program that represents or assumes years are represented with only two digits, any data bases with two-digit year fields, and any query or screen that has two-digit year fields. It can also

[1]This paper is revised and updated from these papers by the same author: "Millennium Now: Solutions for Century Date Change Impact," *Application Development Trends*, Volume 2, Number 1, January 1995, pp. 60-62, 64-66; "Year 2000 Tools," *Year 2000 News*, Issue 1.1, August 4, 1995. Both papers are copyright © 1995 Software Evolution Technology, Inc. All rights reserved.

affect programs, systems, data bases, data base queries, etc. that depend on these programs and data.

In practice, COBOL and assembly code legacy systems are susceptible, though code in almost any language can have this problem. For example, a programmer could have built this assumption into C or C++ code.

Common sources for possible Year 2000 Problems are COBOL Procedure and Data divisions, CICS screens featuring date-oriented or time duration fields, copybooks, database queries, ISPF screens, and JCL procedures.

Are these risks worth it? Tom Oldenburger, Supervisor of Application Support with a health maintenance organization in California, feels that if the Year 2000 is not solved, all of his systems would be affected and business could stop. For him, the risks are not worth chancing.

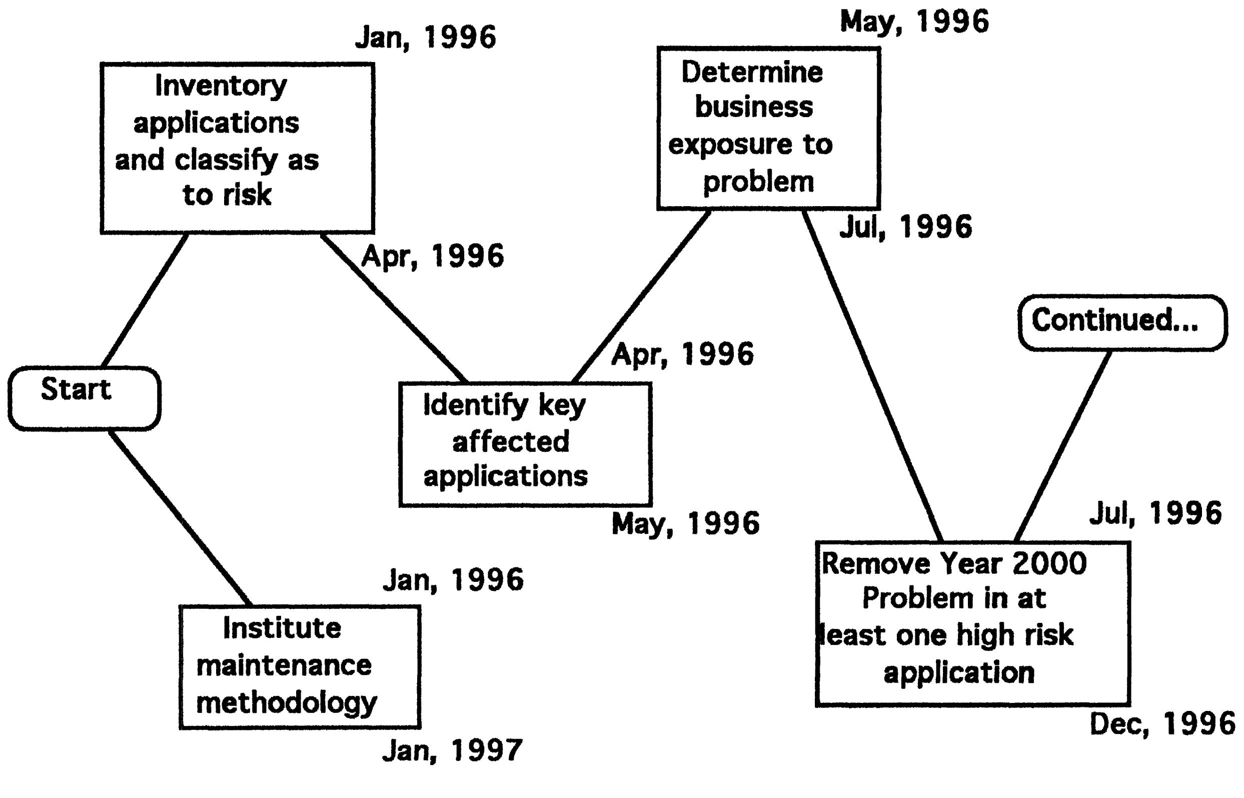
WHY NOW?

