Introducing Software Engineering Methods into Industrial Practice: Module Interface Specification and Inspection

by

Ann M Jackson

B Sc , University of British Columbia, 1977

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr D M Hoffman, Supervisor (Department of Computer Science)

[redacted]

Dr M H van Emden, Departmental Member (Department of Computer Science)

[redacted]

Dr J R Moehr, Outside Member (School of Health Information Science)

[redacted]

Dr V K Bhargava, External Examiner (Department of Electrical and Computer Engineering)

Supervisor Dr Daniel M Hoffman

Abstract

Despite dramatic changes in computing in the two decades since the terms *software crisis* and *software engineering* were coined, problems of deficient quality and unmanageable costs continue to afflict the software industry Improvements in the software engineering process—methods, rather than tools—are now widely considered essential to bringing software quality and costs under control

Diffusion of software process innovations in industry has occurred only slowly to date, leaving a backlog of largely untried research proposals in need of dissemination and evaluation This thesis comprises a case study of a pilot project that introduced two such credible but less-known software engineering methods into an industrial setting

Techniques of module interface specification and inspection were selected and adapted to the context, according to a model of software technology transfer In order to exploit the potential of the inspection process for verifying work product properties, a verification-oriented approach was devised for paraphrasing module interface semantics during the inspection meeting

The new methods were introduced via a carefully designed training program, and were positively received by practitioners Process, product and defect metrics were defined and a measurement program initiated, to support objective evaluation of these technologies, and ongoing software process improvement, in the longer term Both participant response and preliminary examination of the data collected during the pilot project, indicate that these methods are effective in industrial application

Examiners

[redacted]

Dr D M Hoffman, Supervisor (Department of Computer Science)

[redacted]

Dr M H van Emden, Departmental Member (Department of Computer Science)

[redacted]

Dr J. R Moehr, Outside Member (School of Health Information Science)

[redacted]

Dr V K Bhargava, External Examiner (Department of Electrical and Computer Engineering)

Contents

List of Figures

List of Tables

Acknowledgements

This thesis has many sources of inspiration, collaboration, and support. Thanks are due first to my supervisor, Dr. Daniel Hoffman, who devoted generous amounts of time to teaching me about the software engineering techniques described herein. He was a steadfast source of insight, encouragement, and practical guidance when the way forward to completing this project seemed difficult and uncertain.

The Special Service Networks department of MPR Teltech Ltd. extended me the rare privilege of experimenting with software engineering practice in industry. Particular thanks are due to the following individuals. David Reid negotiated and authorized the project and funding. The suggestion that we tackle design documentation—a much more interesting problem than the work on code inspection initially proposed—was his. Susan Wong's oversight ensured the project stayed on track. She was generous in responding to requests for information and assistance, and enriched our efforts at collecting inspection data by integrating them with her metrics project. Shirley Louie was the start-up participant in our training program. She graciously allowed me to try out my less-refined ideas on her, worked very hard at proving out the technology, and voluntarily coordinated a lot of inspection follow-up work. As technical leader, Larry Leblanc withstood overwhelming pressures of *real* work schedules to allocate people and time (including much of his own) to our project activities. Mark Anderson, Ricky Li, and Siegfried Luft were diligent and able participants. Robert Sharkey and Willy Waung, predecessors of Mr. Reid and Ms. Wong, respectively, championed the initial idea of a collaborative research project. Michael Dowling, of Prism Systems, Inc., graciously arranged for us to view company-owned inspection training videotapes at his office.

Dr. D. Michael Miller, Chair of the Computer Science department at the University of Victoria, was unfailingly helpful in dealing with the legal and financial ramifications of my collaboration with MPR Teltech Ltd. Dr. Miller also negotiated adjunct student status at the University of British Columbia for me, when it was convenient for personal reasons to complete my thesis in Vancouver. This unusual assistance is greatly appreciated, as is the help of Shirley Page, Jocelyn Swallow and Ruth Hopkinson of the Computer Science department office, who responded ably to numerous email requests from the out-of-town graduate student.

This thesis was written in its entirety using the graduate student computing facilities of the Computer Science department of the University of British Columbia. I am indebted to Dr. Maria Klawe, Head of the department, for granting me adjunct student status despite the crowded conditions obtaining during construction of a new

building. Thanks are also due to Dr. Jeff Joyce and other members of the Integrated Systems research group, who made me welcome and expressed generous interest in my work.

The hospitality of my parents, Alfred and Cathreen Fischer, emboldened me to enroll at the University of Victoria, and particularly supported me during my first term of study. The temporary separation from my husband was also eased by the companionship of other relatives resident in Victoria, and of the computer science graduate students of 1991–92 (particularly at the Friday Happy Hours).

My husband, Hugo Jackson, has long expressed more faith in and support of my work than I was inclined to accord it myself. The inspiration and courage to pursue this project are truly owed to him, as are more easily expressed thanks for his practical contributions of critiquing presentations, proofreading, and taking on an overload of domestic duties to allow me time to complete writing the thesis.

This work was supported financially by the Natural Sciences and Engineering Research Council of Canada, the University of Victoria, the B. C. Advanced Systems Institute, and MPR Teltech Ltd. This support is acknowledged with much gratitude.

Dedication

Martin Alfred Fischer

1962–1984

In loving memory of my brother, an able and zestful systems programmer who would have taken a lively interest in this work

Rest eternal grant to him, O Lord,
And let light perpetual shine upon him

Chapter 1

Introduction

1.1 The software crisis

The term *software crisis* has been widely used for the past two decades to refer to a well-known litany of problems in the industrial production of software. Software is expensive to create and even more expensive to maintain, late deliveries and significant cost overruns are commonplace and the performance of delivered software is often unreliable or inadequate to user requirements. In a classic book of essays on these problems, Frederick P. Brooks, Jr. wrote in 1972:

> Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems—few have met goals, schedules and budgets. [Bro75, page 4]

In the ensuing two decades, computer software has assumed a central role in much public and private enterprise, including safety-critical systems. Dissatisfaction with the problems of software quality and productivity has accordingly magnified and the discipline of software engineering has emerged in response to this "software crisis," yet the problems persist. Why has software production not been brought under satisfactory control?

Computer professionals readily place their hope in improved hardware and software support, such as workstations, windowing systems and CASE[1] software to address the intractabilities of software production Such tools have aided the development of larger and more powerful software systems than those of the early 1970s, but they have not eliminated the software crisis, as Brooks vividly identifies in a more recent work "There is no silver bullet" [Bro87] Some researchers into software quality and productivity now point to the software development *process* rather than *tools* as the area in need of attention, e g

> the major problems associated with the production of trustworthy software are more associated with the *organization and management of complexity* than with direct technological concerns that affect individual programmer productivity [SP90, page 8]

> On the technology side, relatively simple tools and methods, if well managed, are all that are needed It may be that the development of the Japanese software industry is due more to good management than to Fifth Generation research [BM91b]

> The effective use of software technology is limited by several factors an ill-defined process, inconsistent implementation and poor process management [Hum89, page ix]

1.2 The problem: immature industrial practice

In the late 1980s, the Software Engineering Institute at Carnegie Mellon University (SEI) developed a 5-level "software process maturity framework" for assessing the software engineering capability of software contractors for U S Department of Defence projects Subsequent studies showed some 85% of contractors to be at level 1 (termed *initial* or *chaotic*) and most of the rest at level 2 (*repeatable*) Reports of self-assessments by other software organizations agree that the great majority

[1] Computer Aided Software Engineering

of software organizations lack a repeatable, let alone *defined* (SEI level 3) software process. This state of affairs not only restricts industrial capabilities but will increasingly threaten the very survival of North American software producers in the face of intensifying international competition [BM91a, You92].

The chaotic state of industrial practice is no accurate reflection of the state of software engineering knowledge. Viable methods to tackle many problems of the software development process have already been elaborated, but are not yet widely used. Indeed, the transfer and diffusion of software engineering methods in industry has proven to be such a slow and difficult process that it has itself attracted study [RR85, Chi88, RC89, BM91b]. Our contribution is therefore concerned with addressing the gap between industrial practice and existing best-known methods.

1.3 The approach

> *No technology will transition into widespread use unless there is a recognized need, a receptive target community and believable demonstrations of cost/benefit.* [RR85]

The work described in this thesis was made possible by the supportive collaboration of MPR Teltech Ltd. of Burnaby, a leading high-technology enterprise in British Columbia, where the author was formerly employed in software development. MPR Teltech's support of this project was noteworthy because the absence of strong links between academia and industry is a significant obstacle to software engineering technology transfer in Canada.

Our approach was to identify some practical software engineering methods both offering significant benefit and suitable for immediate application at MPR Teltech. We adapted the methods to the particular requirements of that setting and provided training and assistance with early use (a pilot project). Data collection activities were initiated to support long-term objective evaluation of the methods. At the

end of the pilot project, we produced an experience report for the company. The objectives of this work were to gain insight both into the methods deployed and into the practical issues of software engineering technology transfer.

1.3.1 Industrial setting

MPR Teltech Ltd. is an advanced telecommunications company providing design, consulting, product development and integration services to an international portfolio of customers and partners. Product areas in which the company is active include network management, ISDN and satellite communications. A subsidiary of BC TEL headquartered in Burnaby, BC, MPR Teltech has the largest industrial research and development capability in western Canada and employs some 500 people at three sites across North America.

Within MPR Teltech, our collaborator was the Special Service Networks (SSN) department of the Digital Products division. SSN has about 60 employees, who work largely on digital network management software. The work of the department is primarily in product development rather than research, and its ethos reflects this in a strong orientation toward effective application of proven techniques in the engineering of viable products. In other words, this is a results-oriented organization with little inclination to indulge speculative experimentation, although it is open to innovations of well-justified potential benefit. Modern software development tools are used. UNIX[2] workstations, X windows and C are standard. C++, Eiffel, graphical user interface and automated testing packages have been used on recent projects. Management is disciplined but dependent on leader expertise and experience. Technical staff work in small task groups under the direction of a technical leader, who is responsible for the quality and timeliness of the work of the group. Employees do not have fixed roles as designer, implementor, maintenance programmer, tester or

[2]UNIX is a registered trademark of AT&T.

the like. Instead, they are assigned to different products and tasks, as projects have need of staff. All work products of the software process are reviewed in writing, normally by the individual technical leader.

In our research, we worked with the technical leader and three other members of a group responsible for maintenance and enhancement of an established product, the Digital Networks Support System (DSS). This product consists of about 100,000 C source statements and has progressively evolved from a much more modest system initially produced in about 1986. It is in active use for managing connectivity of heterogeneous T1 and E1 networks at BC TEL and a number of telephone companies in the United States and New Zealand.

1.3.2 Methods studied

The software engineering methods chosen for study were *module interface specification* and *inspection*. Our definition of module interface specification follows the work of Parnas et al. [PCW84] and Hoffman [Hof90a]: a *module* is a programming work assignment, usually consisting of a group of closely related procedures or functions. A *module interface* is the set of assumptions that users (callers) may make about the module's behaviour and that the module implementer may make about the calling context, that is, circumstances under which the module is used. By *inspection* we mean the disciplined peer review technique invented by Michael Fagan at IBM in the mid-1970s [Fag76], sometimes known as *Fagan inspection* or *formal inspection* to distinguish it from less prescriptive review techniques such as the walkthrough.

These particular methods, selected in collaboration with the SSN department, address prominent activities and concerns in the working environment. Much of the work of the department and in particular of the DSS group involves adding, modifying and maintaining features of existing systems. This typically involves modification of existing software modules or design and implementation of small

numbers of new modules The module is therefore the typical focus of interest "architectural" design activities involving entire systems or subsystems are much less frequent This is a common pattern in industry, where perfective and corrective maintenance of existing systems greatly predominates over development of entirely new systems in the work of most enterprises By addressing module specifications we were able to exercise some new methods of appreciable potential value without affecting the methods used in other phases of SSN's software process This restricted scope was important to the practicability of the project

Our work dealt with module *specifications* rather than *implementations* The SSN technical leadership believed that they had more to gain from innovation in the pre-implementation phases of the software development process A subsequent study of historical defect data for DSS confirmed this intuition that relatively few defect reports could be attributed strictly to implementation errors This is consistent with Brooks' observation

> I believe the hard part of building software to be the specification, design and testing of this conceptual construct, not the labour of representing it and testing the fidelity of the representation [Bro87]

The SSN management was interested in the potential of the inspection technique to alleviate the reviewing workload of their technical leaders while maintaining or improving the consistency and effectiveness of reviews Published reports of industrial experience with code inspection consistently claim that this technique, while labour-intensive, is so effective at fault detection that overall software lifecycle costs are significantly reduced [Fag76, Fow86, ABL89, Rus91, Doo92, KSH92]

The most innovative contribution of our work is in the development of inspection procedures for *interface* specifications This form of module design documentation is itself not widely used in industry, although experience reports from the U S Naval Research Laboratory's Software Cost Reduction program are encouraging [PCW84, Hag89] Published reports concerning inspection frequently propose the

applicability of the technique to all kinds of work products, but describe specifically only code inspection or to a lesser extent inspection of requirements specifications

1.4 Thesis overview

Chapter 2 describes the background to our work previous theoretical and empirical results concerning software process improvement, module interface specification and inspection Chapter 3 describes our approach to module interface specification and Chapter 4 the inspection technique as we applied it to interface specifications Chapter 5 describes the technology transfer process we used with MPR Teltech and its results In Chapter 6 we present our conclusions Examples of work products and training materials are provided as appendices

Chapter 2

Terminology, Concepts and Related Work

2.1 Software process improvement

The idea of *improving* a software engineering process is implicitly based on a view of software and software engineering as things that can be modelled, measured, controlled, and changed. The *software metrics* enterprise developed in the 1970s in recognition of the importance of measurement for making comparative judgements of software and software processes. Although the importance of measurement is a fundamental scientific idea,[1] software metrics have been slow to gain acceptance. Skeptical practitioners are conscious of the difficulties of experimental design and appreciate that ad-hoc measurements cannot support credible conclusions about complex products and processes. Leading metrics researchers have continually stressed the importance of theoretical foundations—models—pointing out that measurement is not meaningful unless based on a coherent intellectual model of the artifact or

[1] Writers on metrics like to appeal to scientific tradition with the following quote:

> When you can measure what you are speaking about, and express it in numbers, you know something about it, but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind. (Lord Kelvin, *Popular Lectures and Addresses*, 1889) [CDS86]

activity being measured [PSS81, CDS86, Hum89, BM91a] Some contributions to software modelling are well-known, for example lifecycles (waterfall and spiral), complexity (Halstead's Software Science), costs (Boehm's COCOMO) and behaviour (finite-state machines), yet much research remains to be done in these and other areas such as reliability and defects Software process modelling is a particular focus of activity at centres of software engineering research such as the Software Engineering Laboratory (SEL) [BCM+92] and the Software Engineering Institute [Hum89]

Because software processes are complex and tasks and resources change frequently, the task of software process improvement must itself be seen as a continuous activity rather than a discrete event Mays et al describe an approach to defect prevention [MJHS90] that has been influential within IBM and that illustrates ongoing commitment of resources to defect prevention as a specific activity Continuous process optimization is the distinguishing characteristic of the most mature software process in Humphrey's SEI maturity framework [Hum89]

Improvement implies change and in the case of the software process the change is most often in technology—be it methods or tools Software technology transfer has proven to be a fragile process, which frequently fails in the absence of careful planning and attention to certain critical factors To summarize, software process improvement must be based on adequate models and metrics and implemented as a planned and supported ongoing process with attention to technology transfer issues

2.1.1 The SEI software process maturity framework

The SEI software process maturity framework was developed in the late 1980s by Watts S Humphrey [Hum89] and others, for assessment of U S Department of Defense software contractors It has since become influential in other software development areas such as telecommunications Software suppliers increasingly encounter customer expectations of SEI "ratings," although this is a distortion of

the intended purpose of SEI self-assessments. Although the SEI taxonomy and assessment scheme are not without critics, the SEI model is presently important in providing a widely-known framework—a common language for describing and comparing software engineering processes.

The SEI framework identifies five levels of software process maturity. They are:

1. *Initial*. Management lacks mechanisms for reliably predicting, monitoring or controlling the software process. Individual working groups operate in an ad-hoc manner, with informal procedures or weak adherence to formal procedures. Under crisis conditions, coding and testing activities predominate.

2. *Repeatable*. The organization is able reliably to make and meet estimates and plans based on past experience. However, changes of personnel, markets or technology readily disrupt this experience-based order.

3. *Defined*. Roles, methods and technologies are coherent and clearly defined, so that the process is maintained and monitored in the face of change and crisis.

4. *Managed*. Extensive process data is collected and analyzed for *product* improvement.

5. *Optimizing*. The *process* is modified on the basis of data analysis, to eliminate problems and improve productivity. Automatic data collection facilitates the ongoing optimization process.

The great majority of organizations studied by SEI were found to operate at level 1. Level 4 and 5 organizations are essentially unknown—hence the appropriateness of these definitions may be questioned. The "factory" paradigm of software development that underlies this model is also contentious.

The SEI framework offers an empirically-based sequence of models of the software process. By identifying the level of an existing software process, we also identify

the most immediately suitable process improvements for the organization This is important because improvements suitable for a higher level process may be infeasible or even deleterious for a lower level process For example, innovations in software tool technology may destabilize a level 2 process which stands in more immediate need of process formalization

2.1.2 Software metrics

A software metric is a parameter of measurement Software producers and consumers are vitally interested in quantitative indices of merit for objective and accurate assessment, prediction and control of software quality and costs Direct measurement of these software attributes is normally infeasible Quality is a complex concept to which a variety of factors contribute (e g , correctness, reliability, producibility, usability, testability, interoperability), many of which are themselves intangible and must be related to a measurable aspect of a specific software model Cost is by contrast intrinsically quantitative and measurable However, our chief interest is to discover what determines cost so it can be predicted and controlled before unforeseen expenditure has occurred, this also must be achieved by modelling cost as a function of some measurable aspects of work products and processes

The general procedure of software metrics is to model some process metric of interest—such as team productivity, or defects remaining after release—as a function of some process or product-related independent variables—such as lines of code produced, type of development environment or team size The modelling function may be analytically or empirically derived, or both An oft-cited textbook by Conte et al [CDS86] points out that the meaning and hence applicability of empirical models is unclear when coefficients and exponents other than small integers are derived from data-fitting Preferred models are simple, intuitively plausible and robustly consistent with the relevant data

Process metrics

Process metrics quantify aspects of the software process and its environment such as size of teams, levels of staff experience (with tools, languages, application areas etc), effort expended (e g , person-hours by activity category), computer resource usage, development techniques and tools Measurements of techniques and tools are typically on a *nominal* or *ordinal* scale, i e , a 0/1 value or category ranking is ascribed

Product metrics

The software product metrics encountered in the literature fall into two major categories *size* and *complexity* metrics, due to a compelling intuition that these product characteristics are likely to significantly determine effort and cost of design, implementation, testing, maintenance, etc

Work on software metrics started with the most prominent work product of the software process program code An obvious and therefore early size measure was lines of code (LOC), of which the current accepted definition is "every line of program text that is not a comment or a blank line" [CDS86] Efforts at circumventing the intuitively obvious shortcomings of LOC[2] have taken a variety of approaches to characterizing the essential size and cognitive complexity of a software artifact These include calculations based on counts of operators and operands,[3] program data (inputs, outputs, internal variables, spans of usage and inter-module information flow), and analysis of logic structure (decision counts and graph-based measures such as McCabe's cyclomatic complexity [McC76] and reachability analyses)

---

[2]LOC is affected by choice of language and coding style, it doesn't distinguish between dense or complicated and simple or repetitious code

[3]Halstead's *Software Science* was an important early contribution to software measurement, although subsequent studies indicated that Halstead's measures probably do not offer any significant advantage over the seemingly cruder LOC metric [CDS86, CG90]

Since our work was concerned with module interface specifications, we were interested in measures for assessing the size, complexity and quality of software specifications, independent of implementation. Such *a priori* measurements have received much less attention than the code-based measures [MK89]. Interface specifications fall into the realm of what is broadly termed software design.[4] The best-known quantifiable characterizations of software design quality derive from principles of *structured design* methodology: module size, cohesion, coupling and structure chart complexity.[5]

Interface specifications derive from the information-hiding rather than the functional decomposition philosophy of software modularization and therefore have no natural relationship with the preceding structure-chart and code-based measures. Instead, straightforwardly measurable characteristics of interface specifications (such as numbers of functions, inputs and outputs) are similar to the metrics of *function point* analysis. This approach was first proposed by Albrecht at IBM in the late 1970s. Albrecht modelled development productivity as a weighted sum of five aspects of user-perceived functionality: inputs, outputs, inquiries, interfaces and logical files. Weightings were derived by measuring a set of programs written by IBM's Data Processing Services group. Albrecht's approach was refined and popularized in the 1980s[6] and now includes a complexity weighting based on subjective evaluation of 14 complexity factors. Feature points, DeMarco's *bang* metric [DeM82] and the

---

[4] See also Section 2.2.1 below.

[5] *Cohesion* or strength refers to intra-module relationships. It may be exhibited in a variety of ways, such as dependence on a common resource or data, functional, temporal or, least strongly, coincidental cohesion. *Coupling* refers to inter-module relationships, which may be based on passing or sharing of data or control information. The traditional hypothesis is that low complexity, modest module size, high cohesion and low coupling are associated with high quality. However, SEL-based studies of the relationship between these criteria and software cost and error rate have shown some unexpected results, supporting the value of cohesion but finding module size and data coupling insignificant contributors [CG90].

[6] The metric is supported by an international user group, IFPUG.

British Mark II function point metrics are related or similar schemes [Jon91] Such composite metrics reflect the common-sense intuition that software size and complexity have many contributing factors, notwithstanding the caution of Conte et al about the meaning and validity of empirically-derived weightings (see page 11)

Defect metrics

Since the failure of software to operate correctly (whether defined with respect to explicit requirements or occasional user expectations) is an unhappily familiar phenomenon, much software lifecycle activity is concerned with ferreting out software defects and establishing confidence of correctness Before discussing the uses and findings of measurement in this area, it is necessary to clarify our terminology The endearing "bug" has recently been supplanted by more appropriately somber terms such as "error," "defect" and "fault " Use of this language is not yet standardized We follow the terminology shown in Table 2 1, adapted from [Hum89, Table 15 3], while noting that many of our references pair the terms and definitions listed in uniquely different ways The distinctions made in the table are meaningful when we compare different approaches to software quality assurance The outputs of testing are *failures*, which must subsequently be traced to *defects* in order to correct the software Verification techniques such as inspection find defects directly

Defect metrics include counts, locations, severity assessments (subjective or according to measures such as number of failed test cases or lines of code affected), cost to repair and source of error (by development phase or activity type) They are used for assessing current software usability, predicting software reliability or future

Term	Definition	Cause
Error	Human action	Human mistake
Defect	Program property	Error
Failure	Program malfunction	Defect

Table 2 1 Software problem terminology

defect removal costs, identifying defect-prone modules and error-prone activities and evaluating the software process. Our interest was in the latter two areas, both for selecting candidate methodologies and for setting in place metrics for assessing their contribution.

Metrics and maturity

If we accept a progressive model of software process maturity, such as the SEI framework, it is apparent that appropriate choices of metrics for software process control and improvement depend on the existing level of process maturity. At the initial process level, measurement is not particularly meaningful, except for beginning to establish a baseline. Effort at this level is best devoted to establishing some repeatable procedures, rather than instituting measurement schemes. At level 2, the repeatable process, some simple metrics such as project effort and cost can help with process control. Finer-grained metrics are not appropriate when activities and products are not standardized. At higher process maturity levels, increasingly refined metrics are mandated to drive process control and improvement. In [Pfl89] S. L. Pfleeger develops this idea and recommends basic measures of software size, personnel effort and requirements volatility at SEI level 1. At level 2, she recommends the addition of product complexity measures and defect counts. Shortly after the start of our project, MPR Teltech Ltd. initiated a related metrics study and trial project that was influenced by these recommendations.

2.1.3 Technology transfer

The acceleration of technological change in Western society during recent decades has prompted extensive study by sociologists and management scientists. Computer scientists have more recently noted that advances in software technology are often

slow to be adopted into widespread use [7] By relating this phenomenon to existing theories of technology transfer and some particular characteristics of the software industry and its technologies, a theory of software technology transfer has been developed and some critical success factors and pitfalls identified This theory provides a useful framework for the software engineering innovator

The diffusion process

In speaking about software engineering technology transfer we use the term *software engineering technology* broadly, following [RC89], to include any concepts, tools, techniques and methods used to create software to meet stated objectives Software metrics, the inspection technique, high-level languages and configuration management systems are all examples of software engineering technology The process by which an invention (i e , a new technology) comes into widespread use is termed *diffusion* Diffusion is driven by communications in two directions inventors try to *push* the technology into application and potential users *pull* new technology into use to address perceived needs As noted by Redwine and Riddle, this process has frequently taken as long as 15 or 20 years, improbable as that may seem in such a rapidly evolving industry as software Object-oriented technology is a timely example the key ideas were present in the Simula language (developed in the late 1960s) but did not become popular until the late 1980s

Roles

Buxton and Malcolm [BM91b] identify the following roles in the diffusion process

- *Supplier* A commercial supplier, academic or industrial research laboratory or department which has invented or adopted the technology The supplier

[7] A collection of case studies by Redwine and Riddle showed that that it takes 15 to 20 years for a software technology to be popularized and widely disseminated [RR85]

typically publicizes the technology and promotes its adoption, supplies it and frequently assists the users with learning to use the technology or adapting it to their needs.

- *Gatekeeper*. A person or group within the user organization having authority for assessing new technologies. Other names for this role are *opinion leader* and *change agent*. If convinced of the value of the innovation, the gatekeeper may exert their influence as a champion of the technology, when skeptical of the new technology the gatekeeper may be equally influential in blocking its adoption.

- *Management*. This is the role in which formal decision-making authority resides. Senior or upper management are concerned with the "big picture" and the long term and frequently lack up-to-date technical expertise. Middle managers, by contrast, are preoccupied with responsibilities for relatively short-term schedules and budgets. These two groups are therefore interested in different types of information about a proposed innovation and may evaluate the same technology differently.

- *Educators*. Education in the new technology is normally important to the diffusion process. Without adequate training workers readily neglect the most promising innovation. This role may be exercised by the supplier or an external or internal training organization.

- *Workers*. The success of an innovation ultimately depends on its adoption by the people who can make use of it in their work.

Phases

The technology diffusion process is seen as comprising a number of phases, whether considered across the industry as a whole or within a single organization. Our work

concerns technology transfer to a single organization. Buxton et al. and Raghavan et al. discern the following phases of adoption within the user organization.

1. *Awareness.* Technical staff, especially "gatekeepers," are often the first to be aware of a new technology, although initial awareness may on occasion arise elsewhere in the organization, for example with managers seeking competitive information or being lobbied by technology suppliers.

2. *Assessment and Decision.* Assessment involves gathering information to support decision-making. Two kinds of information are important: *hard* information describing the details of what the innovation does and how it is used and *soft* information such as costs, analyses of economic benefits, organizational impacts and the like. While hard information is of the greatest interest to technologists, management requires soft information for making adoption decisions. As noted above, upper and middle management are interested in different kinds of soft information.

   The role of the gatekeeper is critical in this phase. With the support of the gatekeeper, a preliminary decision may be made to undertake a *pilot project* as a means of gathering final assessment information. Pilot projects are particularly valued because they exhibit in miniature most of the issues and phases of the entire technology transfer process. Eventually, a management decision is made to adopt or reject the new technology.

3. *Adoption.* It is after the decision to adopt the new technology that the maximum organizational resistance is apt to be encountered. Middle managers who are not adequately committed to the new technology resist changes to schedules, budgets, structures and procedures, workers may feel threatened by change, unconvinced that the innovation is practical or simply unwilling to invest effort in learning new ways, especially if not adequately trained. Ap-

plication of a new technology to real work is rarely as straightforward as it appears in a training course or demonstration. Adaptation or customization of the innovation to local customs and requirements is essential, as is training. The support of the gatekeeper or supplier is critical to the former activity, the same people may also exercise the educator role.

4. *Assimilation.* If the innovation survives the difficult early stages of adoption it acquires increasing credibility and eventually becomes "the new organizational orthodoxy" [BM91b]. In the latter stages of adoption the need for formal training diminishes as people hasten to join the bandwagon through peer assistance.

Critical factors

The following characteristics of the new technology itself have been found critical to its successful adoption [RR85, RC89]:

- *Relative advantage.* The technology fills a well-defined and articulated need by offering a significant advantage over the technology it supersedes.
- *Compatibility.* The technology is compatible with or readily adaptable to fit existing organizational values, practices, past experiences and needs.[8]
- *Accessibility.* The technology is perceived to be reasonably easy to understand, learn and use.
- *Suitability for trial use.* The technology can readily be tried out on small demonstration problems or pilot projects.

[8]This factor may be particularly important to relatively large and well-established organizations, which welcome *competence-enhancing* innovations—those that build on existing skills and methods—but resist *competence-destroying* changes that radically replace existing technology [TA86].

- *Visibility* The results of the innovation are easy to see and measure

Pitfalls

Failed technology transfer experiences illustrate the following pitfalls

- *Inappropriate expectations of users* Much software engineering technology is inherently abstract, having a function that is not completely simple and obvious CASE is an example of abstract technology This type of technology is easily misunderstood and misapplied, with inevitably disappointing results Pressures to keep up-to-date and produce concrete results quickly also lead organizations to "shop for methods and tools without well-defined goals" [Chi88], with high risk of failure at the subsequent adoption phase

- *Inappropriate expectations of innovators* Enthusiasm for the new technology can lead suppliers and opinion leaders to overestimate the maturity both of the organization and the technology We have noted already that introduction of an advanced technology may actually be detrimental to an organization that is using a much less advanced software process Another kind of problem occurs when enthusiasts overlook weaknesses of the innovation Redwine et al claim that conceptual integrity is essential to the success of an innovation, noting that conceptual controversy has held back the diffusion of software metrics [RR85] Even a conceptually solid innovation may require considerable refinement to handle real-world applications Under-estimation of this phenomenon leads to adoption fiascos

- *Social systems* Software engineers are not normally trained in sociology and therefore tend not to take adequate account of the difficulties that are frequently involved in dealing with the social systems of the organization A common example of this is failure to recognize that managers are not

technologists—they are concerned with soft issues and prone to making weak commitments that prove insufficient to sustain the project through early problems.

- *Repeating the "learning curve"* The technology transfer process is not necessarily complete following a successful introduction and widespread adoption. Significant personnel turnover, which is common in the software industry, may lead to a loss of benefits from the technology as the overall level of staff training in it decays. Ongoing attention to education is needed to help combat this problem.

Conclusions

Successful technology transfer is likely to start with a carefully-selected innovation, one that rates well on the critical factors listed above, especially those factors to which the target organization seems particularly sensitive. The support of opinion leaders and agents of change within the organization must be gained. Adequate hard and soft information must be gathered to support decision-making; a pilot project can provide convincing empirical evidence for this purpose. Firm management commitment of resources must be secured and workers must be trained to make full and correct use of the new technology. The introduction process should minimize disruption and complexity, typically by expanding in small steps from a few users and the simplest applications. Users must be supported in adapting the technology to local customs and needs.

2.2 Module interface specification

2.2.1 Modules and interfaces in software design

Module interface specifications fall into the realm of software design documentation In everyday language *design* is a broad concept associated with creative activity For our purposes, E S Taylor's more narrow definition is adequate

> *the process of applying various techniques and principles for the purpose of defining a device, a process or a system in sufficient detail to permit its physical realization* [Pie87, page 213]

The undisputed first principle of software design is *modularization*—dividing the software task into a number of separate pieces, each having fewer responsibilities than the whole Such a "divide and conquer" approach is helpful if not indispensable to coping with the intellectual complexity of large programs It enables delegation of work to multiple participants and inspires the hope of reusing portions of existing programs in new work Although the advantages of modularization as a principle are uncontroversial, the questions of what constitutes *good* modularization and how to achieve it have evoked some diversity of answers over the last 20 years, including information hiding, functional decomposition and object-oriented techniques, and new proposals continue to appear [9]

Module interface specifications originate in the work of D L Parnas, who proposed the *information hiding* modularization principle in the early 1970s [Par72] This principle attempts to minimize the long-term expected software cost by requiring the system designer to identify the system details *most likely to change* in future The modular structure of the system is then devised so that each module hides a potential change as its *secret*—ensuring that a change in the secret would affect only the encapsulating module Potentially independent changes are hidden in separate modules Although Parnas' work was quite well-known, functional abstraction

[9] e g , tool abstraction [GKN92]

was the dominant modularization principle in use in the 1970s and early 1980s It was associated with *structured* analysis, design and programming [You79, PJ80] In the late 1980s object-oriented techniques rose to prominence, offering advantages of *encapsulation* and *inheritance* over the structured techniques Encapsulation is an application of Parnas' information-hiding principle, which illustrates lagging technology transfer

Parnas defines a *module* as a programming work assignment (of an individual or team) consisting of a group of closely related procedures or functions [PCW84] At first glance this definition appears conveniently able to embrace the variety of styles of program decomposition practiced by working programmers, but perhaps rather undiscriminating However when we consider what a practical programming work assignment might be, the definition implies that a module must have a complete, precise and intellectually tractable description This favours modules that "correspond to a single, coherent abstraction" [BL91] In order to protect the module's secret, the information hiding principle requires attention to the module *interface*, defined as the set of assumptions that programmers using the module may make about its behaviour [PCW89] Because the notion of an interface is essentially symmetrical, a second set of assumptions is also operative those that the module implementor may make about the user's behaviour D Hoffman notes that in a robust module the first type of assumptions dominate and implementor assumptions are few [HS92] The module interface specification thus describes the interface so as to support the roles of the original *designer*, the *implementor, verifier* and *user*, or author of programs that call the module

As object-oriented technology has come into increasing use, some classical software engineering problems have inevitably resurfaced in the new paradigm, leading to rediscovery or reconsideration of related work from outside the object-oriented community B Meyer has very recently described *design by contract* as a means of

promoting reliability of object-oriented systems [Mey92a] Meyer's *contract documents* have effectively the same definition as interface specifications[10] and similar content, as we shall see in the following section

Other decomposition strategies such as structured design methodology attack complexity but without particular attention to interfaces Precise interface specification ensures that implementors are able to work independently [PC86] The information-hiding principle promotes software flexibility (ease of change), for reduced maintenance costs [Par79] This flexibility, along with the separation of concerns due to information-hiding, can be expected to make the software easier to *reuse* [PCW89] Hoffman has proposed quality criteria for interfaces, pointing out that attention to simple and consistent interface design can lead to reduced costs because users understand the interface easily and use it with fewer errors [Hof90a] These claims by proponents of the methodology are supported by studies of defect metrics which highlight interfaces as trouble-prone For example, Basili and Perricone found interface defects to be the dominant abstract defect type in new and modified modules of a 90,000-line FORTRAN system for satellite planning studies [BP84] In a more recent Japanese study of the development of four commercial measuring-equipment control packages, Nakajo and Kume also found interface defects predominant (accounting for 57% of 670 recorded faults) and misunderstanding of software interface specifications to be the cause of more than half of these [NK91] Chillarege et al also found interface misunderstanding problems prominent in a study of a large operating systems project at IBM [CKC91]

---

[10] "It protects the client by specifying *how much* should be done It protects the contractor by specifying *how little* is acceptable " [Mey92a]

2.2.2 Interface specification contents

> *Precise interface descriptions are an essential part of any large-scale project because they allow the uses of a module to be designed and checked without examining the implementation of a module. For this reason, interface descriptions must include the intended meaning of each module in addition to the format for invoking each of the services the module provides.* [BL91, page 14]

Our work is based on the interface specifications of Hoffman [Hof89, Hof90b, HS92]. Hoffman's objective is to provide documentation schemes that industrial software developers will find practical, noting that formal specification schemes appear unattractive to working programmers. His documents are therefore succinct, multipurpose and make judicious use of mathematical notation (used only where it facilitates communication). While software design specification is an area in which industrial practice remains divergent, there does exist an ANSI/IEEE standard which provides general guidelines as to organization and content [ANS87]. The style of interface specification here described conforms with the Interface Description provisions of that standard.

An interface specification has two major sections: syntax and semantics. The syntax section defines the exported data types and constants and *access programs*—routines through which the module may be accessed. For each access program, the inputs (parameter types), outputs (parameters or return value) and exceptions raised are detailed. In Hoffman's scheme an exception names a user-provided program that will be called under circumstances specified in the interface semantics.

The interface semantics include a module abstract state definition, implementor's assumptions, and for each access program, specifications of its state transition, outputs and exception conditions (if any). The *state* definition uses basic well-known abstract data types (integer, string, set, sequence, tuple, etc.) to facilitate simple yet precise characterization of the contribution to module behaviour made by the history of previous calls. While not all modules have state, this is a familiar way

of describing modules that implement abstract data types, such as that perennially favourite example, *stack*. The state of a stack module might be described as

s : `sequence of integer`

In the object-oriented world, an object is analogous to a module state plus the operations thereon.

The *implementor's assumptions* describe conditions that the implementor may *assume* hold, they are not checked by the implementation. For example, the semantics of a module that requires some state initialization may provide an implementor's assumption that the initialization routine is called before any other.

The access program transitions, outputs and exceptions precisely describe the behaviour that the user may expect of calls to the module. *Transitions* are the effects on the module state and system entities external to the module such as displays or files. *Outputs* are returned values and data elements. *Exceptions* are the conditions under which named exception routines are called. In Hoffman's model, the module state is unchanged by calls which generate exceptions. This approach has the virtue of simplicity and is implementable in many practical cases.

In Parnas' Software Cost Reduction (SCR) method [PC86], module interface specifications are logically mapped to *implementation specifications*, the repository of deliberately hidden information and other implementation-specific details. By contrast, Meyer's *contract documents* are automatically extracted from function headers and assertions in Eiffel implementations [Mey92a] and thus correspond to an intersection of the interface and implementation specification work products. This has advantages of ease of maintenance and executability, but does not support information-hiding. Documentation of implementor assumptions that are not verifiable in the implementation (the most common type in Hoffman's robust modules) also seems problematic when method *preconditions* are expressed only by executable assertions.

State or class *invariants*—predicates that characterize valid configurations—are used in both approaches. Hoffman employs implementation state invariants to facilitate verification of implementations against interface specifications [Hof90b]. Meyer makes particular use of *class invariants* in his exception-handling scheme, where exceptions do not necessarily leave the module state unchanged, but are guaranteed to maintain or restore the invariant. Invariants have a long history in programming languages and formal methods work [Win90, BL91].

2.3 Inspection

Inspection is a technique for team review of work products. It was invented by M. Fagan in the mid 1970s [Fag76] and has since attracted gradually increasing attention, particularly in the last five years or so. Subsequent practitioners have reported most success with the technique when closely following Fagan's prescription, which had been refined in practice at IBM. The method is therefore well defined. Good brief overviews may be found in [ABL89, Rus91] and detailed descriptions in [Gil88, GTE89, Hum89].

The inspection process is carried out by small teams of three to five or six individuals, consisting of the work product author(s) and some peer colleagues. Managers, clients, service representatives, and the like are normally excluded. The process has quite strict guidelines. Participants are required to *prepare* by individual advance study of the material to be inspected and to fulfill specified *roles* of moderator, reader, inspector, and recorder in the inspection meeting. An author may not serve as moderator or reader. In the meeting, the reader systematically *paraphrases* the work product aloud, allowing all participants to check their understanding of the material and raise objections and concerns. Preparation time, the rate at which the material is inspected, and the required outputs of the process are all prescribed and actual results logged. The purpose of inspection is narrowly defined as *defect de-*

tection and to this end, discussion of such matters as design issues and alternatives and defect solution strategies is not permitted in the inspection meeting. Checklists of potential defects or defect sources appropriate to the work product type are used to promote thoroughness.

Inspection outputs are specified by *forms* on which the recorder logs work products, participants, preparation and meeting times, and defects found. The resulting information is intended not only for correcting the defects in the work product inspected, but may also be analyzed for the purpose of *process improvement*. Humphrey identifies inspection as a required activity in the SEI level 3 *defined* software process [Hum89]. Christenson and Huang describe how statistical analysis of code inspection data guided refinements to the inspection process itself at AT&T [CH87]. Mays et al. describe a successful defect prevention process at IBM, based on analyzing inspection data [MJHS90].

Use of the inspection process can also contribute to staff development. By studying each other's work in detail, participants learn about parts of the work product outside of the scope of their regular responsibilities. This can improve cooperation and also preparedness for rotation of assignments and emergency coverage. Norms, standards, and quality criteria become more concrete through exposure to examples of the work of others. As a result of this exposure, coupled with the desire to exhibit respectable work to peer scrutiny, the quality of individual work tends to improve. The inspection process is a safe, controlled forum in which people learn to function as an effective team. These benefits seem to lead to increased confidence and satisfaction on the part of software developers: experience reports note improvement in staff morale [AFE84, Fow86, Doo92].

2.3.1 Comparison with other techniques

Reviews and walkthroughs

According to [KM91], both Babbage and von Neumann regularly asked colleagues to examine their programs. Inspection is by no means the first or only application of this obvious approach to checking software correctness. Two earlier and well-known techniques of this type are the review and the walkthrough. *Review* is a general term that can be used to cover a variety of techniques, including inspection. In industrial settings, however, a work product review is generally a presentation by the author to a group including managers or clients. The presentation is therefore driven from the author's point of view and defect detection is likely *not* to be an objective.

A *walkthrough* is similar to an inspection in that it typically involves only some peers of the author and is conducted for the purpose of validating a design or finding defects. However, as in the case of a review, the walkthrough is normally conducted by the work product author and other participants are less likely to have prepared by detailed advance study of the material. Walkthroughs are loosely defined and thus the activities covered by this term vary in practice, however a frequent activity is simulated execution of test scenarios: "walking through" the code on the basis of some assumed inputs. This contrasts with the paraphrasing approach of inspections, which is more analogous to program proving than to testing. Scenario testing can often be accomplished more cheaply and reliably by automation.

The less prescribed format of review and walkthrough meetings permits more varied kinds of discussion than in the inspection meeting. This may be unproductive unless carefully controlled. Experience reports suggests that the distinctive characteristics of inspection—individual preparation, controlled use of group time with prescribed focus and outcome, defined roles and a paraphrasing approach to the subject—account for its cost-effective rate of fault detection.

Testing

Both inspection and testing are forms of work product verification Use of inspection should reduce testing costs and in the Cleanroom approach code inspection has even replaced unit testing [Dye92] Inspection should not be expected to further replace testing because the two techniques have such complementary strengths and weaknesses that even though they are both costly, use of both techniques is necessary to assure confidence of software quality

The advantages of testing are obvious it is a well-established technique with assured concrete results and quite amenable to automation Inspection is a human-dependent process Its results are therefore not guaranteed (inspectors can misinterpret program logic or overlook defects) and the prospects of useful automation appear distant and problematic [11] On the other hand, for non-trivial programs testing is invariably incomplete it can assure us of the program behaviour under the scenarios tested, but has nothing to say about the general behaviour of the program Inspection can be general the reader argues the adequacy of the specification or program for all possible inputs The results of inspection are also more satisfactory actual program defects The results of testing are program failures (see Table 2 1), which must be traced to defects before they can be corrected This process is typically expensive and imperfectly reliable

2.3.2 Industrial experience

Inspection was invented in industry (by Fagan at IBM) and the inspection literature is almost entirely of industrial origin, lending the technique solid credentials of practicality

[11] Not only a powerful automated theorem prover but also complete, precise and machine-readable formal specifications would seem to be required

> Fagan developed inspection at IBM, therefore his inspection rules had to contribute to net profit in order to survive If you think an inspection rule is unnecessary, you have probably misunderstood the method [Gil88]

Although it is frequently noted that in principle, inspection can be applied to any "readable" product of a development process, the majority of experience reports concern the inspection of software *code* A recent exception is [KSH92], in which Kelly et al report on statistical findings of three years' experience inspecting software requirements, architectural design, detail design, source code, test plans and test procedures at the California Institute of Technology's Jet Propulsion Laboratory Their results showed significantly higher density of defects at earlier phases of the software life cycle, with relatively constant (and low) cost to fix defects found

In reports from the telecommunications industry [Rus91, Fow86], space flight control [KSH92], seismic research [Doo92] and data processing [Fag86], code inspections are reported as finding about 80% of product defects at the rate of about one defect per person-hour invested (including all inspection-related activities) This is much cheaper than finding defects by testing, by factors ranging from two to four times [Rus91] to twenty times [Fow86] For this reason it is recommended that inspection *precede* any product testing The cost of correcting defects found by inspection is much less than that of detects found by testing (by a factor of 10, in one AT&T study [Fow86]) while the cost of correcting a defect reported in the field (i e , under actual customer use) is larger again 4 5–5 days per defect according to [Rus91, Doo92] Both authors conclude that every hour spent on inspection saved about 30 hours of maintenance effort

Several studies have investigated the relationships between preparation time, meeting length, inspection rate and defects detected [Rus91, KSH92, CH87, AFE84, Fag86] It is universally concluded that preparation is essential, that inspection meetings should not exceed two hours in length and the inspection pace must not be too fast Recommended rates of 100–150 lines of C code per hour are typical

[Rus91, AFE84]

Experience reports stress the importance of attention to technology transfer issues (see Section 2.1.3) for achieving a successful introduction of the technique in the workplace [AFE84, Rus91, Doo92]. The importance of participant *training*, particularly for moderators, is especially stressed, although details are lacking. While examples of code inspection checklists and forms are extant [Fag76, Fow86, ABL89, Hum89], specifics on checklists and techniques for inspecting other work products are not available. Indeed, one group of researchers who attempted a brief survey of industrial experience with the technique found organizations reluctant to divulge specific information on their inspection program, apparently because it was seen as an important competitive advantage. The researchers concluded that

> Inspections are a very useful tool. If more could be discovered and published about how to initiate and use inspections, the whole software industry could benefit. [ABL89]

Our work with inspections attempts to answer this suggestion.