"But January 1, 2000 is four years away. Why worry about this now?"

There are several reasons maintenance managers give for not acting now. Some reasons and responses are:

• "It will go away." The system will not be around on 01/01/2000. Why fix a problem that will not occur?

Systems tend to stay around longer than we expect. Some systems are rewrites of previous systems, perpetuating their errors. Someone may salvage this code in the future, not realizing it could have a Year 2000 bug in it. In the meantime, these systems may be outputting data with 2-digit years. When this is fixed there will be a cost for every 2-digit year you produce. Why perpetuate a reengineering problem?



Start
Jan, 1996
Inventory applications and classify as to risk
Apr, 1996
Apr, 1996
Identify key affected applications
May, 1996
May, 1996
Determine business exposure to problem
Jul, 1996
Jul, 1996
Remove Year 2000 Problem in at least one high risk application
Dec, 1996
Continued...
Jan, 1996
Institute maintenance methodology
Jan, 1997

Figure 1. (1 of 2) High Level Steps In Year 2000 Problem Resolution. This is a conservative approach to assessing potential Year 2000 Problems in an application, and acting on them if necessary. Table 1 explains the steps. Time durations for the steps are illustrative only, and will need to be adjusted for each Year 2000 Problem resolution project.

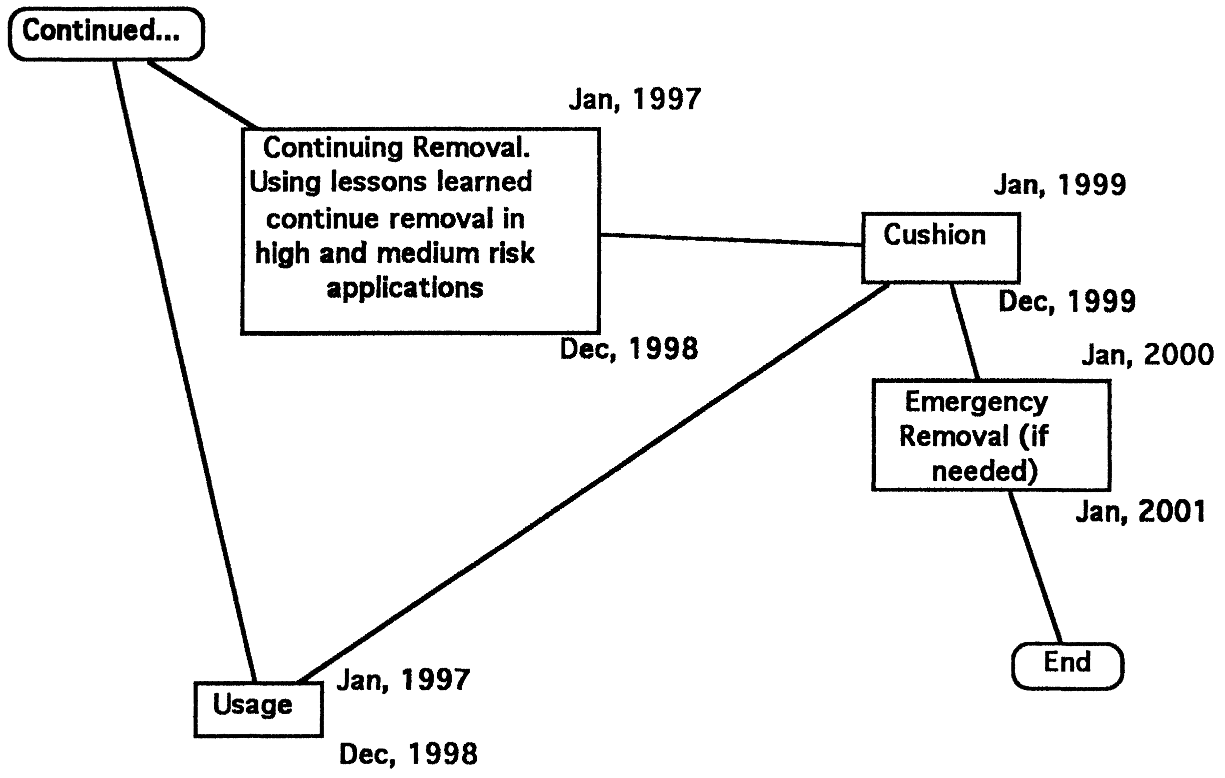

Continued...
Jan, 1997
Continuing Removal. Using lessons learned continue removal in high and medium risk applications
Dec, 1998
Jan, 1999
Cushion
Dec, 1999
Jan, 2000
Emergency Removal (if needed)
Jan, 2001
End
Usage
Jan, 1997
Dec, 1998

Figure 1. (2 of 2) High Level Steps In Year 2000 Problem Resolution.

• "Not on my watch." I probably won't be at this job on 1/1/2000. Why put my short term budget in jeopardy to solve this longer term problem?

The impact on your budget should be based on concrete analysis. It may not be as bad a problem as you expect. But you should determine your organizations exposure. A longer term view in this case could prevent panic fixing later on.

• "Too busy." We have our hands full with maintenance, why add to my backlog by putting the Year 2000 Problem on my plate?

Yes, it is a matter of priorities. But it is marvelous how a potential disaster can change this. You may be betting the farm without knowing it.

• "I really don't want to know." The problem could become a can of worms once we do some checking. If it is, management may question why I didn't look at this earlier.

Professional managers do not hide from reality, they deal with it. Superiors appreciate people that recognize a potentially serious problem and start to act on it. Unless they knew better, but didn't.

• "Just expand the year field. What's the big deal?" This is not as serious problem to solve as you're making it out to be.

A fair amount of reengineering may be involved to fix the problem. You may need to change data schemas, raw data, input screens, procedural logic, and so on. You will not know how much work is really involved until you actually roll up your sleeves and check for yourself.

But why act *now*, rather than six months or a year from now?

For some organizations, acting now is not mandatory. But it could save reengineering work later, particularly with data. For other organizations, delay could be deadly. The longer you delay, the harder (i.e., more costly) it can be to reengineer your code later. Just look at the new Denver airport baggage sorting system.

STRATEGIC PLANNING

Because the risks to your organization can be severe, and because Year 2000 Problems can occur long before 1/1/2000, you should at least determine your organization's exposure at this time.

To get concrete answers, you will need to understand the extent of the actual problem in your environment, what your solution options are (with cost-

Date	Activity	Explanation
1-4/1996	Inventory applications and classify as to risk	In your software portfolio, determine what applications are possibly affected by the Year 2000 Problem. Determine business or operations impact should these applications have serious Year 2000 Problems. Identify applications as high risk, medium risk, low risk.
4-5/1996	Identify key affected applications	Target high risk systems for Year 2000 Problem removal. Create plans for transitioning legacy code and data to use 4-digit years.
1/1996 - 1/1997	Institute maintenance methodology	If you have not already done so, institute procedures to avoid further propagating Year 2000 Problems. (This is a good idea even if you do not have Year 2000 Problems.)
5-7/1996	Determine business exposure	Determine your business exposure to the Year 2000 Problem. This means you have valid estimates of the real problem at your site. You then estimate what would happen to your business should serious problems actually occur. Will a problem bring your business to a halt? Require extra reengineering later on? Just be an occasional nuisance?
7-12/1996	Remove Year 2000 Problem in at least one high risk application.	Do this to get first hand data about how difficult the problem will be for you. This will help you gauge the difficulty in solving the problem for other systems. Delaying this could leave you with little time, before 2000, for removing the problem in all high risk applications.
1/1997 - 12/1998	Continuing removal	Eliminate Year 2000 in other high risk or medium risk applications. This may extend into 1999.
1/1997 - 12/1998	Usage	Some corrected applications should now be in production, especially those already encountering 21st century dates. Fix any Year 2000 Problems not caught by previous fixes. Feed this experience back into other Year 2000 removal efforts. Eliminate problem in low risk applications, as resources allow.
1/1999 - 12/1999	Cushion	This year is for slack in case you need more time to remove Year 2000 Problems in your applications.
1/2000 - 1/2001	Emergency removal (if necessary)	You should allow some time here in case you choose to delay and then do discover critical Year 2000 Problems in your software. Depending on Year 2000 outsourcing specialists to help you at this time could be problematic if outsourcers have their hands full with other customers needing help with Year 2000 Problems.