Because the inspection meeting is a group process, videotaped examples of inspection meetings suggest themselves as a powerful training tool. During our research, we viewed three inspection training tapes, produced by SEI [Dei91], Quality Assurance Institute, Inc. and Bell-Northern Research [Rus91] respectively. The BNR material was the most comprehensive, providing a complete introduction to the inspection technique for the software developer with extensive scenes of a meeting proceeding well and badly. In the good example, participants are punctual, well-prepared, objective, keep to their assigned roles and cooperate effectively. In the bad example, the meeting is chaotic, unprofessional and ineffective. No written description could so convincingly prepare trainees to attempt an unfamiliar technique.

The SEI and QAI tapes also provided examples of meetings going well and badly. The SEI material is of good quality but very brief: seven short scenes each under 3

minutes in length. The QAI material integrated a presentation of inspection benefits for managers with training material suitable for practitioners. This dual focus was confusing, while the meeting dramatizations were artificial.

2.3.3 More rigorous inspection techniques

A weakness of Fagan's inspection technique is that it provides no direct means of ensuring that the work product is scrutinized at more than a superficial level. The participation of technical peers and the mandating and monitoring of preparation time are plainly intended to help ensure that the inspection is thorough and penetrating, but this cannot be guaranteed. A veteran of inspections at IBM has noted that "the typical error categories suggest that inspections as commonly practiced do not probe deeply enough." [Bri88]

This problem is not specific to inspection but is rather a hazard of all human review techniques. Parnas and Weiss have reported success in forcing active reviewer engagement with the work product by Active Design Reviews [PW85]. In this technique, reviewers are provided with *questionnaires* rather than checklists. In-depth study of the documentation is necessary for answering questions such as "Which assumptions tell you that this function can be implemented as described?" [PW85, Figure 1]. Active Design Reviews employ a varied group of reviewers, including software specialists, methodologists and user representatives, each with specialized reviewing responsibilities. The product author meets with reviewers individually to obtain their comments, rather than in a group inspection meeting.

Verification-based inspections are an important component of the Cleanroom software engineering methodology developed at IBM's Federal Systems Division [Dye92]. A script of questions used both in preparation and the inspection meeting is generated from the work product pseudo-code by an automatic analysis tool. The questions focus on showing the algebraic correctness of the program design. This

approach is based on Harlan Mills' work on developing and reasoning about correct programs [LMW79] Dyer's book [Dye92] provides a comprehensive description of all aspects of Cleanroom An earlier paper by Britcher, also of IBM, presents the essential idea of verification-based inspection and also recommends attention to preconditions and invariants [Bri88]

Hoffman has proposed a verification-oriented approach to the inspection meeting, in which the reader takes responsibility not simply for paraphrasing the work product, but for arguing its correctness with respect to its specification [Hof90b] Hoffman argues that controversy over "formality" in proofs of program correctness has needlessly discouraged the use of verification-oriented approaches in industry He cites the controversial paper of De Millo et al [DLP79] in support of the idea that an adequate proof is any argument that convinces a representative group of peers Interestingly, the same paper is also cited in an early report on the implementation of the inspection process at AT&T [AFE84] There it is used to support the more modest claim the inspection is not in principle incompatible with proofs of program correctness At that time, the idea seems not to have been extended to correctness-oriented inspection techniques

Chapter 3

Module interface specification

3.1 Designing the work product

Our mandate from MPR Teltech (see Appendix A) was to study existing module design documentation practices and propose a new standard. We found that existing documents combined module interface and implementation design information. This kind of presentation seems natural to module designers and is frequently seen in association with traditional waterfall models of the software development process [Boe76]. The all-in-one module design document is a straightforward formulation of the deliverable of the detailed design phase which precedes implementation.

It is increasingly recognized that this view of the software development process underestimates the complexity and costs of ongoing product maintenance and evolution, which ultimately dominate the lifecycle of a long-lived software product such as DSS [SP90, section 2.5]. DSS module design documentation tended to be well oriented to the initial design and implementation tasks, but module users and maintainers found them only modestly useful. Consequently, despite good intentions, they were frequently not maintained. Our intention was not to disregard the important needs of implementors, but to achieve a better balance with user needs. The separation of interface and implementation specifications was important to this aim, as well as having the previously-mentioned benefits to design quality resulting

from separation of concerns

Having selected module interface specifications as the recommended type of design documentation, we proceeded to formulate a document standard that would be suitable for use in the SSN department of MPR Teltech As described in Section 2 1 3, adaptation to local needs and practices is vital to successful technology transfer There were two primary objectives to take into account, plus a variety of lesser considerations

3.1.1 Specification completeness

To identify local needs—weaknesses in the software process—that the new work product might usefully address, we conducted a study of recent defect reports against the DSS software product (see Appendix B) This study identified design *incompleteness* as a leading source of defects Programs typically performed correctly when called under circumstances anticipated by the designers However, testers and users were able to create scenarios not anticipated by the designers, under which the software would not behave as desired Since unstated assumptions are at the root of such design defects, we were led to emphasize implementor's assumptions in a document organization that facilitated analysis of the problem space partitioning (see Section 3 2 3 below)

3.1.2 Inspectability and usability

To facilitate inspection of work products, they should be systematically organized, readable, succint and precise These same attributes are equally important to specification usability and maintainability Module users are unlikely to read and maintainers unlikely to maintain rambling, repetitious or vague descriptions

Mathematical constructs and "formal" notations such as sets and predicate logic are supremely precise They are also highly succinct, so long as the power of the

formalism chosen is comparable with the complexity of the task We therefore promoted the use of simple formalisms—sets and sequences, propositional and predicate logic—wherever authors and readers found their meaning clear Comprehensive formal methods have been slow to infiltrate the software industry, we believe that industrial software engineers often find formal notations more intimidating than enlightening For example, the statement

sequence x is sorted, strictly increasing

may be preferred to

$$(\forall i, j)(0 \leq i < j < n \rightarrow x[i] < x[j])$$

Similarly, pseudo-code is sometimes more accessible than unadulterated logic for specifying program functions Our criterion was effective communication—a central purpose of the specification effort in this project We modelled judicious use of formality in training materials but pressed more strongly for consistency than for use of any one particular style in participants' work Following Parnas, we also used tables to promote succinctness and readability of specifications

3.1.3 Compatibility

Some basic compatibility considerations were to link smoothly to existing work products, to ensure that all functions of the replaced work product were accounted for, and to use local terminology For example, the document itself was entitled a Module Access Specification, because the term Interface Specification was already in well-established use to describe a different work product having to do with interfaces to physical devices A Module Implementation Specification document standard was drafted and examples provided, even though subsequent project activities did not involve this type of work product The material provided served to show how

implementation design information could be handled under the new scheme, and in a format very similar to the familiar previous document.

Some mundane considerations governed the choice of document preparation system. It was important that the documents be readily accessible on-screen, because MPR Teltech technical staff are well-supplied with workstations and this is generally the preferred mode of access to information. If the documents were not readily available for viewing, examination of source code would likely continue to be the preferred means of obtaining information. We wanted to discourage this practice in order to facilitate information-hiding and reduce problems due to interface misunderstanding. Since the purpose of the project was not to focus on document-formatting, it was appropriate to rely on document preparation software that was already available in the department (Framemaker and troff). If the documents were to be maintained, it was also best that they be compatible with the local RCS version-control system. RCS manages only ASCII files. For these reasons we elected to prepare the documents in UNIX *man* page format (*nroff* and *troff*).

3.2 Work product example: module spath

Figures 3.1–3.5 contain portions of a Module Interface Specification for a demonstration module named *spath*. The following sections describe the work product contents with reference to this example. A work product standard, the complete spath specification, and a further example are provided in Appendices C, D and H respectively.

The name *spath* is an acronym for *shortest path*. A summary of the module semantics may be found in the DESCRIPTION section at the bottom of Figure 3.1.

NAME

spath, sp_initialize, sp_add_node, sp_delete_node, sp_add_edge, sp_delete_edge, sp_query_path, sp_get_path_length, sp_get_first_edge — shortest-path directed graph module

SYNOPSIS

```
#include "spath.h"

void    sp_initialize(),

bool    sp_add_node(
    int     nod),   /*i*/

bool    sp_delete_node(
    int     nod),   /*i*/

bool    sp_add_edge(
    int     src,    /*i*/
    int     dst,    /*i*/
    int     len),   /*i*/

bool    sp_delete_edge(
    int     src,    /*i*/
    int     dst),   /*i*/

bool    sp_query_path(
    int     src,    /*i*/
    int     dst),   /*i*/

int     sp_get_path_length(
    int     src,    /*i*/
    int     dst),   /*i*/

int     sp_get_first_edge(
    int     src,    /*i*/
    int     dst,    /*i*/
    int *   nod),   /*o*/
```

DESCRIPTION

Module **spath** provides access to a weighted, directed graph with integer-valued node identifiers and edge lengths. Spath allows the user to add and delete nodes and edges in the graph and to examine its paths.

Figure 3.1 SPATH Module Interface Specification—Name, Synopsis, Description

STATE

set of integer *nodes*

set of record {
integer *src, dst, len*
} *edges*

Invariant Properties

for each edge *e* in *edges*
e src and *e dst* are in *nodes*

Figure 3.2 SPATH State

Conditions and *Invariant Properties* Both specify conditions or predicates that restrict the class of valid states. Initial Conditions specifies properties of the initial state (before any calls to the module).

The spath example exhibits an Invariant Property that every node referenced in the edge set must occur in the node set. By attaching this kind of property to the state specification, we restrict the domain of states with which access routines must be concerned. Such restriction permits simpler access routine semantics, since the excluded states need not be addressed. Invariants also guide inspection of access routine semantics, as maintaining the module state invariant is a minimal correctness requirement for all access routines.

The spath state invariant illustrates our principle of flexibility in notation. Software developers expressed a preference for this programming-language like approach over traditional predicate logic notation.

3.2.3 Semantics

The SEMANTICS section (Figure 3.3) contains a separate subsection for each access routine listed in the SYNOPSIS. Each begins with an *overview*, or short English description of the access routine semantics. This feature was suggested by the MPR Teltech participants. It is redundant for a module as simple as spath, but we found

SEMANTICS

sp_initialize()

overview

Initialize the graph to have no nodes or edges

effects *nodes* = *edges* = {}

sp_add_node(*nod*)

overview

Add a node to the graph

assumptions

sp_initialize has been called

exceptions

range *nod* not in the range 0 **SP_MAXNODES**-1
An alarm message is logged

effects *nodes* = *nodes* + {*nod*}

outputs

entry condition	return value
nod in range	TRUE
range exception	FALSE

sp_delete_node(*nod*)

overview

Delete a node from the graph

assumptions

sp_initialize has been called

effects *nodes* = *nodes* - {*nod*}
FOR each edge *e*
IF *e src* == *nod* OR *e dst* == *nod*
THEN *edges* = *edges* - {*e*}

outputs

entry condition	return value
nod in *nodes*	TRUE
nod not in *nodes*	FALSE

Figure 3.3 SPATH Semantics—node operations

it helpful in modules with more complicated semantics

The first access routine, *sp_initialize*, has simple semantics which are fully specified as *effects* on the module abstract state

The semantics of *sp_add_node* illustrate our partitioning of access routine semantics *Assumptions* are implementor's assumptions—conditions that the caller is required to ensure They are not checked in the implementation Access routine behaviour may be arbitrary if assumptions are violated For example, invariant properties may be violated (compromising all subsequent calls) or the program may abort The presence of assumptions therefore indicates that the module is not perfectly robust Although robustness is a strongly desired attribute of software, a designer must include assumptions in a specification when it is infeasible to check some necessary preconditions for correct functioning of a routine in the routine itself Such a check may be infeasible due to lack of facilities for detecting the problem (for example, an unassigned memory address), or impractically expensive in relation to the benefit gained Cost considerations (whether of implementation effort or run-time resource consumption) assume significance when the precondition can easily be ensured by the caller The assumption specified for sp_add_node—that the initialization routine has been called at some time previous—might plausibly be accounted under the latter category, if not the former

Exceptions are abnormal conditions that are detected and signalled by the routine These include inherently illegal requests (as in the sp_add_node example) and detectable system problems, for example, a dynamic memory allocation failure Exception semantics are specified following the exception condition(s) and are expected to comprise solely notification of the occurrence of the exception condition In the DSS environment, this is normally achieved by logging an alarm message to the system event log, as illustrated in the example Where more extensive semantics are unavoidable, maintenance of module state invariant properties is mandated For the

majority of DSS modules, the safe and simple null-effect approach proved adequate, while the work product standard is open to more elaborate exception semantics per [Cri91, Mey92a] when necessary

The *range* exception specified for sp_add_node allows the implementor to restrict the size of digraph supported The constant SP_MAXNODES referenced must be defined in the SYNOPSIS Since it is not specified there directly, the reader would expect to find it in the included file "spath h"

Effects are the normal-case semantics of the routine They include module state transitions and external effects such as input/output This portion of the specification assumes that the assumptions are satisfied and that no exception is detected For sp_add_node, the effects are therefore very simple the given node is added to the module state

Outputs are the values returned by the routine, as return values or output parameter values We make frequent use of tables for listing output values

The semantics of *sp_delete_node* have no exceptions Deletion of a non-existent node is treated as an acceptable call, with a special return value indicating that the module state is unchanged Depending on the software requirements, this design choice could be made otherwise We use exceptions to report inherently illegal calls and system problems Effects and outputs may also provide for the detection and reporting of special cases that are not deemed exceptions This is particularly apparent in modules that process information received from the user interface Rejection of invalid user input may be a normal effect in such modules, rather than an exception

The sp_delete_node routine has more elaborate effects than sp_add_node, due to the necessity of updating the edges set to maintain the state invariant On deletion of a node from the nodes set, all edges connected with that node are also removed

The semantics of the remaining access routines are shown in Appendix D

Semantics partitioning

An access routine semantics specification is a function from module state × input parameters to module state × outputs [1]

$$F : S \times I \longrightarrow S \times O$$

Our specification format is oriented toward ensuring that individual access routine specifications are *complete*, that is, that the specification deals with the entire domain $S \times I$. The domain is partitioned into three cases excluded by assumptions (A), cases specified as exceptions (E), and normal cases (N) handled in the effects/outputs specifications

$$S \times I = \neg A \cup E \cup N$$

We term this partitioning the *specification trichotomy*. By completeness, we do not mean that the access routine behaviour, F, is necessarily defined over the entire domain. Indeed, it is normally defined over $E \cup N$ only. What is important is that the excluded portion, $\neg A$, be identified explicitly, and F be completely defined everywhere else. As illustrated by Figure 3.4, this simple formula guides a systematic approach to ensuring completeness during specification and inspection.

During specification, normal and abnormal cases are first distinguished—the first dichotomy, shown at the top of the figure. The normal case is likely to be foremost in mind. Once the specifier identifies what situations are *not* covered by the intended normal-case processing, the abnormal cases are further partitioned into assumptions and exceptions—the second dichotomy. Our trichotomy has a very evident asymmetry in that we use a negation, $\neg A$, to represent the abnormal cases for which the access routine behaviour is not specified, while the two remaining portions of the

[1] Both the domain and the range may be augmented by external conditions mediated by input/output devices. This extension does not affect the partitioning argument, so for simplicity it is ignored herein.

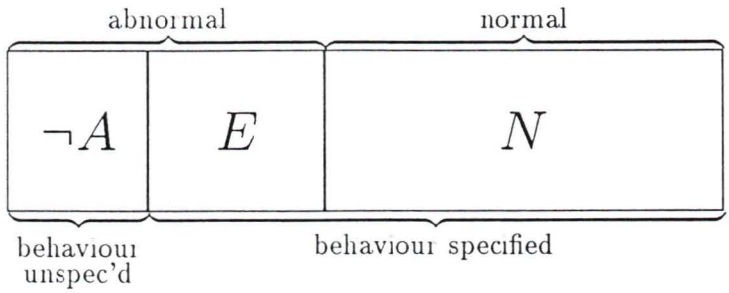

Figure 3.4 The specification trichotomy—two dichotomies in one trichotomy

domain are more simply labelled E and N. This asymmetry occurs because the acceptable case is normally framed as a positive expression. It therefore seems more natural in practice to state positive *assumptions* (A), that are preconditions of the specified behaviour, rather than negative *exclusions* of cases to which the specified behaviour does not pertain. For example, in the specification for sp_add_node, we state the assumption that sp_initialize has been called, rather than the exclusion of cases in which sp_initialize has *not* been called. Exceptions, by contrast, are preconditions of distinctive specified behaviour, so it is natural to frame them as positive conditions with consequences, e.g., "when the value exceeds the limit, raise an alarm," rather than matching the treatment of assumptions, which would lead to expressions such as, "unless the value does not exceed the limit, raise an alarm."

When the specification is inspected, the normal cases are identified by elimination as

$$N = ((S \times I) \cap A) - E$$

The effects and outputs are then inspected to verify adequate handling of all such cases.

3.2.4 Usage

The USAGE section (Figure 3.5) provides illustrative module usage scenarios, to help verifiers and users of the specification work efficiently. The first purpose of

USAGE

1 Graph with 5 nodes and 4 edges is created, a path is followed, intermediate node is deleted, a longer path is found

call	exception	*nodes*	*edges*	outputs
sp_initialize()		{}	{}	
sp_add_node(0)		{0}	{}	TRUE
sp_add_node(1 4)		{0,1,2,3,4}	{}	TRUE
sp_add_edge(0,1,1)		{0,1,2,3,4}	{<0,1,1>}	TRUE
sp_add_edge(1,4,1)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>}	TRUE
sp_add_edge(1,4,5)	dup edge	{0,1,2,3,4}	{<0,1,1>,<1,4,1>}	FALSE
sp_add_edge(0,2,2)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>, <0,2,2>}	TRUE
sp_add_edge(2,4,1)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>, <0,2,2>,<2,4,1>}	TRUE
sp_query_path(0,4)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>, <0,2,2>,<2,4,1>}	TRUE
sp_get_path_length(0,4)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>, <0,2,2>,<2,4,1>}	2
sp_get_first_edge(0,4)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>, <0,2,2>,<2,4,1>}	*nod* = 1, 1
sp_get_first_edge(1,4)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>, <0,2,2>,<2,4,1>}	*nod* = 4, 1
sp_delete_node(1)		{0,2,3,4}	{<0,2,2>,<2,4,1>}	TRUE
sp_get_path_length(0,4)		{0,2,3,4}	{<0,2,2>,<2,4,1>}	3
sp_get_first_edge(0,4)		{0,2,3,4}	{<0,2,2>,<2,4,1>}	*nod* = 2, 2

2 Print all nodes on the shortest path from x to y

```
print x
z = x
DO n = sp_first_edge(z,y,&z), print z
UNTIL z == y
```

DIAGNOSTICS

void **sp_dump**(),

Prints a dump of *nodes* and *edges*

Figure 3.5 SPATH Usage and Diagnostics

the usage scenarios is to demonstrate the sufficiency of the interface for meeting the module requirements. This demonstration does not depend on the existence of a comparable Requirements Specification document. While precise, complete, and current requirements documentation is frequently not available in industrial settings, software requirements are typically implicit in the knowledge of expert individual(s). In such cases, the experts should verify this section against their expectations. The same scenarios provide the module user with model call sequences they can copy. This section furnishes some of the kind of material that reviewers might be asked to generate in an Active Design Review (see Section 2.3.3, [PW85]). By having the author work out some scenarios, immediate deficiencies may be corrected before involving multiple reviewers. The model scenarios speed readers' understanding of the material and may be the basis for more specialized review questions and tests.

The spath USAGE section illustrates both of our recommended formats for usage scenarios: execution tables and pseudo-code. The execution table (scenario 1) shows a sequence of module access calls, with the resultant exceptions signalled, state effects and outputs.

3.2.5 Other information

The work product standard (Appendix C) provides three further sections following USAGE. They are APPLICATIONS, SEE ALSO and DIAGNOSTICS.

APPLICATIONS provides a table of dependencies between top-level system features and access routines called. This material is used as a reference for System Test planning. See Appendix H for an example.

SEE ALSO and DIAGNOSTICS are standard *man* page sections. Figure 3.5 includes DIAGNOSTICS information.

3.3 Quality criteria

Many software quality factors are applicable to module interface designs. Card lists eleven widely-accepted general categories (correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability, and interoperability) originating in the work of McCall et al. at the Rome Air Development Center in the 1970s [CG90]. Numerous specific criteria may be adduced for each such category, leading to hundreds of potentially relevant considerations. Rather than overwhelming designers with an unmanageable number of guidelines, we recommended attention to a small number of fundamental criteria. Our selection was guided by the defect study and general principles, and yielded five items.

- *Completeness.* The behaviour of each access routine for *all* combinations of states and inputs must be clearly specified. This criterion emerged most strongly from the defect study.

- *Comprehensibility.* The specification should be easily understood by the intended audience. This criterion is not the same as readability by a general audience. The intended audience consists of technical peers—inspectors, implementors, users and maintainers—who are familiar with relevant supporting documentation (see Appendix C).

- *Consistency.* The interface must be consistent with local standards, conventions and characteristics of related modules. This applies to matters ranging from obeying naming conventions and using standard return codes to following local idioms of functional decomposition.[2]

[2]For example, if several modules exist to provide communications to the same number of different devices, and each module has access routines called *_initiate_connection*, *_send_message*, *_acknowledge*, etc., the designer of a new module to provide communications to a new device should be extremely reluctant to innovate.