Table 1. Time Line for Resolving Year 2000 Problems in Your Applications. This table explains the steps in Figure 1.

benefit projections), and what your organizational exposure is in case you choose to do nothing now.

Figure 1 shows how checking your exposure now can be part of a more comprehensive schedule for resolving Year 2000 Problem difficulties.

This is a conservative plan. (Riskier plans will delay the steps.) It features determining true risk early on, removing Year 2000 Problems from one high risk application early on, and providing plenty of slack in 1997-1999 in case solving the problems are stickier than expected. Table 1 explains the steps in Figure 1 in more detail.

Tom Miele, Manager of Development Technologies at a data processing center supporting several health insurance providers in Pennsylvania, received a directive from top management to determine the extent of the Year 2000 Problem. This made eliminating the problem a matter of business importance, not just of software maintenance. He has completed a preliminary Year 2000 Problem impact study, with Viasoft's help. His goal is to be Year 2000 Problem "clean" in all affected applications by 1998. 1999 will be used to fix any remaining problems.

FINDING AND REMOVING YEAR 2000 PROBLEMS

Figure 2 presents a baseline process for finding and removing Year 2000 Problems from code and data. It just presents basic steps; many refinements are possible.

The process is to first place your system under control so that uncontrolled change does not introduce new problems while you are fixing Year 2000 Problems. Next comes an initial estimate of the cost of Year 2000 Problem removal. This will help temper your expectations as the work proceeds. Finding problem code looks at your code and data to discover date-oriented problems.

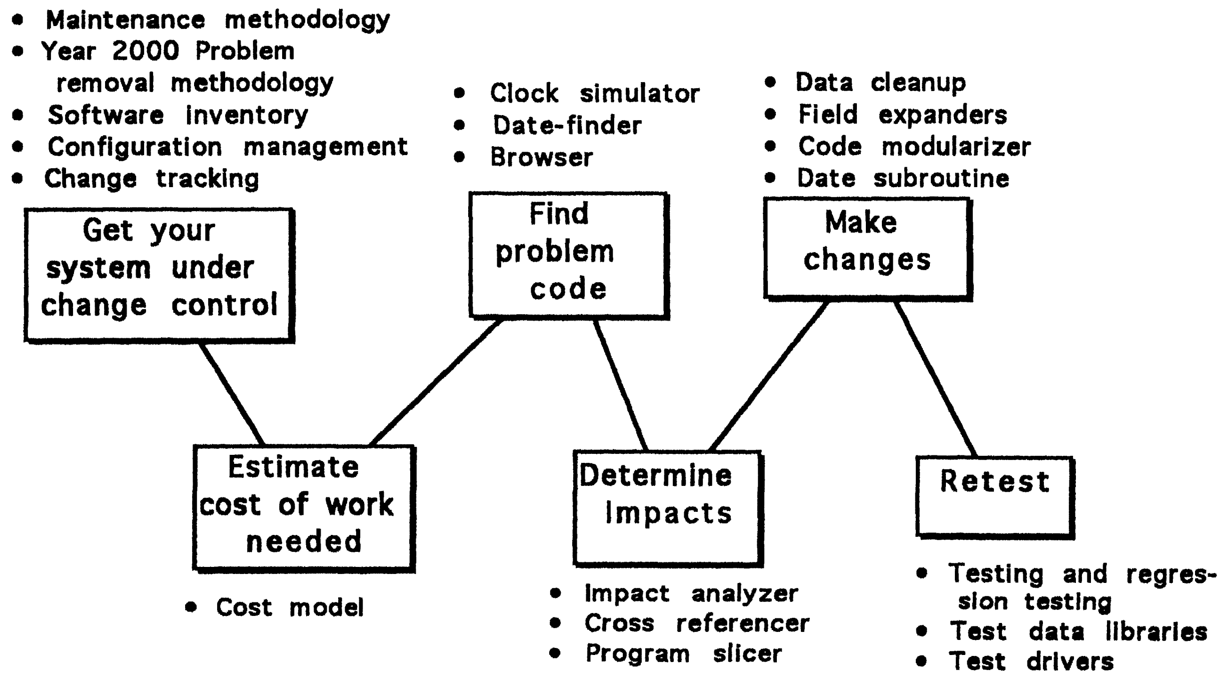


Figure 2. Process and Technology for Eliminating Year 2000 Problems. Year 2000 Problems can be systematically reduced. This figure shows the process steps and supporting technology. Table 2 discusses the technology and gives examples of commercial technology.

Table 2. A Year 2000 Problem Removal Tool Kit

Classes of tools useful for solving Year 2000 Problems are listed here, along with sample commercially available technology. Technology specifically positioned for solving the Year 2000 Problem is emphasized. Most technology listed here works on COBOL legacy code and data. The technology is grouped by the Year 2000 Problem removal step, as shown in Figure 2, to which the technology applies. Some technology can be used with more than one step, even though it is not so listed. Because of this, the table should not be used for product comparison purposes. (Much more technology is available than shown here.)

1. GET YOUR SYSTEM UNDER CHANGE CONTROL

1.1 Maintenance Methodology

Having a sound approach to software maintenance will put order and discipline into software change. This will help in many areas, including instituting standards to lessen the insertion of Year 2000 Problems into your code. See [Hayes1994] for a recent discussion and list of available maintenance methodologies. Another maintenance methodology is the IEEE standard on software maintenance [IEEE1993].

1.2 Year 2000 Problem Removal Methodology

Several methodologies for approaching removal of Year 2000 Problems have recently appeared. These can save you time because they have already thought through how to systematically resolve Year 2000 Problems. *Year 2000 News*[2] also features articles on Year 2000 Problem methodology.

SEVTEC Reengineering Process Library	Software Evolution Technology (SEVTEC)
Viasoft's ENTERPRISE 2000	Viasoft
System Vision Year 2000	Adpac

1.3 Software Inventory

Inventorying your software will help you determine all the code, JCL, data bases, etc. that constitute your system. This is important for providing a complete system on which to for impact analysis. These tools can help you do this.

Century Source Conversion	Quintic Systems, Inc.
GILES	Global Software
VIA / Alliance	VIASOFT
Xpediter / Xchange	Compuware Corporation
System Vision Year 2000	Adpac

1.4 Configuration Management

Configuration management can turn your software into configurations. This is important as you modify and eliminate bugs in the code. You can then systematically record what software you have fixed and what needs fixing. Here are some sample configuration management tools.

Aide-de-Camp	Software Maintenance & Development Systems, Inc.
CCC	Softool Corporation
ChangeVision for SoftBench	Hewlett-Packard
Change Man	Optima Software, Inc.
ENDEVOR	LEGENT Corporation

[2]*Year 2000 News* is an Internet newsletter on the Year 2000 Problem. To subscribe, contact Robert Arnold at rarnold@sevtec.com or 703-450-6791.

PVCS	Intersolv, Inc.

1.5 Change Tracking

Change tracking will log requests for changes and track them to resolution. The change tracking database can be a useful place to look for changes regarding dates or bugs concerning dates. You can use an off-the-shelf tool, or create a change-tracking database application easily enough.

AWARE!	Integritech Systems, Inc.
CaseWare/PT	CaseWare
CCC/Pro	Softool Corporation
InfoTrak	Trident Software
IPM/MVS, IPM/PC	The Orcutt Group
Maestro II	Softlab

2. ESTIMATE COST OF WORK NEEDED

Cost Model

This tool, spreadsheet, or methodology predicts how much work will be needed to remove Year 2000 Problems. Check how the model you intend to use has been validated. That is, its predictions compared with actual human performance in removing the Year 2000 Problem. Because carefully measured empirical data on finding and removing Year 2000 Problems is currently hard to find, the cost model may not been separately calibrated with valid data collected from real Year 2000 Problem removal projects.