- *Sufficiency* The interface must be powerful enough to satisfy all the relevant software Requirements This is the minimal criterion for satisfying Requirements

- *Simplicity* The interface should be as simple as it can be without violating the other criteria This is a basic principle of what people normally recognize as "good" design it does everything necessary and nothing superfluous To promote undistracted attention to the preceding fundamental criteria, we treat simplicity as a heuristic criterion that is not further broken down or quantified [3] Instead, we call upon experienced designers and inspectors to keep it in mind and "know it when they see it"

[3]Hoffman proposes essentiality, generality and minimality as more precise interface quality criteria which participate in our broad heuristic of simplicity [Hof90a]

Chapter 4

Inspection of interface specifications

4.1 Introduction

The basic elements of the inspection process—roles of participants, preparation guided by checklist, controlled focus of meeting, data collection using forms—are quite well standardized in the literature. Our process was modelled on the common practices described by [AFE84, Fow86, GTE89, Hum89, Rus91], with adaptation to suit our industrial setting and work product. Since published inspection materials and techniques address work products quite different from interface specifications (i.e., code, pseudo-code, and requirements specifications), it was necessary to invent our own checklists, metrics and paraphrasing techniques.

4.2 Roles

Our inspection teams consisted of three to five individuals, taking the roles of Author, Moderator, Reader, Recorder, Inspector and Observer. All participants except Observers fulfill Inspector responsibilities. An Author may serve as Recorder. The minimal inspection team thus has three participants: Moderator, Reader and Recorder/Author.

4.2.1 Author

The participation of an author of the work product is a standard feature of the inspection process. The author may not serve as Reader because the author's familiarity with the work product is likely to lead them to read too quickly for the other inspection participants, and to interpolate over omitted or unclear descriptions. The author may not serve as Moderator because their identification with their work is likely to be incompatible with the moderator's task of promoting impartiality. Nonetheless, the author's participation is critical. The author is best able to answer questions and clarify confusing issues, so that the other inspectors do not needlessly waste time [1]. Also, when freed of the duty of presenting their work (since that task is handled by the Reader), an author's expertise can make them the most insightful inspector of all. Since it is the author's responsibility to subsequently resolve issues raised in the inspection meeting, an author may also appropriately take on the role of inspection Recorder.

The author's responsibilities are itemized in Figure 4.1. The Overview mentioned in item 4 is a brief oral presentation (perhaps five to ten minutes long) that sets the module in context for the other inspectors. The moderator may request such a presentation for a module that is especially complicated or in some way unusual.

4.2.2 Moderator

The moderator is responsible for organizing and conducting the inspection meeting in a professional and productive manner, and for ensuring that follow-up work is completed satisfactorily. The inspection literature stresses the importance of the moderator's role and the need for specialized moderator training, with special attention to the psychology of groups dealing with contentious issues [Fag86, Hum89, You89]. In our project at MPR Teltech, inspection meetings were lively but not ac-

[1] This does not affect the recording of defects in clarity.

1. Ensure that the work is ready to be inspected
2. Support the Moderator in planning the inspection meeting(s)
3. Make the inspection package available on time to all participants
4. Provide a product Overview if required
5. Maintain objectivity and avoid defensiveness
6. Exercise patience at covering familiar ground
7. Explain clearly whenever requested
8. Resolve all identified issues, checking with relevant inspector(s) whenever something is not clear from the inspection meeting records

Figure 4.1 Author responsibilities and guidelines

rimonious, even though we did not offer special training or emphasis on the moderator's role in controlling disputes. (This result is further discussed in Section 5.4.2).

The moderator's responsibilities, itemized in Figure 4.2, emphasize accountability for ensuring that the inspection process is executed thoroughly and correctly, before, during and after the inspection meeting.

4.2.3 Reader

The task of the reader in the inspection meeting is to verbally present the specification so as to convincingly demonstrate its correctness, or to ensure that any deficiencies come to light. Thus the reader paraphrases the material in such a way as to express their understanding of it, as a representative well-informed peer of the author. This provides all the other participants, including the author, a normative interpretation to compare with their own understanding of the material and check for questions and concerns. A measured verbal presentation assists participants to be mentally engaged with the material at a pace that encourages thorough evaluation. This moderate pace is key to the detection during the meeting of defects that

1. Verify that the work is ready to be inspected
2. Determine whether an Overview is required
3. Select and notify inspection participants
4. Schedule and book inspection meeting(s)
5. Ensure that the inspection starts and ends on time, with all participants present and prepared (If not, reschedule the inspection)
6. Conduct the inspection meeting so that
   (a) The pace of the meeting is appropriate for cost-effective defect detection (Too slow is expensive and leads to "tuning out" Too fast leads to superficiality — major defects are overlooked)
   (b) All participants contribute
   (c) The focus is on finding defects
   (d) Discussion is on-topic and objective
   (e) All required inspection data and detected problems are recorded
7. Review the Inspection Record and Summary (prepared by the Recorder)
8. Ensure that all problems found are followed up on schedule
9. Review the follow-up work or initiate the re-inspection meeting

Figure 4.2 Moderator inspection responsibilities and guidelines

1 Paraphrase the work out loud, that is, restate the meaning of each portion in your own words

2 Read slowly enough that other inspectors can follow the paraphrasing and mentally compare it with their own understanding

3 Maintain a consistent reading pace, spending equal time on sections of equivalant difficulty (Simple and uncontentious material should be paraphrased more briefly than complex sections) Avoid

   (a) *verbatim reading*—simply repeating the printed material verbally

   (b) *skimming* or *browsing*—superficial summarization which discourages defect detection

4 For each access routine, make verbal arguments to satisfy the inspection team that the stated semantics are appropriate In particular, argue

   (a) *completeness* that the routine correctly handles all possible calling contexts (module states, input parameters, environmental conditions)

   (b) *invariants* that the routine preserves invariant properties of the module state, if any

Figure 4 3 Reader inspection responsibilities and guidelines

escape notice during individual inspection preparation [2]

The reader's responsibilities are itemized in Figure 4 3 We promote active engagement with the specification by requiring the reader to make verbal arguments (informal proofs) rather than simply reciting the printed material This paraphrasing technique is further described in Section 4 4 below The list of reader responsibilities does not explicitly mention *sufficiency* arguments because the DSS software Requirements Specifications were not sufficiently detailed to support this kind of argument Hence we focus on completeness, which is a self-contained property of the Module Interface Specification alone The inspection process is also helpful for promoting sufficiency in the absence of detailed formal requirements, because it sub-

[2]Studies at IBM, AT&T and Bell-Northern Research have shown that the most defects are found in systems code when the inspection rate is about 150 source lines per hour [Fag76, AFE84, Rus91]

jects the work product to the scrutiny of multiple individuals, who may be chosen to bring different areas of expertise to their evaluation of the specification

4.2.4 Inspector

All inspectors are responsible for preparing conscientiously and participating actively in the inspection meeting Our guidelines for inspectors, itemized in Figure 4 4, emphasize courteous objectivity, and focusing on substantive rather than trivial concerns Extended discussion in the inspection meeting of minute or subjective criticisms is not only unproductive but apt to be perceived by the author as harassment When other inspectors avoid this temptation, the author is more likely to welcome their assistance at finding substantive defects Solution-seeking discussion is discouraged for similar reasons First, it infringes on the author's responsibilities Even when the author does not find such infringement insulting, it risks weakening the author's sense of motivation and responsibility for achieving the best resolution of the issues raised Furthermore, spontaneous problem-solving is typically a poor use of group time, and yields solutions that have flaws because they were not carefully thought out While the interaction of a group of experts can be effective for problem solving, we recommend that such discussions be deferred until after the close of the inspection meeting proper

4.2.5 Recorder

The recorder is responsible for the written records of the inspection meeting The inspection process mandates the use of standard forms for data collection Our forms are described in Section 4 5 below The recorder's responsibilities are itemized in Figure 4 5

1 Be prepared for the inspection

2 Mentally analyze the reader's paraphrase, asking

   (a) Is that the same as my understanding of the work product?

   (b) Are there any problems with the work product?

3 Maintain objectivity and politeness

4 Evaluate the work product, not the author

5 Concentrate on detecting problems

6 Acknowledge good work
(Objective feedback includes the positives as well as the negatives)

7 Avoid criticism of style

8 Insist on more explanation whenever you don't understand something

9 When you turn out to be wrong, forget it

10 Avoid discussion of trivial problems
(Just make sure they're recorded)

11 Avoid problem-solving
(If you have a good suggestion for the author, offer it after the meeting)

12 Support the Moderator

Figure 4 4 Inspector responsibilities and guidelines

1 Complete all relevant sections of the forms

2 Record every issue raised, in clear and objective language

3 Ensure that the person(s) raising the issue agree with your description
(Unless the issue is straightforward and uncontentious, state verbally what you're writing)

4 Classify the problems according to the project classification scheme

5 Prepare and distribute the Inspection Record and Inspection Summary promptly

Figure 4 5 Recorder inspection responsibilities and guidelines

4.3 Preparation

Individual advance preparation by inspectors is essential to the inspection process. If inspectors are not adequately prepared, the meeting itself is apt to find superficial defects at best—an inadequate return on the staff time invested. To encourage adequate preparation and to collect data for formulating and refining preparation time guidelines, individual inspector preparation times are logged by the recorder at the beginning of the inspection meeting.

The preparation activity is much like an individual work product review or desk-check, but with more explicitly prescribed criteria and less prescribed outputs. Inspection preparation has a clear defect-finding orientation and is guided by a *checklist* which is customized for the work product and according to local experience. This guidance is particularly helpful to the software developer who is inexperienced at reviewing the work of others. The output of inspection preparation is simply notes of questions and concerns, for the inspector's own use during the inspection meeting. This is quicker and easier to prepare than review notes that the author must use.

The inspection checklist shown in Figures 4.6–4.7 has two major divisions: *document* criteria and *interface* criteria. Document criteria are concerned with the quality of the specification as a document: that it complies with the work product standard and is readable. Interface criteria are concerned with the quality of the interface described by the document. The detailed interface criteria in the checklist are derived from the general quality criteria of Section 3.3.

4.4 Paraphrasing and proofs

As discussed in Section 2.3.1, inspection and testing are complementary techniques for assuring product quality. Inspection is labour-intensive, but has a potential

1. All required document sections are present
2. The order and format of information complies with the document standard
3. Every data element (type and constant) referenced is defined in a Data Definition File
4. Description is a brief and accurate overview suitable for a technical reader who has some familiarity with the overall system but not with this module
5. State description is clear and as simple as possible
6. State description contains all information needed to make effects of access routines precise
7. State description is not unnecessarily implementation-specific
8. All assumptions about the state are documented as initial conditions or invariant properties
9. Access routine semantics (overviews, assumptions, effects and outputs) are clear and precise
10. The meaning and use of each parameter is clear
11. The usage scenario for every service the module is required to provide is either obvious or documented

Figure 4.6 Module Access Specification inspection checklist—document criteria

12 Access routine names, returned types and behaviour are consistent with other routines and modules

13 Parameter names, types and ordering are consistent with other routines and modules

14 Assumptions are reasonable and consistent

15 Exceptions and their effects are reasonable and consistent

16 Access routine semantics are complete, i e , every combination of parameters and state not excluded by the assumptions or identified as an exception is acceptable as a "normal case "

17 Effects are precise for all normal cases

18 All outputs (return value and output parameters) are specified for each case

19 Effects and outputs satisfy the relevant Requirements

20 Parameters and return values are adequate to express all the required service variations

21 No access routine duplicates the services of other access routines

22 No unnecessary services are provided

23 Independent services are provided by independent access routines

24 The most likely future enhancements, extensions and implementation changes to this module could be accommodated without changing the interface syntax

Figure 4 7 Module Access Specification inspection checklist—interface criteria

advantage of generality over testing, the results of which are inevitably scenario-specific In order to maximize the benefits of the inspection meeting, we believe that the reader's paraphrasing technique should intentionally aim at verification of desired properties of the work product Since an inspection meeting is an assembly of a competent peer group, it is an appropriate venue for *proofs*, as defined by Mills a proof is an argument that satisfies people with competence in the topic [LMW79]

The principal type of proof we require of readers is the completeness argument It must be shown that the effects are well defined over all normal cases

$$N = ((S \times I) \cap A) - E$$

(see also Section 3 2 3), and that the specified exception semantics and effects are at least feasible The reader normally also argues that the effects are sufficient to meet the requirements When the requirements are not completely documented, this provides a check on the shared understanding of what is required, as well as on the module interface design

4.4.1 Paraphasing example: spath semantics

Here is a concrete example of how a reader might paraphrase the semantics of *sp_add_node* and *sp_delete_node* shown in Figure 3 3 This is a transcription of an unrehearsed verbal paraphrase, with colloquialisms and weaknesses of exposition retained to provide a realistic example of inspection-meeting argumentation The reader starts with a straightforward paraphrase of the semantics specification

> Procedure sp_add_node adds a node to the graph The node to add is specified by the parameter
>
> It has one assumption—that sp_initialize has been called—so the graph has been initialized at some point
>
> It has one exception, called *range*, which is signalled when the node number supplied is outside the range zero to SP_MAXNODES minus

one So we have a limit on the maximum number of nodes in the graph If the exception occurs, an alarm message is logged, as per usual

The effect of the procedure, so long as the node is in range, is that the node is added to the nodes set

The outputs are just the boolean return value It's true if the node was in range and added to the set, and false if the range exception occurred

Other inspectors might interject questions and comments during this monologue, which is moderately paced because the reader is not the author and therefore does not know all the details by heart Because the reader needs time to think, there is a tendency to make summarizing comments such as "so we have a limit on the maximum number of nodes in the graph " This helps the other inspectors keep up with the reading, with time to formulate questions and comments The reader continues with the completeness argument

Let's consider the completeness argument for this procedure Since we assume that sp_initialize has been called, we know that the invariant property holds on entry to the procedure, and that characterizes the module state The node parameter value can be any integer However, the range exception eliminates all values outside the legal range zero to SP_MAXNODES minus one

If the node is in range, there are two possibilities either it's already in the graph, or it's not If it's not already in the graph, the case we probably normally expect, it gets added to the node set That seems appropriate If it is already in the graph, adding it won't change the nodes set, because sets can't contain duplicates, so the add operation has a null effect

I think these semantics are complete no matter what the module state was to start out with It appears that attempting to add a node that's already in the graph has no effect, and the user isn't notified in any way that the node was already there Any comments on that before we consider the invariant argument?

Finally, the argument that the module state invariant property is preserved by the semantics

OK, we know the invariant holds on entry to the procedure and we have to assure it holds on exit There's two possibilities as we've seen either a node gets added or it doesn't If it doesn't then the procedure doesn't change the module state so the invariant is maintained

If a node is added, then we need to look at the invariant It says every node mentioned in the edge set is in the node set If that was true on entry, then since the edge set is unchanged by the procedure and the node set is only enlarged, it will still be true on exit Even if the graph started off empty, we'll get a graph with a node but no edges That's OK and it satisfies the invariant So in either case the invariant is maintained I think that's it for this procedure

The reader pauses momentarily to ensure there's an opportunity for discussion, then continues

Let's move on to the inverse procedure, sp_delete_node, which also takes a node as parameter and deletes it from the graph

Once again, we assume that sp_initialize has been called

There are no exceptions for this procedure but the effects are a bit more complex than sp_add_node First of all, the node is deleted from the nodes set That makes sense Then every edge in the edge set is

> examined, and if either end node is the node that was deleted from the nodes set, then that edge is deleted from the edge set Finally, the outputs are that if the node was in the node set on entry we return true, and otherwise we return false So as add_node didn't object to attempts to add a node that was already in the graph, similarly delete_node doesn't object to attempts to delete a node that was not there However, the return value tells us that's what happened, which seems inconsistent with the add_node behaviour where the user can't tell Does anyone else think that's a problem?

Some comment from the author and moderator is likely at this point The inconsistency is minor but might make the module more difficult to use It's worth noting that this inconsistency in the interface design was not deliberately seeded in Figure 3 3, but only came to light during the inspection exercise Finally, the completeness and invariant arguments for this procedure

> OK, let's deal with the completeness argument There are two possibilities here the case where the node specified is in the graph, and the case where it's not If it is in the graph to start off, then it gets deleted from the node set and edges that connect to it are also deleted from the edge set That seems reasonable If it's not in the graph, then the set operation of taking the node from the nodes set won't do anything Now, we'll also delete any edges involving that node from the edge set, but, since by the invariant property if the node wasn't in the node set it couldn't be on any edges, that also won't change anything That seems reasonable and it applies even if one or both of the sets was empty to begin with
>
> The reasonableness of these effects depends on the invariant property, so let's consider that On entry to the procedure the invariant property

holds In our first case, where the node is in the graph, we take it out of the node set and we take its edges out of the edge set Since we don't make any other changes to either set, that preserves the invariant property that every node on an edge, is in the node set If the node is not in the graph, the graph isn't changed so the invariant is trivially maintained That concludes my argument for sp_delete_node

4.4.2 Proof technique: case partitioning

As illustrated by the preceding example, the most frequently useful approach to completeness and invariance arguments is the case partition The reader divides all the possible states and input values into a collection of cases, then argues each case in turn Typically, the individual case arguments are very simple the specification logic either straightforwardly handles the case or just as evidently doesn't take the case into account What most inspectors find more difficult is seeing the appropriate partition This is an important skill not only in the inspection role but for design, implementation and test planning Weakness at dividing the problem domain into a complete set of suitable cases is probably a cause of completeness defects in designs

The rigours of the inspection meeting bring this skill to the foreground, which motivates inspectors to improve Some basic guidelines can be taught, for example, to consider boundary conditions and the natural divisions according to data type shown in Table 4.1 Then the cross-product of state and input parameter cases must be addressed Facility at seeing the right partition comes with practice

Data Type	Cases
integer	negative, zero and positive values
string	null pointer, empty, non-empty
boolean	TRUE, FALSE
set	empty, partially-filled, full
record	groupings of key field values

Table 4.1 Standard cases by data type

4.5 Data collection

Collection of data for the purposes of both product and process improvement is a central tenet of the inspection philosophy The use of forms is standard, with very similar models of inspection meeting notices, defect logs, meeting reports and defect summaries appearing in the literature, e g , [Fag76, AFE84, Fow86, Hum89, KSH92] Defect information predominates, in keeping with the central mission of the inspection process Metrics on the inspection process itself receive attention, in conjunction with simple measures of product size used to support evaluations and comparisons of the process productivity Our data collection efforts generally followed this typical pattern

4.5.1 Defect data

During the inspection meeting, the recorder creates the Inspection Defect List, using the form shown in Figure 4 8

To minimize writing time, the recorder identifies defect locations by marking up a copy of the work product with the defect identification numbers (sequentially assigned from 1 as defects are recorded) The Tally column on the defect list counts multiple occurrences of the same defect

Circulation of this handwritten list is limited to the author and moderator, who use the defect descriptions to ensure the issues raised are resolved To provide defect metrics to those responsible for monitoring and product and process quality, the recorder prepares an Inspection Summary tables of defect counts by type, class and severity The elements of this categorization scheme are shown in Table 4 2 and an Inspection Summary form in Figure 4 9

The defect *type* identifies the general quality criterion violated We started with a larger number of categories (including performance, maintainability and human factors), modelled more closely on examples in the above-cited literature, but found

INSPECTION DEFECT LIST

Project: thesis examples

Item: spath Module Interface Spec, version 1 0

Id	Tally	Type	Class	Sev	Description
1		CO	W	P4	add_node, delete_node return vals
					inconsistent when null effect

Page ___ of ___

Figure 4 8 Inspection Defect List form

INSPECTION SUMMARY

Date 15Sep92
From John Doe

Project thesis examples
Item sample Module Interface Specification

		P1 Defects				P2 Defects			
		M	E	W	P1	M	E	W	P2
ST	standards								
CO	consistency								
DE	description					1			1
FN	functionality			3	3	5	1	1	7
SU	sufficiency					2		1	3
CM	completeness								
SI	simplicity						1		1
OT	other								
	Total			3	3	8	2	2	12

		P3 Defects				P4 Defects				Grand
		M	E	W	P3	M	E	W	P4	Total
ST	standards			2	2			1	1	3
CO	consistency			3	3					3
DE	description		5	9	14	2		5	7	22
FN	functionality	3	2		5					15
SU	sufficiency	7		1	8			1	1	12
CM	completeness									
SI	simplicity		1	1	2					3
OT	other	1			1					1
	Total	11	8	16	35	2	0	7	9	59

Figure 4 9 Inspection Summary form

Type	
ST	standards
CO	consistency
DE	description
FN	functionality
SU	sufficiency
CM	completeness
SI	simplicity
OT	other

Class	
M	missing
E	extraneous
W	wrong

Severity	
P1	critical
P2	severe
P3	minor
P4	non-functional

Table 4.2 Defect categories

that recorders did not use the narrower categories. The reduced list (in Table 4.2) closely reflects the concerns we had trained inspectors to look for: the quality criteria of Section 3.3.

The defect *class* is a standard categorization. While extraneous content is rare in module interface specifications, we retained this metric to help identify completeness problems (apt to be designated *missing*, even if the *completeness* type is not selected).

The defect *severity* rankings replicate the SSN department's established scheme for problem reporting, with the addition of category P4 for non-operational defects.

4.5.2 Process metrics

Basic inspection process measures of preparation time, meeting type, duration, and outcome are entered by the recorder on the Inspection Meeting Record form shown in Figure 4.10. Categories of *meeting type* and final *disposition* used on this form are listed in Table 4.3. The disposition of the work product is decided by the moderator,

Meeting Type
Overview
Inspection
Re-inspection

Disposition	
Accept	no defects found
Conditional	moderator to review corrections
Re-inspect	corrected work product must be re-inspected

Table 4.3 Inspection record categories

INSPECTION MEETING RECORD

Project: thesis examples

Item: sample Module Interface Spec

Mtg Type: Inspection

Date: 15Sep92 Start Time: 13 30

Disposition: Conditional End Time: 15 10

Role	Name	Prep Time
Moderator	Kate Chen	50 min
Reader	John Smith	80 min
Recorder	Bill Lee	20 min
Author	Bill Lee	
Inspector	Mary Broz	60 min

Preparation coverage ____ %

Notes

Figure 4.10 Inspection Meeting Record form

after general consultation The guideline is that conditional acceptance may be granted when the required revisions in the work product amount are expected to change it by less than 15% Work products expected to undergo more extensive change are normally slated for re-inspection The moderator may also subsequently elect to call a re-inspection on a work product granted conditional acceptance, if the revisions turn out to be more extensive than anticipated Re-inspection meetings are expected to proceed more quickly than initial inspections, i e , at up to twice the original pace

The *preparation coverage* item on the Inspection Record form is a measure of value of the inspection meeting over and above individual preparation by inspectors During the meeting, as each defect is logged by the recorder, its identification number on the defect list is announced so that each inspector can maintain a list of those items they noted during their preparation One such list might read

1, 2, 5, 7, 11, 12, 13, 14, 15, 19

At the end of the meeting, the recorder collects all these lists, which are anonymous, and computes the preparation coverage metric the percentage of inspection-meeting defects which were detected by *any* inspector during preparation

4.5.3 Product metrics

The difficulty and cost of creating, inspecting, and revising a Module Interface Specification may be expected to vary with the size and complexity of the work product The measures shown in Figure 4 11 were chosen to assess these factors

The work product *source* identifies the circumstance of its creation or revision, as listed in Table 4 4

Counts of *total lines* are measured after *man* page formatting (*nroff*), to avoid measuring variability in author usage of *man* macros, line breaks, and blank lines

As noted in Section 2 1 2, the metrics counting *access routines, inputs, outputs,*

MODULE INTERFACE SPECIFICATION METRICS

Project thesis examples

Item spath Module Interface Spec, version 1 0

Source New Date January 4, 1993

Attribute	Count
total lines	396
access routines	8
input variables	13
output variables	1
state variables	4
exception conditions	6
design time	12 hrs
rework time	1 5 hrs

Figure 4 11 Module Interface Specification Metrics form

Source	
New	Specification for a new module
Update	Revision of existing specification
Reverse-engineered	Specification created for existing module

Table 4.4 Module Interface Specification source categories

etc., are conceptually similar to Function Points. When counting inputs, outputs and state variables, record structures contribute the number of their constituent fields, e.g., a record of six fields counts as six variables. (Unless any fields are themselves records, in which case these in turn contribute the number of their constituent fields, and so on). Sets and sequences, however, are counted as a single variable of their component type, e.g., a sequence of six integers counts as one. Return values are included in the count of output variables.

Chapter 5

The pilot project

5.1 Introduction

After developing the methods and materials described in the preceding chapters, we conducted a pilot project at MPR Teltech Ltd , as set forth in the proposal (Appendix A) In terms of the technology transfer model of Section 2 1 3, we functioned as the technology *supplier* We selected and adapted software engineering methods that rated well on the *critical factors* advantage, compatibility, accessibility, suitability for trial, and visibility Our inside knowledge of the company helped us effectively promote *awareness* in *gatekeepers* and *management*, and secure resource commitments We proposed the pilot project as a recognized means of technology *assessment* by the company Happily, the characteristics of pilot projects that recommend them as technology assessment vehicles—cost-effectiveness and flexibility through restricted scope, meaningful results through participation of real workers doing real work—also made for a feasible research project

The pilot project had two major components first, training of the participating MPR Teltech employees in the techniques of module interface specification and inspection, then application of the new techniques to ongoing maintenance of the Digital Networks Support System (DSS) software product

5.2 Participants

The MPR Teltech employee participants seconded to the project comprised the Technical Leader and three staff members of the DSS development and support group. Software professionals at MPR Teltech are designated "Members of Technical Staff," which we henceforth abbreviate as MOTS.[1] These people were not volunteers, nor were they selected for possessing any special knowledge or skills, or lack thereof. Rather, they were the staff currently assigned (according to the departmental practice of rotating assignments for technical staff) to maintaining the DSS product. Management nominated their project to participate in our study because the relatively long history of the DSS software might provide a benchmark against which the effect of new methods could be assessed, and because current work on the product was of modest scope and risk. In summary, our results were achieved by and with employees of average ability and motivation.

A company Software Quality Specialist had responsibility for overseeing our project. As a recent former member of the SSN department, she was well qualified to evaluate our activities, electing to attend the training program and to observe some inspection meetings. We collaborated with her on initiating a trial metrics program in conjunction with our project.

One additional MOTS shared in the writing of two Module Interface Specifications and participated in the associated inspections, because he was assigned at the time to assist with the design and implementation of those modules. Although he did not participate in the training sessions, as a recent graduate of the University of Victoria he was acquainted with some of the background of our work, and succeeded

[1] Nomenclature for software professionals is problematic. Designations such as "programmer" and "analyst" are too narrow for workers whose responsibilities at different times include proposals, feasibility studies, product design, implementation, testing, field trials, customer support, etc. It is potentially misleading, and in some jurisdictions illegal, to term persons "software engineers" unless they are certified Professional Engineers, which computer science graduates rarely are.

in catching up with and contributing to the project

All these people hold degrees in computer science or computer engineering from Canadian public universities. Their working experience in software development ranged from two to seven years.

5.3 Training

The success of an innovation in software engineering technology clearly depends on its favourable reception and effective use by software engineers. The education or training received in the new technology strongly influences these outcomes, hence the training program is a critical component of a pilot project.

Because employee training is an overhead cost, MPR Teltech preferred that employee time dedicated exclusively to training be minimized. We therefore devised a program of four short sessions (of one half day or less), to be given at weekly intervals. Spreading out the training activity had several advantages. First, it minimized interruption of participants' regular work. In the SSN department working environment, it was easier for MOTS to spare a few hours each week from their regular tasks, than to arrange to be entirely absent from other duties for the equivalent concentrated two-day training session. Second, it gave participants a chance to rest and reflect upon concepts learned between sessions, rather than overloading them with more new material than was likely to be absorbed in a concentrated course. Finally, it provided the opportunity to assign practice work for students to tackle between training sessions. Because our principal learning objectives were *skills*—of creating and inspecting Module Interface Specifications—practice would be more valuable than presentations for helping students achieve competence [Kno86, Chapter 5].

To prove our training curriculum and fine-tune it to participant needs, at minimum cost to the company, a single MOTS was seconded to the project earlier than her fellows. We worked through the first two training sessions interactively with this

individual student, who wrote several Module Interface Specifications during this trial training period. This resulted in some early refinements to the work product and training plan. It also strengthened confidence in the viability of the project at MPR Teltech.

5.3.1 Objectives

The broad objectives of the training course were that participants acquire:

1. knowledge and comprehension of the principles of module interface design;
2. skills of designing module interfaces and documenting such designs as Module Interface Specifications;
3. knowledge and comprehension of the principles of inspection;
4. skills of carrying out inspection roles;
5. positive appreciation of this technology and willingness to participate in its further application;
6. ability to evaluate practice of the technology, propose improvements, and participate in planning new applications.

These general aims were elaborated as a detailed set of performance-oriented objectives, listed in the Course Description (Appendix F).

We acknowledged the importance of social issues in the technology transfer process by including affective learning objectives explicitly (item 5 above, details in Appendix F). Such an approach is recommended by current theories of adult education, which stress the importance of making all learning objectives explicit [Kno86]. Positive affective outcomes are not achieved by indoctrination—infeasible with competent adults—but can be encouraged by discussion of benefits and drawbacks that the instructor and students see in the technology.

5.3.2 Syllabus

The training syllabus (also detailed in Appendix F) had four sessions as follows:

1. *Introduction and module design*
   (a) Lecture introducing the project and training program, followed by introduction to the basic features of the Module Interface Specification.
   (b) Student assignment including comprehension and writing of simple Module Interface Specifications.
2. *Module Interface Specifications*
   (a) Lecture on advanced features of Module Interface Specifications.
   (b) Introduction to inspection.
   (c) Student assignment including more advanced comprehension and writing of Module Interface Specifications.
3. *Inspection*
   (a) Viewing of an inspection training videotape.
   (b) Follow-up lecture.
   (c) Student assignment of preparation for an inspection meeting.
4. *Inspection practice*
   (a) A practice inspection meeting, with the instructor serving as Reader.
   (b) Discussion of the inspection techniques during periodic "time out" from the practice meeting.

Lectures addressed knowledge objectives, while assignments provided opportunities for developing skills of application and evaluation. Writing assignments involved

reverse engineering Module Interface Specifications for real DSS modules, which would have lasting practical value beyond the learning experience Examples of lecture notes and an assignment are provided in Appendix G

Inspection meeting skills were introduced to the students via training videotapes, then practised in a participatory demonstration The Module Interface Specification inspected during the practice meeting was seeded with several defects, ranging from trivial document defects to a subtle flaw in the interface described The instructor served as the Reader during this meeting, in order to demonstrate our paraphrasing techniques described in Chapter 4 (The training videotapes showed scenes of code inspection meetings, using intuitive paraphrasing techniques This was not adequate to prepare students to paraphrase Module Interface Specifications) During this training session, we alternated 20-minute periods of "inspection meeting," in which all participants adhered to inspection meeting roles and protocols, with 10-minute "time out" periods for questions and discussion evoked by the preceding inspection period By deferring questions and comments until the next "time out" we ensured some meaningful immersion in inspection-meeting roles and situations

5.3.3 Results

The initial trainee, assigned to help us test out and refine the technology and training program, tackled all the student assignments associated with the training program In particular, she wrote a total of three Module Interface Specifications first two reverse engineered specifications of existing DSS modules, subsequently an original specification for a new module that she had been assigned to design Her facility and competence at preparing Module Interface Specifications clearly increased as she gained experience It was also the most probable cause of subsequent superior competence at inspection preparation, described below

By contrast, the subsequent participants, although faithful in attendance at the

class sessions, attempted few out-of-class assignments, reportedly due to pressures of other work. This detracted from the hoped-for benefit of interspersing classroom presentations with skill-building assignments. As much of the software industry is noted for heavy demands on employee time, we conclude that it is not prudent to expect industrial students to accomplish significant out-of-class reading or "homework." Classroom presentations should be comprehensive, including material that in an academic setting might profitably be assigned as student reading. Also, since skills development during the training program is inevitably modest, it is critical that early application *on the job* be strongly supported by educators or technology suppliers. Subtle issues of understanding and application are apt to arise not during training activities but rather when workers are obliged to put the new technology to use in their work. This phenomenon did not cause any particular difficulties for us because a pilot project lends itself well to such support. Although not anticipating that MOTS have quite so much difficulty finding time for training assignments, we had also anticipated providing significant educational support during the application portion of the project, because the formal training program was minimized from the beginning.

Participants' early efforts at writing Module Interface Specifications exhibited weaknesses in abstraction and use of logic. Module abstract states tended to be closely modelled on implementation states (a natural temptation when reverse engineering the specification), carrying inappropriate implementation detail and complexity into the interface specification. Although MOTS had encountered propositional and predicate logic during their university training, they were not accustomed to using this apparatus to describe and reason about software entities. This unfamiliarity caused awkwardness at formulating and understanding *conditions* in the Module Interface Specification, such as initial conditions and invariant properties of the module state, and assumptions of access routines.

It is important to note that the predicates required to express relevant conditions were not complex, and there was little or no manipulation thereof to be performed The essential problem with the use of logic was not a matter of difficulty with *logic* per se [2] Rather, the problem is that software engineers are not accustomed, by either academic training or workplace practices, to explicitly *use* logical concepts in designing and reasoning about programs Thus, outright omission of necessary state conditions was a frequent defect of first efforts at writing a Module Interface Specification, and development of inspection-meeting arguments concerning invariance and completeness was initially challenging, as described in Section 4 4 2 This experience illustrates a significant barrier to the adoption of thorough-going Formal Methods such as Z, VDM et al [Win90] by the software industry

The task of reverse engineering specifications for existing modules presented peculiar difficulties of abstracting from the implementation, especially of oversized modules with complex effects On a more positive note, specification tables proved helpful in elucidating complex and confusing code

The videotaped inspection training materials proved very effective at imparting some understanding of what to expect an inspection meeting to be like, and how to behave in this setting While lectures and written materials can convey facts and principles, a demonstration inspires confidence in the onlookers that the process is comprehensible and feasible

At the beginning of the practice inspection meeting, people were hesitant to speak up and participate After the first discussion break, at which there was some discussion of the need to overcome this hesitancy, the reader increased the pauses between items and other inspectors began to interject comments All 14 seeded work product defects were detected, as were 7 other unintended defects

[2]Our curriculum included a brief review of elementary formal logic MOTS, as high-achieving university graduates, had few intrinsic difficulties with this material, although a more thorough review of the use of quantifiers would have been helpful to some

Even though the work product was known to be a reverse-engineered specification of an existing module, valid concerns were raised as to the quality of the interface design (inappropriate assumptions imposed on the caller), and of comments in the associated interface data declarations (insufficiently informative) The scrutiny of inspectors was evidently more than superficial

The seeded defects included a subtle sufficiency defect, which we had feared would be too difficult to detect One of the inspectors suggested that something didn't seem quite right in the relevant area of the specification, but attributed the trouble incorrectly The initial trainee then spoke up and explained the defect flawlessly, remarking that her extra preparation time (more than the other inspectors) had been spent on tracing this problem This gratifying result impressively demonstrated the rewards of experience at using the new techniques

Although the team together found many problems, it seemed clear that no single person had noted all or nearly all Each person appeared to have strengths and weakness at detecting different types of problems Some problems that had evidently escaped all participants during preparation were detected as a result of discussion It was as a result of this early observation that we devised the preparation coverage metric (see Section 4 5 2), to measure the proportion of problems detected only in the meeting

5.4 Application

In the course of the project (including the training program), Module Interface Specifications were written for 15 DSS modules two by the author, ten by MPR Teltech MOTS, and three by MOTS in collaboration with the author The collaborative efforts occurred in early work with the initial trainee, and in evolving an approach to specifying the semantics of interaction with the DSS user interface Five MPR Teltech and two collaborative specifications were inspected—with the collaboration

Even though the work product was known to be a reverse-engineered specification of an existing module, valid concerns were raised as to the quality of the interface design (inappropriate assumptions imposed on the caller), and of comments in the associated interface data declarations (insufficiently informative) The scrutiny of inspectors was evidently more than superficial

The seeded defects included a subtle sufficiency defect, which we had feared would be too difficult to detect One of the inspectors suggested that something didn't seem quite right in the relevant area of the specification, but attributed the trouble incorrectly The initial trainee then spoke up and explained the defect flawlessly, remarking that her extra preparation time (more than the other inspectors) had been spent on tracing this problem This gratifying result impressively demonstrated the rewards of experience at using the new techniques

Although the team together found many problems, it seemed clear that no single person had noted all or nearly all Each person appeared to have strengths and weakness at detecting different types of problems Some problems that had evidently escaped all participants during preparation were detected as a result of discussion It was as a result of this early observation that we devised the preparation coverage metric (see Section 4 5 2), to measure the proportion of problems detected only in the meeting

5.4 Application

In the course of the project (including the training program), Module Interface Specifications were written for 15 DSS modules two by the author, ten by MPR Teltech MOTS, and three by MOTS in collaboration with the author The collaborative efforts occurred in early work with the initial trainee, and in evolving an approach to specifying the semantics of interaction with the DSS user interface Five MPR Teltech and two collaborative specifications were inspected—with the collaboration

Source†	Name	Description
R-E	netso	Network scheduling computations
R-E	difpd	Query interface to remote database server
New	comap	Generic interface to various communications server tasks
New	comll	Generic user login/logout to various network elements
R-E	conaf	User interface support for DSS administrative functions
R-E	difpx	Export interface to remote database server
New	difdd	Builder of export transactions to remote database server

†See Table 4.4 for categories. "R-E" is an abbreviation for "reverse-engineered."

Table 5.1 Module Interface Specifications inspected

Module Name	MIS Size Pages	MIS Size Lines	Acc Rtns	Inp Vars	Out Vars	State Vars	Exc Cons
netso	4.7	462	6	19	18	0	1
netso†	5.6	594	6	19	18	0	3
difpd	3.5	396	5	4	8	0	10
comap	8.0	792	7	36	13	19	17
comll	2.5	264	2	8	2	0	4
conaf	6.1	594	7	9	10	11	7
difpx	3.9	396	6	4	7	8	0
difdd	3.5	396	6	17	40	0	0
Total	37.8	3894	45	116	116	38	42

†At re-inspection

See Figure 4.11 for unabbreviated column headings.

Table 5.2 Module Interface Specifications inspected—metrics

occurring during the follow-up revisions. All of the trainees wrote specifications and served at least once in each of the inspection meeting roles. The specifications inspected are described in Table 5.1, followed by work product metrics in Table 5.2.

The metrics of Table 5.2 are as per Section 4.5.3, with the addition of page counts estimated by looking at the printed (*troff'ed*) documents.

Table 5.3 lists the inspection meetings, with some basic Inspection Record data, as described in Section 4.5.2. The application phase of the project comprised the number of inspections specified by the Proposal (eight, see Appendix A), but took place over a greater number of elapsed weeks due to scheduling difficulties caused by employee vacations during the summer months, and preemption by various urgent

Date 1992	Mtg Type	Module Name	Lines Data	Lines MIS	Team Size	Mins Prep Total	Mins Prep Med	Mins Mtg
Jul 15	Insp	netso	72	462	6	355	53	110
Jul 29	Re	netso	86	594	4	100	25	90
Jul 29	Insp	difpd	0	396	4	115	30	90
Aug 07/24	Insp	comap	444	792	4	305	68	225
Aug 14	Insp	comll	0	264	4	40	10	60
Oct 02	Insp	conaf	0	594	4	140	35	100
Nov 02	Insp	difpx	0	396	4	110	20	60
Nov 09	Insp	difdd	128	396	4	65	15	120
Totals per median participant			730	3894			256	855
Totals for all participants						1230		3640

Table 5 3 Inspection meetings held

assignments in the early fall

As shown by the tables, the **netso** Module Interface Specification (MIS) was inspected then subsequently re-inspected, after revisions which increased its size This was our only re-inspection meeting All other MIS were conditionally accepted on first inspection Two meetings were required to inspect the **comap** MIS, which was large and complex [3] We inspected another MIS in the week between these two meetings because the principal author of the **comap** MIS was away on vacation

Where appropriate, module exported data was inspected along with the MIS The data-file line counts include commentary lines, contravening the standard definition cited in Section 2 1 2, because the comments, which occupied more lines than the source code, were inspected

At each inspection meeting, the Recorder collected the individual preparation times of all inspectors This information appears in summary form in Table 5 3, in the double column entitled "Mins Prep" Under this heading, the "Total" column contains the sum of the preparation times of all participating inspectors, and the

[3]The inspection literature recommends that meetings not exceed two hours in duration, as productivity appears to fall off beyond this point, presumably due to loss of concentration We adhered to this guideline

"Med" column the median of these values, for each meeting We expected the median preparation times to be more representative than the mean (i e , total divided by team size), because Authors typically report minimal time spent preparing for the meeting per se, while Readers tend to spend extra time ensuring they are ready to paraphrase the material As it happened, these two divergences from the norm were approximately equal in magnitude, hence mean and median preparation times for each meeting are about the same

The "Totals" rows in this table show that the time spent on inspection activities by a median participant in this project was about 4 25 hours of preparation and 14 25 hours in inspection meetings The total time spent by all participants was 20 5 hours of preparation and 60 67 hours in inspection meetings

5.4 1 Metrics

Eight or nine inspection meetings is too small a sample to support statistically sound analysis The purpose of instituting the measurement program during the pilot project was therefore not to obtain sound quantitative results Rather, we intended to identify appropriate metrics and start collecting data so that MPR Teltech may be able to carry out meaningful analysis in the future, after accumulating adequate data In addition, while recognizing the limitation of our small sample, we examined the data from the pilot project for preliminary indications as to the costs and effectiveness of the inspection activity

Preliminary indications

Tables 5 4 and 5 5 show some inspection data and calculations of inspection and defect detection rates for the pilot project

According to Table 5 4,[4] MIS inspection meetings proceed at rates of between 2 5

[4]The primary data in Table 5 4 is recapitulated from Tables 5 2 and 5 3 The calculation of

Mtg Type	Module Name	Lines Data	Lines MIS	Pages Total†	Median Mins Prep	Median Mins Mtg	Pages/Hour Prep	Pages/Hour Mtg
Insp	netso	72	462	5 8	53	110	6 6	3 2
Re	netso	86	594	7 4	25	90	17 7	4 9
Insp	difpd	0	396	4 0	30	90	7 9	2 6
Insp	comap	444	792	15 3	68	225	13 5	4 1
Insp	comll	0	264	2 6	10	60	15 8	2 6
Insp	conaf	0	594	5 9	35	100	10 2	3 6
Insp	difpx	0	396	4 0	20	60	11 9	4 0
Insp	difdd	128	396	6 1	15	120	24 4	3 0

†LinesData/60 + LinesMIS/100

Table 5 4 Inspection rates—median preparation and reading

and 4 source document pages per hour Our single re-inspection meeting proceeded at a faster rate, as expected Preparation rates are typically about three times the inspection rate in the meeting, although there is much variation in this

Monitoring of preparation and inspection rates is important for managing and improving the inspection process There is an evident need for more data and analysis, in order to understand the sources of variation and determine optimal rates for MIS inspections at MPR Teltech While line and page counts are intuitively easy to work with, the other work product measures (number of access routines, inputs and outputs etc) may be more indicative of work product complexity No simple relationships based on these measurements are evident in the small pilot project data sample

Table 5 5 adds preparation coverage (see Section 4 5 2) and defect counts to data recapitulated from Table 5 3 The preparation coverage figures show that only about half of the total defects recorded were detected during preparation by any inspector, for the three inspection meetings at which this data was collected [5]

---

nominal page counts is based on the observation that in Table 5 2, MIS line counts (measured on the *man* formatted documents, as described in Section 4 5 3) are consistently about 100 times the visually estimated page count of the respective printed documents (the mean ratio is 103 with a standard deviation of 6) Data files are printed at the standard 60 lines per page

[5]Authorization to collect this information was delayed during finalization of the concurrent

Mtg Type	Module Name	Team Size	Minutes Prep	Minutes Mtg	Prep Covg	Defects Total	Defects Func	Defects/Hour† Total	Defects/Hour† Func
Insp	netso	6	355	110	?	34	9	2.0	0.5
Re	netso	4	100	90	?	11	6	1.4	0.8
Insp	difpd	4	115	90	?	19	5	2.4	0.6
Insp	comap	4	305	225	?	58	27	2.9	1.3
Insp	comll	4	40	60	?	24	5	5.1	1.1
Insp	conaf	4	140	100	56%	32	9	3.6	1.0
Insp	difpx	4	110	60	63%	11	7	1.9	1.2
Insp	difdd	4	65	120	42%	7	5	0.8	0.6
Total defects						196	73		

†Defects/(MinsPrepTotal + TeamSize × MinsMtg) × 60

Table 5.5 Inspection meeting productivity—defects detected

This table presents separate counts of Total and Functional defects. Functional defects are all those of types FN (functionality), CM (completeness) and SU (sufficiency), according to the Inspection Summaries (see Section 4.5.1). Defects of these types may be expected to lead to implementation of software that will fail under some conditions. The other types of defects typically make the MIS or the module itself more difficult for people to use or maintain. Such defects may increase costs or the risk of programmer error, but are not themselves direct causes of software failure. MPR Teltech therefore regarded functional defects as more significant than the other types, regardless of severity. Table 5.5 shows that the pilot project inspection process yielded about one functional defect per person-hour, including time spent by all participants, both in preparation and in the inspection meeting. This result matches the consistent testimony of the inspection literature: one non-trivial defect detected per person-hour invested [BD84, Hum89, Rus91, KSH92].