SEVTEC Reengineering Process Library	Software Evolution Technology, Inc.
IMPACT 2000	Viasoft
System Vision Year 2000	Adpac Corporation

3. FIND PROBLEM CODE

3.1 Clock simulator

These change your system clock, unknown to your programs. They are an easy way to quickly check if your code has Year 2000 Problems.

HourGlass 2000	MainWare, Inc.
TICTOC	Isogon Corporation
VIA / ValidDate	VIASOFT
Portal 2000	Prince Software Products
Xpediter / Xchange	Compuware Corporation

3.2 Date-finder

These tools or lists of names help you locate date-oriented data, variables, or information in the code.

Century Source Conversion	Quintic Systems, Inc.
General Purpose Systems Analyzer (GPSA/PC)	COBOL Maintenance Technologies
SE/ONE (D-Ray)	Software Eclectics, Inc.
System Vision Year 2000	Adpac Corporation

3.3 Browser

Browsing tools allow you to scan through code and inspect. Scanning can be speeded by looking at structure charts, selecting code, and immediately jumping to the code. They can save a lot of time over just using a text editor. Browsers can include some reverse engineering tools or maintenance workbenches.

COBOL / Softbench	Hewlett - Packard
General Purpose Systems Analyzer (GPSA/PC)	COBOL Maintenance Technologies
GILES	Global Software
InterCASE Reverse Engineering Workbench	Intersolv
Legacy Workbench	KnowledgeWare
McCabe Tools	McCabe & Associates
Micro Focus COBOL Workbench	Micro Focus
Refine / COBOL	Reasoning Systems
Revolve	Burl Software Laboratories / Micro Focus
SEEC/CARE Cobol Analyst	SEEC

4. DETERMINE IMPACTS

4.1 Impact Analyzer

These essential tools do the hard work of finding what is related to what. They can save you hours of grunge work. But they are subject to accuracy and "safety" limitations. Just because an impact analysis tool estimates that an item is affected, does not mean that the item will need to be changed to fix a specific Year 2000 Problem. Impact analysis tools will tend to overshoot their estimates of what is affected, to be on the safe side. The extent of the this overshoot, the difference between what actually needed to be changed and what was originally estimated to be changed, is not known.

McCabe Slice Tool	McCabe & Associates
Revolve	Burl Software Laboratories / Micro Focus
SEEC/CARE Cobol Analyst	SEEC
PM/SS, System Vision Year 2000	Adpac Corporation
Via / Alliance, VIA / Insight	Viasoft

4.2 Cross Referencer

These tools tell you where variables, procedures, or other items are in the code. They are helpful in speeding up browsing and providing a limited form of impact analysis.

Revolve	Burl Software Laboratories / Micro Focus
SEEC/CARE Cobol Analyst	SEEC
SE/ONE	Software Eclectics, Inc.
PM/SS, System Vision Year 2000	Adpac Corporation
Via / Insight, Via / Recap	Viasoft

4.3 Program Slicer

Code slicing tools--a powerful type of static impact analyzer--help you see all the code affecting a given variable or statement. Forward slicing starts with a name or statement, and tells you what it affects. Backward slicing also starts with a name or statement, but tells you all parts of the program that could affect it. Despite their power, the tools do not replace the need for regression testing.

INTERSOLV Maintenance Workbench	INTERSOLV
VIA / Renaissance	VIASOFT
REFINE / COBOL	Reasoning Systems

5. MAKE CHANGES

5.1 Data Cleanup

These tools help you improve the consistency and accuracy of your data. Sometimes this is done in the context of extracting and coverting data from a non-relational to a relational DBMS.

Bachman / Re-engineering Product Set	Bachman Information Systems

Extract Tool Suite	Evolutionary Technologies, Inc.
Legacy Data Mover	Legent
Pacreverse	CGI Systems
Warehouse Manager	Prism Solutions Warehouse Manager
Integrity	Vality Software
DATATEC	Compuware Corporation

5.2 Field Expanders

These tools help you automatically expand 2-digit year fields into 4-digit year fields. Some data cleanup tools (mentioned above) can also be used.

Century File Conversion	Quintic Systems, Inc.
Vantage YR2000	Great American Insurance Companies

5.3 Data Name Rationalization

These tools help you introduce standard or more uniform names to date-oriented fields.