All of the above-cited studies include code inspection, some also report on inspection of one or more of requirements specifications, high-level or "architectural" designs, and detailed designs. The form and content of the requirements and high-

---

metrics project

level design documents in question are not described, but they are probably dissimilar from each other, from Module Interface Specifications, and also from code ("Detailed design" typically means pseudo-code). Why would the effort expended per defect detected be about the same for these dissimilar work products? Buck and Dobson call this (one defect per person-hour) and some related constants observed at IBM "natural numbers of programming," suggesting that they are inherent to the *process* rather than the personnel, product, tools, etc. [BD84]. Recalling that inspection rates vary with the work product type and content, the consistency of the result may also suggest that defect incidence and detection are approximately proportional to complexity or some broader measure of intellectual content. Achievement of the "industry-standard" defect detection rate could then be taken to indicate that the local inspection process exhibits normative efficiency.

Open questions

If module interface specifications and inspection are effective technologies at MPR Teltech, one would hope to observe:

1. Improved quality of module interface designs.
2. More defects detected by inspection than by individual review.
3. Fewer defects detected during subsequent development phases (implementation and testing), and in the field.
4. Reduced costs over the entire software lifecycle (development and maintenance).

Item 1 is difficult to measure, because specific quality criteria of interest must be identified and these are not always easily quantifiable. (For example, our *comprehensibility* criterion). There is no existing base of quality measurements on DSS

modules, to use for comparison There is, however, an existing base of defect data It would be reasonable to evaluate the success of our focus on *completeness* by studying whether the relative incidence of defects attributable to completeness problems declines, especially in subsequent phases of the software process (item 3)

The preparation coverage metric addresses item 2 The finding that only about half the defects recorded during inspection meetings were noted during preparation (see Table 5 5), contrasts strikingly with the group's normal practice of solo reviewing It would be appropriate to refine studies in this area to determine what kinds of errors tend to go undetected, and optimal numbers of inspectors or reviewers For example, in an experiment using inspection at the Software Engineering Laboratory, it was determined that 75% of code defects were detected by only one inspector [BGKW90] As a consequence, new SEL software development guidelines now require all code to be read by at least two reviewers, even on projects not using inspections [LWM+92, page 117]

Items 3 and 4 represent the acid test of whether or not the technology provides direct economic benefit to the company These are long-term assessments, well beyond the scope of the pilot project Different categories of work should be considered—for example, it is quite possible that inspection may not be cost-effective in all cases The Defect Study (Appendix B) identified that defects tend to cluster in certain modules, these are obvious candidates for the intensive scrutiny afforded by the inspection process The MPR Teltech metrics project is attempting to put in place means of evaluating such questions Our specific recommendations for future studies are reproduced in Section 5 6

5.4.2 Experiential results

The theory of software technology transfer supports the common-sense intuition that less quantitative observations—such as perceived social effects in the work

environment—may also be relevant when evaluating the impact of technological change This is particularly true of a technology such as inspection, which is itself primarily a social process

The difficulty of reverse engineering MIS for existing modules was mentioned above with respect to the training program (see Section 5 3 3) It was originally intended that the application phase of the project should address new modules being designed for the next major release of DSS Intervening revision and rescheduling of the product evolution plan greatly reduced the new module design activity scheduled during the project time frame As a result, most of the MIS written during the pilot project documented existing modules slated for maintenance activity This has the advantage for beginners of eliminating the effort associated with the design task, and focusing attention on specification alone However, some of these modules were several years old, and while nominally still "modules," had grown into virtual sub-systems of substantial size and complexity For such subjects, the MIS format—designed for precise specification of smaller units—tended to become cluttered with a mass of time-consuming detail The difficulty was exacerbated in some cases by lack of appropriate supporting documentation that might be referenced—for example, detailed requirements specifications We attempted to exercise judgement in omitting certain types of detail from the MIS The subjectivity of this proceeding was unsatisfying—although also a useful learning exercise—and the resulting incompleteness undermines the value of the documentation Nonetheless, the MIS format did prove adaptable to these less-than-ideal circumstances In particular, we evolved an approach to specifying modules with user interface interaction, which required some extra MIS features to account for the DSS user interface model This robustness of the work product is important, because industrial software not uncommonly exhibits irregularities such as the foregoing

Project participants found working on specifications for new modules much more

satisfying, noting that the MIS approach suits the normative module size and complexity well,[6] and that the MIS format seemed to prompt consideration of "the right questions" to achieve a complete and useful design.

By their nature, Module Interface Specifications call attention to interface design. In our inspection meetings, people frequently pointed out weaknesses in the interfaces of existing modules for which specifications had been reverse engineered—such as, redundant or confusing parameters, inconsistent behaviour of different access routines, or functionality that might better have been allocated to a different module altogether. These were not infrequently designs in which the inspectors themselves had formerly taken a hand, what was new was the explicit consideration of interface design criteria. This supports the expectation that the use of MIS and inspection has potential to reduce the incidence of unsystematic interface design, leading to improved module usability and maintainability and thereby reduced errors and costs.

Most of our inspection meetings involved four participants (including the instructor/researcher) and appeared to achieve an industry-standard rate of defect detection (see Section 5.4.1). Judging by appearances, meetings of three inspectors would be satisfactory for the majority of work products, with additional experts perhaps being called upon to help inspect unusually complex or specialized pieces of work, or when high reliability is a critical requirement. By contrast, six to eight or even more participants appear to be commonplace at inspection meetings elsewhere in industry. Based on our research, we are skeptical of the marginal value of adding extra participants. It is difficult to envisage that this would be cost effective at

[6] SSN department guidelines specify that a module should be of 300 to 500 C language statements in size, with component functions in the range of 30 to 100 C statements. These limitations, combined with conventional recommendations of low coupling and high cohesion, and expectations that the entire task can be handled by one designer and one implementor, ensure that normative modules conform well to D. L. Parnas' definition as "a programming work assignment" (cited in Section 2.2.1).

MPR Teltech

During the first couple of inspection meetings we were repeatedly obliged to delay proceeding in order to allow the recorder to finish writing up a defect. It was as a result of this experience that we devised the simple scheme of identification numbers and tallies for identifying defect locations, described in Section 4.5.1. This approach satisfactorily eliminated waiting for the recorder.

Lack of interpersonal difficulties was a pleasantly distinctive feature of our experience with the inspection process. As noted in Section 4.2.2, the moderator's role was not found to be taxing or problematic, even though we did not particularly emphasize conflict management skills for moderators. Likewise, the involvement of the Technical Leader did not appear inhibiting, although the inspection literature strongly warns against the involvement of managers or supervisors, on the grounds that objectivity will be compromised due to concern about career impacts on the part of subordinates. Our lack of difficulty in this area is perhaps indicative of variation in working environments, attitudes, and expectations across the software industry. At MPR Teltech, the ethos of the working environment is overtly well-disciplined and professional, and MOTS are accustomed to sharing and rotating responsibility for work products rather than preserving personal "ownership" of particular pieces of work. Procedures for personnel evaluation are carefully specified and controlled, so participants had no disposition toward anxiety about misuse of inspection data. In such an environment, the interpersonal hazards of initiating an inspection program may be relatively modest, even when (as in this case) the participants are little accustomed to dealing with other people as part of their work duties.

5.5 Response

Our principal observations of participant responses during the project are described in the preceding section. In addition, in January 1993, MPR Teltech conducted a

meeting of participants to solicit candid feedback on the project This meeting was attended by employees only, we received a written summary of the results, which did not identify individual persons

This material confirmed our observations of the preceding section and portrayed a highly positive view of the technology and pilot project The only negative issues were skepticism over the value of reverse engineering specifications for the types of older (and highly stable) modules described in the preceding section, and dislike of the *troff* document processor The value of incorporating documents into the *man* page system was affirmed, since *man* pages can now be prepared by Framemaker, this newer word processing technology is likely to be preferred in future

Responses particularly emphasized the secondary benefits of inspection meetings for providing an open forum for feedback on designs (since peers sometimes have more knowledge than the Technical Leader of the context and issues addressed by a particular piece of work), for improving visibility of the current project status and changes, for learning about parts of the system outside of one's currently assigned responsibilities (helpful for future work), and for enhancing team cooperation This contrasts with the SSN department's "pens down" protocol for software engineering, in which group meetings are minimized in the interests of promoting individual responsibility and creativity, and avoiding time-wasting conversational discussions [Cro86, page 4] Inspection meetings appear to present a controlled forum in which MOTS may benefit from team synergy and information-sharing, without unduly dissipating individual responsibility and productivity

Invariance arguments (seen as particularly effective for finding defects), the inspection meeting scenarios in the training videotape from Bell-Northern Research, Ltd , and the instructor's modelling of inspection roles during meetings were singled out for praise The ability "to present criticism without personal attacks" was identified as an important skill that participants felt they had learned from the project

Although people appeared very comfortable doing this from the first (as discussed in Section 5 4 2), this response suggests that the training aimed at fostering this skill was not redundant, but at the least inspired self-confidence in essaying negative as well as positive feedback

The use of short checklists to guide inspection meetings, ensuring that no issues are missed, was recommended as a potential improvement to the process It was thought that the project had made the DSS group more conscious of software quality issues Participants unanimously recommended future use of this technology in their department, expressing concern that the main obstacle to this would be time and resource pressures in other areas, which might make it difficult for managers to adequately support deployment of the new technology

5.6 Recommendations

While recognizing the resource pressures that managers face, also amply documented by previous studies of software technology transfer (see Section 2 1 3), we contend that other pressures increasingly mandate efforts to improve the software process in industry The following paragraphs constitute the recommendations of our final project report to MPR Teltech Ltd

> The increasing pressures of international competition and demands for software process quality certifications are widely recognized in the software industry today [BM91a, You92] Such pressures must be met through continuous improvements in the software development process Since the techniques introduced by the pilot project appear viable and productive, we recommend their cautious adoption or continued exploration, along with the metrics project While it may not be possible to write and inspect Module Access Specifications for all modules, it would seem important to attempt to do this for known or anticipated

trouble spots The defect study showed that problems cluster in certain modules

The following issues might be investigated

- Categorization of defects The defect type categories used in the pilot project seem to have been more confusing than useful [7]
- Relationship between the production and inspection of MASs for new modules and costs of implementation, testing and maintenance
- Relationship between number of defects found in inspection and number of defects found by testing
- Relative costs of defect detection and problem resolution via testing and inspection
- Inspection of other work products Note that the Cleanroom technique [Dye92] involves *replacement* of unit testing by verification-oriented code inspection
- Defect prevention (based on analysis of defect reports)

[7] This issue was discussed further in an appendix to the report We allude to it in reference to the reduction in defect type categories, in Section 4 5 1

Chapter 6

Conclusions

6.1 Summary of results

The foremost result of this work is the unequivocal demonstration of the viability of the two techniques, module interface specification and inspection. Both were applied to a convincing sample of real industrial software modules, by industrial practitioners who claimed to find the techniques useful and recommended their adoption for regular use.

Attention to known issues of software technology transfer seemed helpful in gaining a positive response from project participants and managers. We believe that our "insider" knowledge of the company was indispensable to the success of this project. We brought a well-informed understanding of existing needs, methods, and terminology to the tasks of choosing, adapting, presenting and teaching candidate technology. The credibility of our recommendations was enhanced by previously established working relationships.

It is indeed difficult to change the way things are done in the working environment, especially in the face of the resource and schedule pressures typical of the software industry. Management skepticism about change under these conditions is understandable, and likely to be combatted only by concrete evidence. The technology we chose rated well on the critical factors *relative advantage, compatibility,*

accessibility, suitability for trial, and *visibility* of *hard* or technological results. However its *soft* benefits—principally, cost reduction—are anticipated to accrue only in later phases of the software lifecycle [1] This makes it more difficult to secure management commitment. Evidence of soft benefits could be obtained by continued small-scale application in conjunction with metrics efforts. The pilot project also served to demonstrate the general feasibility of software process innovation. New methods were tried, workers were able to use them successfully and judged them useful. Such experience helps create a positive climate for further innovation.

The viability result confirms our hypothesis that module interfaces would be a good starting point for introducing software process change. "Modules" are indeed not too difficult to find, and there are immediate benefits of usability and maintainability from documenting interfaces. The Module Interface Specification work product is sufficiently flexible to accommodate over-sized and highly complex modules, although best results are obtained with modules of more standard size and complexity (see Section 5.4.2).

Our pragmatic approach to use of formal notations in the specifications was well accepted by practitioners. Nonetheless, the writing and inspection of Module Interface Specifications is time consuming,[2] and there were unsatisfying trade-offs between specification time and precision. There is evidently room for more work at finding the best descriptions for interface designs.

The current interest in software product *reuse* also has relevance to the issue of specification cost. It is difficult and risky to reuse a software artifact whose behaviour is not precisely specified. Interest in reuse is particularly strong in the world of object-oriented software, hence it is significant that several speakers at the

[1] The italicized terms are defined in Section 2.1.3.

[2] Collection of effort data commenced only at the end of the pilot project. Recorded design times for the last three specifications inspected ranged from two hours to two days.

OOPSLA '92 conference identified interface specification as a *sine qua non* of reuse [Atk92, Coo92, dLF92, Mey92b] As the software industry moves towards a culture of reuse, it will be a culture of fewer and better software components, and they will necessarily be well specified Reuse makes specification more compelling and more affordable

The pilot inspection process detected defects at a rate of one functional defect per person-hour (preparation and meeting time of all participants) This is the same rate reported by other companies that use inspection About half[3] of the defects detected in the meetings were not detected during preparation by any meeting participant These satisfactory-seeming results were obtained by participants—including the instructor—with no prior experience of the inspection process We attribute them to careful training, centred about an excellent videotaped demonstration, and customization of forms and checklists for the local environment As inspection neophytes, some of the inspection literature led us to be concerned that we might not be able to reproduce the apparently incalculable and mysterious qualities of the inspection process For example, Fagan writes

> When participants in the moderator training classes are questioned about their case studies, they invariably say that they sensed the presence of the "*Phantom Inspector*," who materialized as a feeling that there had been an additional presence contributed by the way the inspection team worked together The moderator's task is to invite the Phantom Inspector [Fag86]

Happily, our results suggest that straightforward application of the more tangible principles and procedures set forth in this same literature can yield an effective and even synergistic process We have contributed specific training materials, checklists, procedures, metrics, and forms for the inspection of Module Interface Specifications, a useful type of software design documentation which has not been addressed by previous research on inspection

[3]According to data collected at three meetings, and consistent with personal observation

Participants particularly noted the secondary benefits of the inspection process in promoting education, team work and attention to quality. We observed that the public process of inspection appeared to foster more discriminating attitudes about quality of work. When this happens in a group process, quality standards tend to emerge and be supported by consensus. This promotes a culture in which workers advance from awareness, through willing compliance to active self-identification with standards.[4] Intuition suggests that committed practitioner support of this kind is going to be very helpful if not essential to achieving software process improvement at advanced levels (whether according to the SEI taxonomy or some other scheme). The inspection process contributes to establishing a culture of commitment.

6.2 Future work

The most immediate open questions concern the impact of this work in the context of the overall software lifecycle, in pragmatic terms of costs, and incidence and detection of defects. How do these methods compare with other forms of design documentation and verification? How are later phases of the software lifecycle (implementation, testing, maintenance, and reuse) affected? Such questions could be addressed by longer-term studies, with particular attention to metrics.

Analysis of more defect data, collected both from MIS inspections and from subsequent process phases, could guide *refinement* of these methods toward defect prevention. The labour-intensive nature of inspection is evident, it would be very useful to determine whether it is more cost-effective in certain cases than in others. More data on preparation coverage, and analysis of the types of defects missed, would help answer this. Some organizations allegedly find the preparation process much more productive than the inspection meeting. Do inspection experience and support materials such as checklists enable people to perform increasingly thorough

[4]This description follows the standard taxonomy of affective learning objectives [KBM64].

individual reviews?

As recommended to MPR Teltech (see Section 5.6), further investigation of the inspection technique should also extend its application to other work products. Our approach to specifying and inspecting module interface designs is philosophically compatible with the ideas of correctness-oriented software development underlying the Cleanroom technique [Dye92], of which verification-based inspection is a major component. Cleanroom has not yet been extensively used outside IBM; case studies in other environments would be of interest.

The preceding section makes mention of the need for more work at finding the best methods and notations for specifying interfaces and software designs generally. Practitioners find existing formal methods intimidating and cumbersome; less formal approaches, such as we used in our Module Interface Specifications, risk semantic imprecision and are less open to automated support. There is need for comparative case studies and techniques that bring together the more advantageous features of both approaches.

Bibliography

[ABL89] A Frank Ackerman, Lynne S Buchwald, and Frank H Lewski Software inspections An effective verification process *IEEE Software*, 6(5) 31–36, May 1989

[AFE84] A Frank Ackerman, Priscilla J Fowler, and Robert G Ebenau Software inspections and the industrial production of software In H L Hausen, editor, *Software Validation*, pages 13–40 Elsevier Science Publishers B V , Amsterdam, 1984

[ANS87] IEEE recommended practice for software design descriptions Technical Report ANSI/IEEE Std 1016-1987, The Institute of Electrical and Electronic Engineers, Inc, New York, July 1987

[Atk92] Bob Atkinson Reuse Truth or fiction (panel) In *OOPSLA'92 Conference Proceedings*, pages 41–42 ACM Press, 1992

[BCM+92] Victor R Basili, Gianluigi Caldiera, Frank McGarry, Rose Pajerski, Gerald Page, and Sharon Waligora The software engineering laboratory—an operational software experience factory In *Proceedings of the Fourteenth International Conference on Software Engineering*, 1992

[BD84] Robert D Buck and James H Dobbins Application of software inspection methodology in design and code In H L Hausen, editor, *Software Validation*, pages 41–55 Elsevier Science Publishers B V , Amsterdam, 1984

[BGKW90] Victor Basili, Scott Green, Ara Kouchakdjian, and David Weidow The cleanroom case study in the software engineering laboratory Project description and early analysis Technical Report SEL-90-002, National Aeronautics and Space Administration, Systems Development Branch, Code 552, Goddard Space Flight Center, Greenbelt, MD, 20771, March 1990

[BL91] Valdis Berzins and Luqi *Software Engineering with Abstractions* Addison-Wesley, 1991

[BM91a] Victor R Basili and John D Musa The future engineering of software A management perspective *IEEE Computer*, 24(9) 90–96, September 1991

[BM91b] J N Buxton and R Malcolm Software technology transfer *Software Engineering Journal*, pages 17–23, January 1991

[Boe76] Barry W Boehm Software engineering *IEEE Transactions on Computers*, C-25(12) 1226–1241, December 1976

[BP84] Victor R Basili and Barry T Perricone Software errors and complexity An empirical investigation *Communications of the ACM*, 27(1) 42–52, January 1984

[Bri88] Robert N Britcher Using inspections to investigate program correctness *IEEE Computer*, 21(11) 38–44, November 1988

[Bro75] Frederick P Brooks, Jr *The Mythical Man-Month* Addison-Wesley, 1975

[Bro87] Frederick P Brooks, Jr No silver bullet Essence and accidents of software engineering *IEEE Computer*, 20(4) 10–19, April 1987

[CDS86] S D Conte, H E Dunsmore, and V Y Shen *Software Engineering Metrics and Models* Benjamin/Cummings, Menlo Park, CA, 1986

[CG90] David N Card and Robert L Glass *Measuring Software Design Quality* Prentice-Hall, 1990

[CH87] D A Christenson and S T Huang Code inspection management using statistical control limits In *Proceedings of the National Communications Forum*, volume 41, pages 1095–1100, 1987

[Chi88] Elliot J Chikofsky Issues in the transfer of software engineering tool technology In S Przybylinski and P J Fowler, editors, *Transferring Software Engineering Tool Technology*, pages 14–15 IEEE Computer Society Press, 1988

[CKC91] Ram Chillarege, Wei-Lun Kao, and Richard G Condit Defect type and its impact on the growth curve In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 246–255 IEEE Computer Society Press, 1991

[Coo92] William R Cook Interfaces and specifications for the Smalltalk-80 collection classes In *OOPSLA'92 Conference Proceedings*, pages 1–15 ACM Press, 1992

[Cri91] Flaviu Cristian Understanding fault-tolerant distributed systems *Communications of the ACM*, 34(2) 56–78, February 1991

[Cro86] D B Crowson *MPR Design Methodology* MPR Teltech Ltd , May 1986

[Dei91] Lionel E Deimel Scenes of software inspections—video dramatizations for the classroom Educational Materials CMU/SEI-91-EM-5, Software Engineering Institute, Carnegie-Mellon University, May 1991

[DeM82] Tom DeMarco *Controlling Software Projects* Prentice-Hall, 1982

[dLF92] Dennis de Champeaux, Doug Lea, and Penelope Faure The process of object-oriented design In *OOPSLA'92 Conference Proceedings*, pages 45–62 ACM Press, 1992

[DLP79] Richard A De Millo, Richard J Lipton, and Alan J Perlis Social processes and proofs of theorems and programs *Communications of the ACM*, 22(5) 271–280, May 1979

[Doo92] E P Doolan Experience with Fagan's inspection method *Software—Practice and Experience*, 22(2) 173–182, February 1992

[Dye92] Michael Dyer *The Cleanroom Approach to Quality Software Development* Wiley, 1992

[Fag76] Michael E Fagan Design and code inspections to reduce errors in program development *IBM Systems Journal*, 15(3) 182–211, 1976

[Fag86] Michael E Fagan Advances in software inspections *IEEE Transactions on Software Engineering*, SE-12(7) 744–751, July 1986

[Fow86] Priscilla J Fowler In-process inspections of workproducts at AT&T *AT&T Technical Journal*, pages 102–112, March/April 1986

[Gil88] Tom Gilb *Principles of Software Engineering Management*, chapter 12 Addison-Wesley, 1988

[GKN92] David Garlan, Gail E Kaiser, and David Notkin Using tool abstraction to compose systems *IEEE Computer*, 25(6) 30–38, June 1992

[GTE89] The GTE Software Quality Engineering Workshop. *Guideline for Effective Software Engineering Inspections and Review Meetings*, October 1989.

[Hag89] James A. Hager. Software cost reduction methods in practice. *IEEE Transactions on Software Engineering*, 15(12):1638–1644, December 1989.

[Hof89] Daniel Hoffman. Practical interface specification. *Software—Practice and Experience*, 19(2):127–148, February 1989.

[Hof90a] Daniel Hoffman. On criteria for module interfaces. *IEEE Transactions on Software Engineering*, 16(5):537–542, May 1990.

[Hof90b] Daniel Hoffman. Software inspection, verification and testing: A hybrid approach. University of Victoria, Department of Computer Science, November 1990.

[HS92] Daniel Hoffman and Paul Strooper. Software design and verification. manuscript in preparation, September 1992.

[Hum89] Watts S. Humphrey. *Managing the Software Process*. Addison-Wesley, 1989.