DATATEC	Compuware Corporation
$NAME	Global Software
Revolve	Micro Focus
PM/SS	Adpac Corporation

5.4 Code Modularizer

These tools help you to identify chunks of code and may wrap an interface around them, making them into procedures or modules. Fall-throughs into this code are replaced by calls to it. This is helpful because certain changes in the module can be hidden without fear that other parts will be affected.

Via/Renaissance	Viasoft
REFINE / COBOL	Reasoning Systems

5.5 Date Subroutine

These subroutines are reusable code modules that have correctly implemented the handling of dates. If you're worried about the accuracy of your date handling, these tools could help.

COBOL Language Environment/370	IBM
TransCentury Calendar Routines	Transcentury Data Systems

6. RETEST

6.1 Testing and Regression Testing

Testing is essential for checking whether software changes introduced unwanted problems. Since automatic impact analyzers do not check impacts in all possible inter-code relationships (e.g., timing), testing is nearly always a necessary part of software change.

Automator QA	Direct Technology
CICS/Replay	Interactive Solutions, Inc.
CA-DATAMACS	Computer Associates International
QES/Architect	QES, Inc.
Playback	Compuware Corporation
SQA Manager	Software Quality Automation
SQA Robot	Software Quality Automation
STW/REG	Software Research, Inc.
CA-VERIFY	Computer Associates International

6.2 Test Data Libraries

These tools help you organize your test data for use and reuse.They often are part of test and retest tools, or you can create and manage your own.

6.3 Test Drivers

These tools take a lot of the drudgery out of running test cases and comparing actual with expected results. They are often part of test and retest tools.

[End of Table 2. A Year 2000 Problem Removal Tool Kit]

These initially discovered problems "prime the pump" for automatic impact analysis, which is the next step. Once you have an estimate of all affected code and data, you use this to plan and perform a systematic solution in your application. Finally you perform a retest since current impact analysis technology does not guarantee all possible date-oriented problems are revealed.

Table 2, "A Year 2000 Problem Removal Tool Kit," presents technology associated with each step in Figure 2.

IMPACT ANALYSIS METRICS AND THE YEAR 2000 PROBLEM

Articles have been published that estimate the cost to correct the Year 2000 Problem in the industry to be in the millions. For specific decision-making on the front lines of software maintenance, such numbers are not very helpful. Managers need to have specific, reliable estimates tailored for *their* environments. This information is important for deciding whether to act now or later.

One way to estimate how much of your code is affected by the Year 2000 Problem is to use an impact analysis tool. Figure 3 shows the steps, which are expanded in Table 3. The steps are based on the impact analysis capability comparison framework described in [Arnold1993].

To use an impact analysis tool to estimate the items you will need to change to solve a Year 2000 Problem, first you find date-oriented items in your code likely to be affected. This initial set of affected items is then input to the impact analysis tool. (For example tools, see Table 2.) The tool estimates other affected items that you should check. This estimate is used to gauge the size of your Year 2000 Problem.

Currently impact analysis tools have limitations that make their usefulness, *as estimators of the effort needed to solve a Year 2000 Problem*, unclear. You may be tempted to run an impact analysis tool, discover that the tool predicts most of your code is affected by a Year 2000 Problem, and conclude you have a serious problem on your hands. Maybe, but maybe not!

What we do not have yet are clear measurements of "safety" and "scope" of the analysis that impact analysis tools do. (The actual metrics are defined in [Arnold1993].) "Safety" checks whether the tool regularly includes, in its impact predictions, *all* the code that will *actually* be modified when the year 2000 is corrected. That factor will give you more confidence that the tool's impact analysis is reasonably complete for your purposes."Scope" checks the extent to which the tool's impact predictions are excessive. This influences the amount of extra work you will need to do when checking affected code and data.

Impact analysis tools will tend to be conservative (overshoot) their estimates of what must actually be changed, to be on the safe side. The extent of the this overshoot, the difference between what actually needed to be changed and what was originally estimated to be changed, is not known.