[Jon91] Capers Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, 1991.

[KBM64] David R. Krathwohl, Benjamin S. Bloom, and Bertram B. Masia. *Taxonomy of Educational Objectives: The Classification of Educational Goals, Handbook 2: Affective Domain*. David McKay Company, New York, 1964.

[KM91] John C. Knight and E. Ann Myers. Phased inspections and their implementation. *Software Engineering Notes*, 16(3):29–35, July 1991.

[Kno86] Alan B. Knox. *Helping Adults Learn: A Guide to Planning, Implementing and Conducting Programs*. Jossey-Bass, San Francisco, 1986.

[KSH92] John C. Kelly, Joseph S. Sherif, and Jonathon Hops. An analysis of defect densities found during software inspections. *Journal of Systems and Software*, 17(2):111–117, February 1992.

[Lea83] J. F. Leathrum. *Foundations of Software Design*. Reston, Reston, VA, 1983.

[LMW79] Richard C. Linger, Harlan D. Mills, and Bernard I. Witt. *Structured Programming: Theory and Practice*. Addison-Wesley, 1979.

[LWM+92] Linda Landis, Sharon Waligora, Frank McGarry, Rose Pajerski, Mike Stark, Kevin Orlin Johnson, and Donna Cover. Recommended approach to software development, revision 3. Technical Report SEL-81-305, National Aeronautics and Space Administration, Software Engineering Branch, Code 552, Goddard Space Flight Center, Greenbelt, MD, 20771, June 1992.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(12):308–320, December 1976.

[Mey92a] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[Mey92b] Bertrand Meyer. Ensuring semantic integrity of reusable objects (panel). In *OOPSLA'92 Conference Proceedings*, pages 301–302. ACM Press, 1992.

[MJHS90] R. G. Mays, C. L. Jones, G. J. Holloway, and D. P. Studinski. Experiences with defect prevention. *IBM Systems Journal*, 29(1):4–32, 1990.

[MK89] John C. Munson and Taghi M. Khoshgoftaar. The dimensionality of program complexity. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 245–253. IEEE Computer Society Press, 1989.

[NK91] Takeshi Nakajo and Hitoshi Kume. A case history analysis of software error cause-effect relationships. *IEEE Transactions on Software Engineering*, 17(8):830–837, August 1991.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Par79] David L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, March 1979.

[PC86] David L. Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, February 1986.

[PCW84] David L. Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. In *Proceedings of the Seventh International Conference on Software Engineering*, pages 408–417. IEEE Computer Society Press, 1984.

[PCW89] David L. Parnas, Paul C. Clements, and David M. Weiss. Enhancing reusability with information hiding. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability. Volume 1 Concepts and Models*, chapter 6. ACM Press, New York, 1989.

[Pfl89] Shari Lawrence Pfleeger. Recommendations for an initial set of software metrics. Technical Report CTC-TR-89-017, Contel Technology Center, 15000 Conference Center Drive, PO Box 10814, Chantilly, VA 22021-3803, December 1989.

[PJ80] Meilir Page-Jones. *The Practical Guide to Structured Systems Design*. Yourdon, New York, 1980.

[Pre87] Roger S. Pressman. *Software Engineering. A Practitioner's Approach*. McGraw-Hill, second edition, 1987.

[PSS81] Alan Perlis, Frederick Sayward, and Mary Shaw, editors. *Software Metrics. An Analysis and Evaluation*. MIT Press, Cambridge, MA, 1981.

[PW85] David L. Parnas and David M. Weiss. Active design reviews: Principles and practices. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 132–136. IEEE Computer Society Press, 1985.

[RC89] Sridhar A. Raghavan and Donald R. Chand. Diffusing software engineering methods. *IEEE Software*, pages 81–90, July 1989.

[RR85] Samuel T. Redwine, Jr. and William E. Riddle. Software technology maturation. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 189–200. IEEE Computer Society Press, 1985.

[Rus91] Glen W. Russell. Experience with inspection in ultralarge-scale developments. *IEEE Software*, 8(1):25–31, January 1991.

[SP90] Andrew P. Sage and James D. Palmer. *Software Systems Engineering*. Wiley, 1990.

[TA86] Michael L. Tushman and Philip Anderson. Technological discontinuities and organizational environments. *Administrative Science Quarterly*, 31(3):439–465, 1986.

[Win90] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.

[You79] Edward Yourdon. *Managing the Structured Techniques*. Prentice-Hall, second edition, 1979.

[You89] Edward Yourdon. *Structured Walkthroughs*. Prentice-Hall, fourth edition, 1989.

[You92] Edward Yourdon. *Decline and Fall of the American Programmer*. Prentice-Hall, 1992.

Appendix A

Project proposal

Module Design and Inspections Pilot Project Proposal

Ann Jackson
University of Victoria

March 4, 1992

1 Overview

This document proposes a project for inclusion in the MPR Teltech Ltd 1992 Software Development Process Strategic Technology Program (STP) It is the outcome of a meeting[1] at which a previous proposal [1] was discussed

The overall objective of the Module Design and Inspections Pilot Project is to improve development team effectiveness in the project design phase (immediately following the Technical Proposal) The primary goal underlying this objective is to alleviate the excessive workload typically experienced by technical leaders, by enabling delegation of some reviewing tasks without loss of quality A secondary goal is to improve the quality of work and the productivity of designers

These goals will be attacked via a pilot project applied to current work on the Digital Networks Support System (DSS) product The project focuses on module designs because this is the predominant area of design effort Some attention to subprogram designs may be incorporated if relevant to current DSS work The project has the following key components

1. **Documentation of the module prologue standard**
   The module prologue is a key delegation tool for technical leaders The format and contents of this document are well understood by experienced designers but are not adequately documented Specification of a systematic standard for module design documentation is a prerequisite for efficient training and delegation of module design and reviewing tasks

2. **Training and application of inspection to module designs**
   Inspection is a peer review technique that focuses on defect detection Its reported effectiveness in industrial settings such as Bell-Northern Research [2] recommends it for trial at MPR Inspection is especially relevant to the primary goal of the pilot project because it provides an excellent forum for training participants in systematic reviewing skills

[1] Thursday February 27th, 10 a m at MPR Teltech Ltd , attended by D Reid and W Waung (MPR), D Hoffman and A Jackson (UVic)

3 **Collection and analysis of defect data**
Prior DSS defect history will be analyzed to provide a basis of comparison with defects found via inspection. The defect detection results of the pilot project are not expected to be statistically significant, but will provide quantitative input to MPR's evaluation of the pilot project and a basis for ongoing collection of defect statistics and future work in defect causal analysis.

2 Project Outline

The pilot project will be conducted by A. Jackson under the direction of D. Reid, with technical advice from Dr. D. Hoffman (University of Victoria).

2.1 Module Prologues

1 **Collect and study existing documents**
Documentation and examples of module and subprogram prologues from the Special Service Networks (SSN) department will be provided by D. Reid. Similar documentation from other MPR divisions and departments will be provided by W. Waung.

2 **Interview key users**
Experienced users of prologue design documents will be identified by D. Reid and interviewed to obtain their views on strengths, weaknesses and usage of prologues. Between two and four such interviews are anticipated.

3 **Document proposed standard**
The proposed standard will be based on the material and interview results obtained from MPR and work product criteria developed in previous work by D. Hoffman. It will define design criteria, standard document sections and notations to be used.

4 **Review and revision**
The standard document will be reviewed by MPR and revised as required for acceptance.

2.2 Defect History Analysis

The DSS defect history documented via ITRS[2] and DTRS[3] will be analyzed in order to categorize defects by the development phase in which they *occurred* (e.g.,

[2]Internal Trouble Reporting System: defects detected during internal testing.

[3]DSS Trouble Reporting System: defects detected during customer site testing and production use of the product.

technical proposal, design, implementation). Design phase defects will be analyzed to identify characteristic DSS design phase problems.

2.3 Design Inspection Criteria

The inspection criteria will be based on the prologue standard document, related MPR work products and the identified characteristic design phase problems. They will be documented via an inspection checklist and possibly an inspection preparation guide.

2.4 Training

Three or more DSS team members will be trained in the application of the prologue standard and in inspection techniques. Training materials will consist of the previously developed documents, model examples of excellent module prologues and inspection procedures and forms.

A stated MPR objective is to minimize training time not directly applied to actual current work. This is a realistic objective because the proposed techniques are intentionally straightforward and practical. Careful advance preparation of the necessary materials by A. Jackson will be important to minimize "off-task" time for MPR employees.

2.5 Application

Weekly inspection meetings will be conducted on current designs. Inspection teams will comprise three employees. There may be up to three inspection teams, according to the number of project participants assigned by MPR.

3 Schedule

Weeks 1–4. Preparation

1. Project initiation
   (a) Proposal acceptance
   (b) Non-disclosure agreement
   (c) Service contract
2. Module prologue documentation
3. Defect history analysis
4. Inspection criteria

Weeks 5–8 Training

Four weekly four-hour sessions (preferably of two two-hour segments, morning and afternoon), the first two sessions to involve a single MPR employee, for the purpose of testing and refining the training materials. The remaining two sessions will involve two or more additional members of the DSS development team. Participation of the original trainee in the group sessions may not be necessary.

Weeks 9–16 Application

Weekly inspection meetings on current work. Participants will be required to spend approximately four hours weekly on the pilot project (two hours preparation, two hours in the inspection meeting). Follow-up time will also be required of authors and moderators, to ensure that defects are corrected appropriately. This effort should be comparable to the normal expectations for review follow-up by authors and the technical leader.

Week 17+ Evaluation

A project report will be delivered in mid-July. If MPR's evaluation of the pilot project is favourable, further collaboration may be entertained.

4 Results

1. **A basis for project evaluation**
   Previous and current defect counts and categorizations, records of time invested, the project report and participant opinion.

2. **Documented standards and procedures**
   The module prologue standard, inspection forms and checklists.

3. **Trained staff**
   Several DSS designers trained as inspectors. Their skills may be applied in further use of inspections and/or individual review assignments.

4. **Training materials**
   Curriculum outline, model examples and exercises.

5. **Opportunities for further work**
   This project provides an excellent basis for further work on the Software Development Process STP, independently or in collaboration with the University of Victoria. Of particular interest are:

   (a) Application of the inspection technique to other work products such as requirements specifications, code and test plans.

(b) Defect prevention initiatives causal analysis of defects and action to eliminate the identified causes from the software development process

5 On Inspection

The following paragraphs are reproduced from the previous proposal [1] for the benefit of readers who have not seen that document

Inspection is a disciplined peer review technique that is carried out by a team of at least three individuals The participants are required to prepare by advance study of the material to be inspected and to fulfill specified roles in the inspection meeting (moderator, reader and inspector) Preparation time, the rate at which the material is inspected and the required outputs of the process are all prescribed and actual results logged The focus of the meeting is on defect detection—design issues, defect solutions and the like are not discussed Inspection therefore differs from the type of review currently performed at MPR in that it is executed by a small team rather than an individual and that it has a more prescribed focus and outcome It differs from other forms of group review and walkthroughs in that it involves individual preparation, controlled use of group time, defined roles and a paraphrasing approach to the subject Walkthroughs, for example, frequently involve unproductive discussion and simulated execution of test scenarios, which can be more cheaply and reliably accomplished by testing Experience reports suggest that all of the distinctive characteristics of inspection are important to its superior rate of fault detection

Introduction of inspection in other companies has resulted in products being shipped more cheaply, with fewer defects BNR's published findings [2] are that code inspections find up to 80% of product defects at the rate of one fault per person-hour invested (including all inspection-related activities), this being two to four times cheaper than fault detection by testing These results are confirmed or bettered in other studies It should be noted that inspection does not replace testing although it does reduce testing cost Inspection and testing have complementary strengths and weaknesses, quality assurance requires both

Inspection has important benefits for the development team Designers learn about different parts of the system and are thereby equipped for new assignments The criteria for good design and coding become more apparent and the quality of individual work improves The inspection process is a safe, disciplined forum in which people learn to function as an effective team All these benefits seem to lead to increased confidence and satisfaction on the part of designers morale improvement is consistently reported in the literature

This project can help lay the foundation of ongoing software development process improvement The inspection technique is applicable to all the concrete intellectual

products of a development process, including specifications, designs, code and test plans As software technology progresses our work products will change, but the need for peer review will remain This phenomenon is well illustrated by the hardware experience the inspection technique was originally developed for hardware and was used for low-level checking that was subsequently automated However hardware inspection has recently been rediscovered (at BNR and elsewhere) for attacking the kinds of errors that occur in present-day hardware design A second way in which inspection provides a basis for further improvement is in providing defect data which may be analyzed to identify areas requiring attention

6 References

[1] A Jackson Code Inspection Pilot Project Proposal

[2] G W Russell Experience with Inspection in Ultralarge-Scale Developments *IEEE Software* January 1991, pp 25-31

Appendix B

Defect study report

The following report has been abridged by elimination of the majority of the data tables. The omitted information is specific to MPR Teltech Ltd. and did not contribute to the thesis research.

DTRS/ITRS Analysis Report

Ann Jackson
University of Victoria

April 21, 1992

1 Introduction

DTRS and ITRS reports against DSS releases 3 2 1 and 3 2 2 to March 31, 1992 were categorized by defect detection phase, occurrence phase, type, severity, correction cost and modules involved. A few reports that were "non-problems" or unaddressed and unclear as to nature of the problem were discarded, leaving 104 reports against 3 2 1 and 81 reports against 3 2 2.

2 Findings

1. It is not possible to estimate phase containment of defects from DTRS/ITRS data, since the reporting system is not used for problems contained by phases prior to Feature Test.

2. About half the problems reported appeared to have originated in a prior release, i.e. the problems were present in the prior DSS release (3 2 0 9 for 3 2 1, 3 2 1 for 3 2 2). Current-release and prior-release problems are tabulated separately in the appendices. *Release* containment of problems is therefore imperfect.

3. Although these were not specific categorization items, it was observed that the vast majority of problems had one or more of the following causes:

   (a) Unanticipated Cases

   Only infrequently does a problem analysis identify that program logic is simply wrong i.e. does not do what is was intended to do. The majority of problem reports appear to result in the addition of new code to handle cases that were not anticipated when the segment or module was designed or modified. The typical problem report identifies that something doesn't work as a user might expect when there's a DS0A intermediate hop in the middle of a predominantly DS0B path, or when a map is blocked on a force-out but its schedule says it should be in service, or when a user changes the order but not the addresses of X.25 links at a node, or when

multiple users try to update the same database object simultaneously, or when a parameterless operation is delegated to the EC task, or when

(b) Uninitialized Data

Particularly prevalent with fields of database records and global data, some instances with local variables also

(c) Misuse of Global Data

Typically initialization problems, often in scenarios such as the following A problem with the `DEFINE THING` operation is resolved by creating a new module variable which is set in one segment and referenced in another Subsequently, a problem is reported against `REMOVE THING` or `VIEW THING` etc The new problem turns out to arise because the first segment is not called for these operations, but the second segment is Hence the second segment now references uninitialized data

4 Some subprograms and modules are trouble-spots Even if there was a lot of work on NET in these releases, the large number of prior release problems in that subprogram suggests an ongoing problem area [1]

3 Recommendations

1 Increase attention to requirements specification, whether as part of the formal Technical Proposal or subsequently Many "unanticipated case" problems appear likely to be due to designers' not being aware of detailed requirements

2 Use case analysis in module and segment design

3 Require documentation of assumptions in module design documents

4 Look for invalid assumptions during inspections and reviews

5 Look for initialization problems during inspections and reviews

6 Discourage use of global data

7 Allow static data in segments, so that module data declaration files are not cluttered with private data for individual segments

8 Where global data is referenced, ensure that it is appropriately initialized on every path leading to the reference

[1] This is no surprise of course, but it seems worth noting that the statistics confirm popular mythology

4 Current-Release Defects

The following tables categorize defects that were originated and detected within the same DSS major release (3 2 1 or 3 2 2)

4.1 Occurrence Phase

Occurred In	3 2 1	3 2 2	Total
Requirements Spec	6	4	10
Human Interface Spec		1	1
Design	41	18	59
Code	6	7	13
Problem Resolution	3	7	10
Configuration Mgmt		3	3
Feature Test	2		2
System Test	1	2	3
Field Trial	1		1

4.2 Defect Type

Defect Type	3 2 1	3 2 2	Total
Changed	1		1
Extra		1	1
Misunderstood	5		5
New	3	4	7
Omitted	24	14	38
Side-effect	2	2	4
Wrong	25	21	46

5 Prior-Release Defects

The following tables categorize defects that were detected during DSS 3 2 1 or 3 2 2 activities, but that originated in some prior release, i e the defect was present prior to 3 2 1 or 3 2 2 development respectively

5.1 Occurrence Phase

Occurred In	3 2 1	3 2 2	Total
Requirements Spec	3		3
Design	34	19	53
Code	7	20	27

5.2 Defect Type

Defect Type	3 2 1	3 2 2	Total
New	3	2	5
Omitted	19	14	33
Wrong	22	23	45

Appendix C

Module interface specification standard

The following document uses the MPR Teltech terms *Module Access Specification* rather than *Module Interface Specification*, and *segment*, meaning subroutine or function

Module Access Specification Standard Version 4

Ann Jackson
University of Victoria

November 6, 1992

This document describes the MPR Teltech Ltd Department B61 standard for the Module Access Specification work product

1 Audience

Designers, reviewers and implementors of this module, designers and implementors of other modules, maintenance designers and implementors

2 Prerequisites

1. Technical Proposal
2. Subprogram Design (ie original design activities only)

3 Scope

Describes the module interface i e everything that

1. callers may expect about module behaviour and
2. the module internal designer and implementor may assume about calls

4 Objectives

Specify the module

1. interface synopsis
2. conditions maintained
3. implementor assumptions

4 exceptions checked and how reported

5 segment effects and outputs

so as to enable correct

1 implementation

2 maintenance

3 use of the module by callers, without reference to the implementation

5 Contents

Titles of optional sections and subsections are bracketed e g **[State]** Such sections may be omitted only when inapplicable

1 **Name**
Short and long forms of the module and access (externally visible) segment names

2 **Synopsis**
Function headers for all access segments, in logical order Parameters are identified as inputs and/or outputs

3 **Description**
Succinct description of the overall module service

4 **[State]**
Description of the module's dependence on the past, typically characterized as abstract data

This section has subsections

(a) **[Initial Conditions]**
Properties of the state that are assumed to hold before the first call to any access segment

(b) **[Invariant Properties]**
Properties of the state that are maintained by all access segments

5 **Semantics**
For each access segment, describe

(a) **[Overview]**
The overall purpose and function of the segment. Second person active voice sentences are preferred. Omitted when the Effects are straightforward enough that the segment name is adequate introduction.

(b) **[Inputs]**
The expected content and purpose of each input parameter. Omitted when this is plain from the parameter names and types.

(c) **[Assumptions]**
Conditions that the implementor assumes hold on entry to the segment. Assumptions are not checked in the implementation and must be guaranteed by the caller.

(d) **[Exceptions]**
Conditions detected by the implementation that prevent normal functioning of the segment. The effect of each exception on the module state and external entities (e.g. error messages logged) is specified.

(e) **Effects**
Effects of a valid call (all assumptions satisfied, no exceptions raised) to the segment on the specified module state and on entities external to the module (e.g. records written to files, IPC messages posted).

(f) **[Outputs]**
The segment return and output parameter value(s), tabulated by exit condition.

6 **Usage**
Descriptions of required module usage scenarios.

7 **Applications**
Table of dependencies between application features and access segments.

8 **[See Also]**
Cross-reference list of related modules.

9 **[Diagnostics]**
Invocation and outputs of diagnostic operations.

Tables and pseudo-code are recommended formats for Semantics and Usage descriptions.

Appendix D

Module interface specification: spath

This appendix contains the complete Module Interface Specification excerpted in Figures 3.1–3.5.

NAME

spath, sp_initialize, sp_add_node, sp_delete_node, sp_add_edge, sp_delete_edge, sp_query_path, sp_get_path_length, sp_get_first_edge — shortest-path directed graph module

SYNOPSIS

```
#include "spath.h"

void    sp_initialize();

bool    sp_add_node(
    int     node);  /*i*/

bool    sp_delete_node(
    int     node);  /*i*/

bool    sp_add_edge(
    int     src,    /*i*/
    int     dst,    /*i*/
    int     len);   /*i*/

bool    sp_delete_edge(
    int     src,    /*i*/
    int     dst);   /*i*/

bool    sp_query_path(
    int     src,    /*i*/
    int     dst);   /*i*/

int     sp_get_path_length(
    int     src,    /*i*/
    int     dst);   /*i*/

int     sp_get_first_edge(
    int     src,    /*i*/
    int     dst,    /*i*/
    int *   node);  /*o*/
```

DESCRIPTION

Module **spath** provides access to a weighted, directed graph with integer-valued node identifiers and edge lengths. Spath allows the user to add and delete nodes and edges in the graph and to examine its paths.

STATE

set of integer *nodes*

set of record {
integer *src*, *dst*, *len*
} *edges*

Invariant Properties

for each edge *e* in *edges*
e src and *e dst* are in *nodes*

SEMANTICS

sp_initialize()

overview
Initialize the graph to have no nodes or edges

effects *nodes* = *edges* = {}

sp_add_node(*nod*)

overview
Add a node to the graph

assumptions
sp_initialize has been called

exceptions
range *nod* not in the range 0 **SP_MAXNODES**-1
An alarm message is logged

effects *nodes* = *nodes* + {*nod*}

outputs

entry condition	return value
nod in range	TRUE
range exception	FALSE

sp_delete_node(*nod*)

overview
Delete a node from the graph

assumptions
sp_initialize has been called

effects *nodes* = *nodes* - {*nod*}
FOR each edge *e*
IF *e src* == *nod* OR *e dst* == *nod*
THEN *edges* = *edges* - {*e*}

outputs

entry condition	return value
nod in *nodes*	TRUE
nod not in *nodes*	FALSE

sp_add_edge(*src, dst, len*)

overview

Add an edge to the graph

assumptions

sp_initialize has been called

exceptions

invalid node

src not in *nodes* OR *dst* not in *nodes*

duplicate edge

(exists *anylen*)(<*src, dst, anylen*> in *edges*)

invalid length

len not in range 0 **SP_MAXLEN**

An alarm message is logged

effects *edges* = *edges* + {<*src, dst, len*>}

outputs

entry condition	return value
no exception	TRUE
exception	FALSE

sp_delete_edge(*src, dst*)

overview

Delete an edge from the graph

assumptions

sp_initialize has been called

effects FOR each edge *e*

IF *e src* == *src* AND *e dst* == *dst*

THEN *edges* = *edges* - {*e*}

outputs

entry condition	return value
(exists *len*)(<*src, dst, len*> in *edges*)	TRUE
otherwise	FALSE

sp_query_path(*src, dst*)

overview

Check existence of a path

assumptions

sp_initialize has been called

outputs

TRUE iff there is a path in *edges* from *src* to *dst*

sp_get_path_length(*src*, *dst*)

overview

Get path length

assumptions

sp_initialize has been called

exceptions

no path

not **sp_query_path**(*src*, *dst*)
An alarm message is logged

outputs

entry condition	return value
path(s) exist	length of shortest path from *src* to *dst*
no path	-1

sp_get_first_edge(*src*, *dst*, *nod*)

overview

Get first edge on a path

assumptions

sp_initialize has been called

exceptions

no path

not **sp_query_path**(*src*, *dst*)
An alarm message is logged

outputs

condition	*nod*	return value
path(s) exist	node at end of first edge on shortest path from *src* to *dst*	*len* such that <*src*, *nod*, *len*> in *edges*
no path		-1

If there are two or more shortest paths from *src* to *dst* then the first edge on one such path is returned. For single-edge paths, the single edge is returned.