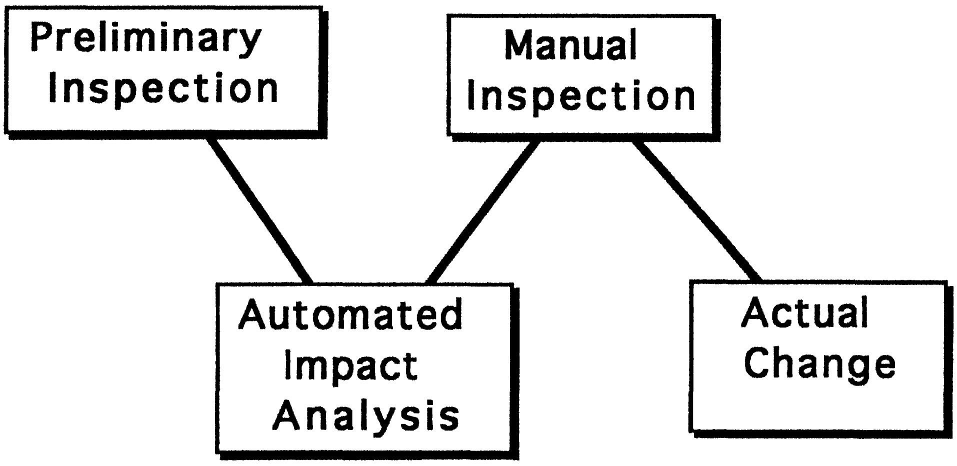


Figure 3. Basic Steps in Code-Level Impact Analysis. Impact analysis proceeds systematically from finding initially affected date-oriented items, to estimating other items that are affected, to planning and performing the change that reduces Year 2000 Problems. Table 3 expands on the steps listed here.

Unfortunately, most impact analysis tools today do not provide these impact analysis measurements. You will have to calculate them yourself. So be careful when using impact analysis tools to estimate the effort to resolve your Year 2000 Problems.

EARLY ASSESSMENT IS KEY

There is much technology and planning expertise available for resolving Year 2000 Problems in your applications. Whether you choose to act now or later can influence how your business will operate in several years. Certainly your business flexibility can be impacted, and embarrassing and costly problems could be coming your way. Assessing the problems at this time is key to covering yourself before disaster strikes.

REFERENCES

[Arnold1993] Arnold, Robert S. and Shawn Bohner, "Impact Analysis - Towards a Framework for Comparison," *Proceedings of the Conference on Software Maintenance*, Los Alamitos, CA: IEEE Computer Society Press, September 1993.

[Hayes1994] Hayes, Ian S., "Project Management: How It Can Improve the Maintenance Process," *Application Development Trends,* October 1994, page(s) 48 - 50, 52, 54 - 57, 60 - 61.

[IEEE1993] IEEE, "IEEE Standard for Software Maintenance," Standard 1219-1993, IEEE, 345 East 47th St., New York, NY 10017, June 2, 1993.

Step	Activity	Example
1. Preliminary Inspection	Done manually, with the help of browsing software or date-name finding software. The goal of this work is to "prime the pump" with entities (variables, statements) that are known to be date-oriented. This set of entities is called the "starting impact set."	Preliminary inspection finds 38 program variables that are date-like. These are contained in 5% of the system's lines of code. This example uses "percentage of non-commentary source lines of code" as the measure of impact.
2. Automated Impact Analysis	The starting impact set is input to the automated impact analysis tool. This may be done all at once, or piecemeal, depending on the tool. The tool then estimates the lines of code, or other entities, that are potentially affected. This collection of entities is called the "estimated impact set."	The tool predicts that 75% of the code is potentially affected by changes to the 38 variables. This 75% is what is most often used to estimate the size of your Year 2000 Problem. The estimate could vary from tool to tool.
3. Manual Inspection	This uses the impact estimate as a guide for checking the code manually. This is needed because impact analysis results do not imply that everything must be changed, just what is potentially affected. It is useful for creating a plan for systematically removing Year 2000 Problems in the code.	
4. Actual Change	The Year 2000 Problem is removed. The collection of entities actually modified is called the "actual impact set."	It is found that changes are needed in code lines comprising 25% of the lines of code. 8% was not in the estimated impact set.

Table 3. Basic Steps in Code-Level Impact Analysis. The steps are based on the impact analysis capability comparison framework described in [Arnold1993].