USAGE

1 Graph with 5 nodes and 4 edges is created, a path is followed, intermediate node is deleted, a longer path is found

call	exception	*nodes*	*edges*	outputs
sp_initialize()		{}	{}	
sp_add_node(0)		{0}	{}	TRUE
sp_add_node(1 4)		{0,1,2,3,4}	{}	TRUE
sp_add_edge(0,1,1)		{0,1,2,3,4}	{<0,1,1>}	TRUE
sp_add_edge(1,4,1)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>}	TRUE
sp_add_edge(1,4,5)	dup edge	{0,1,2,3,4}	{<0,1,1>,<1,4,1>}	FALSE
sp_add_edge(0,2 2)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>, <0,2,2>}	TRUE
sp_add_edge(2,4,1)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>, <0,2,2>,<2,4,1>}	TRUE
sp_query_path(0 4)		{0,1 2,3,4}	{<0,1,1>,<1,4,1>, <0,2,2>,<2,4,1>}	TRUE
sp_get_path_length(0,4)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>, <0,2,2>,<2,4,1>}	2
sp_get_first_edge(0,4)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>, <0,2,2>,<2,4,1>}	*nod* = 1, 1
sp_get_first_edge(1,4)		{0,1,2,3,4}	{<0,1,1>,<1,4,1>, <0,2,2>,<2,4,1>}	*nod* = 4, 1
sp_delete_node(1)		{0,2,3,4}	{<0,2,2>,<2,4,1>}	TRUE
sp_get_path_length(0,4)		{0,2,3,4}	{<0,2,2>,<2,4,1>}	3
sp_get_first_edge(0,4)		{0,2,3,4}	{<0,2,2>,<2,4,1>}	*nod* = 2, 2

2 Print all nodes on the shortest path from x to y

```
print x
z = x
DO n = sp_first_edge(z,y,&z), print z
UNTIL z == y
```

DIAGNOSTICS

void **sp_dump**(),

Prints a dump of *nodes* and *edges*.

Appendix E

Abstract data types

The following document defines the abstract data types used to specify module abstract State

Module Design and Inspections Abstract Data Types

May 13, 1992

1 Elementary Types

1.1 Boolean

Usage boolean *variable-list*

example boolean *a, b, c*

Constants **TRUE** and **FALSE**

Operations NOT AND OR == !=

1.2 Character

Usage char *variable-list*

example char *a, b, c*

Constants C-style ASCII character constants

examples 'a', '\n'

Operations == != < <= > >=

1.3 Integer

Usage integer *variable-list*

example integer *a, b, c*

Constants integers

examples -52, 9706

Operations == != < <= > >= + - * /

1.4 Real

Usage real *variable-list*

example real *a, b, c*

Constants real numbers

examples 1.5, -9706

Operations == != < <= > >= + - * /

1.5 String

Usage string *variable-list*

example string *a, b, c*

Constants C-style ASCII string constants

example "The cow jumped over the moon.\n"

Operations == != < <= > >= + (catenation) [(element or substring)

examples "abcdefg" <= "abchhh"
"Jack" + " and " + "Jill" (is "Jack and Jill")
"Jack"[1] (is 'a'), "Jack"[0..2] (is "Jac")

2 Type Constructors

2.1 Record

Usage record {*field-list*} *variable-list*

example

```
record {
  integer id
  string lastname, firstname
} rec1, rec2
```

Constants ordered list of constants of each field type, enclosed in angle brackets

example <33, "Smith", "Mary">

Operations == != . (field reference)

examples *rec1 == rec2, rec1.firstname*

2 2 Sequence

Usage sequence [*low high*] of *type variable-list*
If the index range is omitted then [0 ?] is assumed

examples sequence of integer *s, s1, s2*
sequence ['a' 'z'] of boolean flag

Constants constants of the base type, enclosed in brackets

examples [] empty sequence
[3, 5, 12 14, -10] sequence of integer ("12 14" is equivalent to "12, 13, 14")
[{"john", "robert"}, {}, {"alpha"}] sequence of set of string

Operations == != + (catenation) [(element or subrange)
in (member test) | | (length)

examples *element* in *s*, *s1* + *s2*, |*s*|, *s*[0], *s*[5 10]

Notes Sequences are ordered and may contain duplicates, hence

$$[1,2,5,10] \neq [1,5,2,10] \text{ and } [1,2,5,10] \neq [1,2,2,5,10]$$

These properties make *sequence* conceptually more complex than *set* Unless the notion of ordering is essential to module semantics, *set* should be preferred over *sequence* for describing module states

2.3 Set

Usage set of *type variable-list*

example set of integer *s, s1, s2*

Constants constants of the base type, enclosed in braces

examples {} empty set
{3, 5, 12 14, -10} set of integer ("12 14" is equivalent to "12, 13, 14")
{{"john", "robert"}, {}, {"alpha"}} set of set of string
$\{x | x > 0 \text{ AND } x < 12\}$ math-style constructor for set of integer

Operations ==! = + (union) * (intersection) - (difference)
in (subset or member test) | | (number of elements)

examples *s1* != *s2*, *s1* in *s2*, *element* in *s*, *s1* - *s2*, |*s*|

Notes Sets are unordered and contain no duplicate elements, so that

$$\{1, 2, 5, 10\} == \{5, 2, 5, 10, 1, 1, 2\}$$

If ordering or duplication are essential for describing module semantics, use *sequence*

Appendix F

Training course description

Module Design and Inspections

Ann Jackson
University of Victoria

November 30, 1992

1 Introduction

This document describes the objectives and syllabus of the training component of the Module Design and Inspections Pilot Project[1] for MPR Teltech Ltd.

2 References

[1] A. Jackson, "Module Design and Inspections Pilot Project Proposal," March 1992.

[2] "Module Access Specification Standard," Version 4, 6 November 1992.

[3] "Module Internal Design Standard," Version 1, 1 May 1992.

[4] "Module Design Inspection Checklists," Version 3, 30 November 1992.

3 Objectives

On completion of this course, you will be able to do the following.

3.1 Module Design Skills

1. Explain the purpose of splitting module design into interface and internal design activities and documents.
2. Explain why the Module Access Specification is preferable to the code as a caller's guide to module behaviour.
3. Given a Module Access Specification document and a module trace (sequence of calls to the module), write the outputs of the final call and final module state.
4. Given a Module Access Specification document and a statement of a requirement using a service of the module, write correct caller code.

5 Given the document standard [2],

   (a) describe in your own words the purpose of each document section

   (b) give an example of acceptable content for each document section

6 Give two examples of abstract data types and explain why they would be preferable to implementation data types for use in a Module Access Specification

7 Given an existing DSS module, write a Module Access Specification that complies with the document standard

8 Given interface design deliverables, identify violations of inspection checklist[4] items

9 Given a Technical Proposal and Subprogram Design, design and document a module interface that complies with the standards for the deliverables

10 Justify your module interface designs with reference to completeness, comprehensibility, consistency, sufficiency and simplicity

3.2 Inspection Skills

1 Briefly describe the main ideas of the inspection technique (purpose, process and roles)

2 As inspector in an inspection meeting, express questions and concerns

   (a) clearly, so that other participants understand

   (b) in terms of software behaviours only, without discussion of

      i objections to the requirements specification

      ii solution strategies

      iii author performance

      iv unrelated topics

3 As author of an inspected work product, complete recommended follow-up activities to the satisfaction of the inspection moderator

4 As recorder in an inspection meeting,

   (a) log all required information on the designated forms

   (b) record all defects discovered so that the author and moderator can understand the items on follow-up

5 As reader in an inspection meeting, summarize a Module Access Specification so that other participants

   (a) understand the summary

   (b) find the inspection rate appropriate

6 As moderator in an inspection meeting, ensure that

   (a) all required information is logged

   (b) the inspection rate is appropriate

   (c) discussion is focussed and courteous

   (d) follow-up activities are completed

3.3 Attitudes and Evaluative Skills

1 Value effective, efficient use of group resources

2 Prefer to work from inspected inputs

3 Look at the Module Access Specification before or instead of the code

4 Schedule and perform documentation revision in step with software revisions

5 Suggest improvements to standards, checklists and procedures

6 Evaluate designs in terms of agreed key design principles

7 Propose and justify design alternatives

8 Plan and participate in future applications of module interface design and inspections

9 Plan and participate in objective evaluation of the effectiveness of inspection and other development process methods and procedures

4 Syllabus

I Module Design

A Topics

1 Project and course overview
2 Interface and internal design
3 The Module Access Specification — introduction
4 Execution tables

B Activities

1 Lecture
2 Assignment

II Module Access Specifications

A Topics

1 Interface design criteria
2 Module abstract state
3 Abstract data types
4 Inspection — introduction

B Activities

1 Discussion of previous assignment
2 Lecture
3 SEI video scenes of inspection
4 Assignment

III Inspection

A Topics

1 Process and roles
2 Forms
3 Preparation

B Activities

1 BNR inspection video
2 Discussion
3 Lecture
4 Overview of inspection material
5 Assignment — inspection preparation

IV. Inspection Practice

A. Topics

1. Reading (paraphrasing) Module Access Specifications

B. Activities

1. Lecture
2. Inspection meeting

Appendix G

Training materials

The following sample training materials are for course session II: Module Interface Specifications.

1 Lecture outline

Total time [90 minutes]

I Overview [2 min]

- A Interface Design Criteria
- B Module Abstract State
- C Abstract Data Types
- D Inspection — introduction

II Interface Design Criteria [15 min]

- A Many possibilities
- B 5 key principles
  1. Why these principles?
     - a needs identified from ITRS/DTRS survey
     - b fundamental concerns — essential, not too many
  2. Completeness
     - a ITRS/DTRS analysis indicated this is neglected
     - b Example how does it handle intra-mux and one-hop paths? Unusual path statuses?
  3. Comprehensibility
     - a Audience is the criterion
  4. Consistency
     - a Already under control at MPR
     - b Applies to naming conventions, params, exceptions etc
     - c Example dss_completion_status_enum for return values
  5. Sufficiency
     - a Basic level of "meeting" requirements
     - b Example Parameters for all node types?
  6. Similar but not the same as Completeness
  7. Simplicity
     - a Basic principle of "good" design
     - b Everything necessary but not more
     - c You know it when you see it

III Module Abstract State [20 min]

- A Dependence on the past
- B Not all modules have state (e g previous assignment)
- C Typical of objects — awkward to describe otherwise
- D Description criteria
  1. comprehensibility and simplicity
  2. *not* implementation considerations (language constructs, efficiency)
- E Example — monen abstract state and semantics

IV Abstract Data Types [15 min]

- A Simple and powerful common concepts
- B C style syntax for MPR
- C Elementary types real, integer, boolean, char, string
- D Constructors set, sequence, record
- E Compound types may be "indexed by" a field or other type
- F Constants
  1. Sets use braces {1, 2, 3}
  2. Sequences use brackets [1, 2, 3]
  3. Records use angle brackets <123, "Smith", "John">
  4. Ranges expressed as a z e g [5, 9, 20 30, 35], {1 10}
- G Operations
  1. Use in effects descriptions
  2. Standard arithmetic and relational
  3. Set operators
- H Use informally — comprehensibility is the key (auipl example)
- I Example utlpg

V Summary [2 min]

- A Interface Design Criteria
  5 key principles
- B Module Abstract State
  simple, comprehensible characterization of dependency on the past
- C Abstract Data Types
  simple, powerful, common concepts

VI Questions, comments and discussion

A Access Spec Standard, Design Tutorial, assignment problems

VII Break

VIII Inspection — Introduction [10 min]

A Disciplined peer review technique

B Purpose is fault detection

C Industrial origins and application
Successful at IBM, AT&T, BNR

D Benefits

1 Quality improvement

2 Cost reduction

a In testing and maintenance

3 Staff development

a Technical education

b Team-building, morale improvement

E Basis for future process improvement

1 Inspection technique is applicable to to other work products

2 Inspection outputs may be analyzed for defect prevention

IX SEI Video

2 Materials list

1 Abstract Data Types handout (adt.tex)

2 Module Design Inspection Checklists (checklist.tex)

3 man page examples (utlpg.1, monen.1)

4 Inspection Process Description handout (from GTESEQW)

5 Assignment (ex2.mm)

6 SEI videotape and notes

3 Assignment

1. Read the *Abstract Data Types* handout

2. Using the **utlpg** Module Access Specification, write pseudo-code for the following function

   > Given a digraph in **utlpg** and two nodes a and z that are in the graph, find all edges that are on every path from a to z

3. Write a Module Access Specification for one of the following DSS modules. If you are short of time, ensure you complete at least the STATE and normal segment effects portion of SEMANTICS.

Module and Access Segments	Assigned To
BILPB bilpb_execute_billing_request bilpb_initialize_billing_interface bilpb_log_billing_event bilpb_perform_operation bilpb_read_billing_attribute bilpb_select_billing_attribute	
CONLA conla_cancel_message_watch† conla_initialize_logfile_interface conla_initialize_message_classes conla_perform_operation conla_provide_hardcopy_mode†	
DSOPE dsope_initialize_interface dsope_interrupt_operations dsope_receive_operation_status dsope_receive_parameter dsope_request_operation	

†Semantics for this segment may be omitted

   For help with formatting, see man(7) and existing DSS examples.

4. Read the Inspection Process Description (from the GTE Software Engineering Quality Workshop, 16 October 1989)

5. Read the *Module Design Inspection Checklists*, sections 1 through 3.

Appendix H

Industrial specification

The following Module Interface Specification was written by an employee of MPR Teltech Ltd. and inspected during the pilot project.

NAME

difpx, dif_provide_database_export,
difpxip, difpx_initialize_producer,
difpxot, difpx_open_transaction,
difpxqt, difpx_queue_transaction,
difpxic, difpx_initialize_consumer,
difpxgt, difpx_get_next_transaction,
difpxct, difpx_close_transaction — DSS export transaction services

SYNOPSIS

```
void                            difpx_initialize_producer(
    char *                          directory),   /*i*/

dss_completion_status_enum      difpx_open_transaction(
    FILE **                         file),        /*o*/

dss_completion_status_enum      difpx_queue_transaction(
    bool                            error),       /*i*/

void                            difpx_initialize_consumer(
    char *                          directory),   /*i*/

dss_completion_status_enum      difpx_get_next_transaction(
    FILE **                         file,         /*o*/
    char *                          filename),    /*o*/

dss_completion_status_enum      difpx_close_transaction(
    bool                            error),       /*i*/
```

DESCRIPTION

Module **difpx** provides services which implement an ordered transaction queue for exporting data using intermediate files for storage. These services are used for exporting DSS database operations to SIS.

A process using these services may be either a *producer* or a *consumer* of transactions (but not both). There may be multiple active instances of transaction producer processes, but only one consumer may be active at a time.

The module provides two similar sets of services for producers and for consumers: one initialization service, which must be called first, and services to open and to close transactions, which must be called in pairs.

The producer services implement transaction creation and are used by DSS tasks to export the database transaction upon its successful completion in DSS. The services provided are INITIALIZE_PRODUCER, OPEN_TRANSACTION, and QUEUE_TRANSACTION.

The consumer services implement transaction import and are used by the DSS EXPORT task to open queued transaction files and remove them from the queue.

The services provided are INITIALIZE_CONSUMER, GET_NEXT_TRANSACTION, and CLOSE_TRANSACTION.

These services work with UNIX standard library file streams and do not impose any restrictions on the contents of the transaction files. The caller provides only a directory name upon initialization and otherwise need have no knowlege of the transaction file names.

This module also implements the DSS EXPORT task, which is not described in this Module Access Specification.

STATE

string *queue_directory*

record {
    boolean *open*
    integer *timestamp*
    string *filename*
    FILE * *descriptor*
} *transaction*

set of record {
    boolean *error*
    integer *timestamp*
    string *filename*
} *queue*

Invariant Properties

All *timestamp* values in *queue* are unique.

SEMANTICS

difpx_initialize_producer(*directory*)

overview:
    Initialize the module as a producer of transactions.

assumptions:
    This segment has not been called before.

effects: *queue_directory* = *directory*
    transaction open = FALSE

difpx_open_transaction(*file*)

overview:
    Create a new transaction.

assumptions:
    difpx_initialize_producer has already been called.

exceptions
access Unable to access file
A software error is logged, aborting the operation

effects If a transaction is not already open
A new temporary UNIX file is opened for writing
The temporary filename is stored in *transaction filename*
The opened stream descriptor is stored in *transaction descriptor*
transaction open = TRUE

outputs
**file* = *transaction descriptor*

exit condition	return value
Successful	dss_success_v
This process already has an open transaction	dss_duplicate_name_v
The transaction file could not be created	dss_failed_v

difpx_queue_transaction(*error*)

overview
Add the currently open transaction to the queue

assumptions
difpx_initialize_producer has already been called

effects A unique timestamp is assigned to *transaction timestamp*
The currently open transaction stream descriptor, *transaction descriptor*, is closed
The file, *transaction filename*, is uniquely renamed to place it in the transaction queue directory *queue_directory*
transaction open = FALSE
queue = *queue* + <*error, transaction timestamp, transaction filename*>
A SIGUSR1 signal is sent to the DSS EXPORT task to notify it of the newly enqueued transaction

outputs

exit condition	return value
Successful	dss_success_v
This process has no open transaction	dss_no_object_by_that_name_v
The transaction file could not be enqueued	dss_failed_v

difpx_initialize_consumer(*directory*)

overview
Initialize the module as a consumer of transactions. The *queue* may already contain zero or more arbitrary elements

assumptions
This segment has not been called before
directory exists and is accessible
This process is the only transaction consumer for the specified directory

effects *queue_directory* = *directory*
queue is initialized
transaction open = FALSE

difpx_get_next_transaction(*file, filename*)

overview
Find and open the oldest transaction in the queue

assumptions
difpx_initialize_consumer has already been called

effects If *queue* is not empty and no open transactions
The element *e* is found whose *timestamp* is the smallest of any element in *queue* If found
e filename is stored in *transaction filename*
If *e error* is FALSE
e timestamp is stored in *transaction timestamp*
The file *e filename* is opened for reading
The opened stream descriptor is stored in *transaction descriptor*
transaction open = TRUE

outputs
**file* = *transaction descriptor*
filename = *transaction filename*

exit condition	return value
Successful	dss_success_v
There is no queued transaction	dss_no_object_by_that_name_v
This process already has an open transaction	dss_duplicate_name_v
The oldest queued transaction has an error flagged in it	dss_would_block_v
Could not open the oldest queued transaction	dss_failed_v

difpx_close_transaction(*error*)

assumptions
difpx_initialize_consumer has already been called

overview
Close the open transaction and either remove it from the queue or flag an error in it

effects If there is an open transaction
The currently open stream descriptor, *transaction descriptor*, is closed
The element *e* in *queue* is found such that
e timestamp = *transaction timestamp*
If *error* is TRUE
e error = TRUE
e remains in *queue*
else
If the file, *transaction filename*, exists
delete file
If file is deleted
queue = *queue* - *e*
transaction open = FALSE

outputs

entry condition	return value
Successful	dss_success_v
This process has no open transaction	dss_no_object_by_that_name_v
The transaction file could not be closed or deleted	dss_failed_v

USAGE

To place transactions into the queue

```
difpx_initialize_producer("queue");

status = difpx_open_transaction(&fp);
if (status == dss_success_v)
{
    /* write data to file fp */

    status = difpx_queue_transaction(error_flag);
}
```

To read transactions from the queue

```
(void) difpx_initialize_consumer("queue"),

for(,,)
{
    status = difpx_get_next_transaction(&fp, filename),
    if (status == dss_success_v)
    {
        /* read data from file fp */

        /* process data */

        status = difpx_close_transaction(error_flag),
        if (status != dss_success_v)
            /* log an error */
    }
    else
        /* log an error */

}
```

APPLICATIONS

Application	Operation	Calls
DSS	Export Database Transaction	difpx_initialize_producer() difpx_open_transaction() difpx_queue_transaction()
DSS_EXPORT	Read Database Transaction	difpx_initialize_consumer() difpx_get_next_transaction() difpx_close_transaction()

VITA

Surname Jackson Given Names Ann Margaret

Place of Birth Christchurch, New Zealand Date of Birth May 12, 1956

Educational Institutions Attended

University of Victoria	1991 to 1993
University of British Columbia	1973 to 1978
University of Victoria	1971 to 1973

Degrees Awarded

B Sc (Honours) University of British Columbia	1977

Honours and Awards

Martlet Chapter IODE Graduate Scholarship	1992-93
Natural Sciences and Engineering Research Council Postgraduate Scholarship	1991-92
Advanced Systems Institute Graduate Recruitment Scholarship	1991-92
President's Research Scholarship	1991-92
National Research Council Postgraduate Scholarship	1977-78
Faculty of Science Dean's Honour List	1977
Kapoor Singh Scholarship	1974-75
British Columbia Telephone Company Scholarship	1973-74
University of Victoria Faculty Association Scholarship	1973-74
Seaspan International Ltd Scholarship	1973-74
Faculty Women's Club Prize	1973-74
Francis Gold Wrist Watch	1973-74
Adeline Julienne Deloume Memorial Scholarship	1972-73
University Women's Club Scholarship	1972-73
President's Scholarship	1972-73
President's Entrance Scholarship	1971-72

Publications

Learning Assembly Language: A Guide for BASIC Programmers, with Hugo T. Jackson. Harper and Row, New York, 1985.

An Informative Computerized Laboratory Records Maintenance System, with Karel Hartman. *Journal of Chemical Education*, 54(8) 507–508, August 1977.

PARTIAL COPYRIGHT LICENSE

Title of Thesis

Introducing Software Engineering Methods into Industrial Practice: Module Interface Specification and Inspection

Author [redacted]

Ann M. Jackson

April 13, 1993