Software process improvement in a medium-sized company

Sanjay Mistry

Software Process Improvement in a Medium-sized Company.

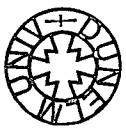
Sanjay Mistry

MSc.

University of Durham.

Computer Science.

2000.



Software Process Improvement in a Medium-sized Company.

Sanjay Mistry

Abstract

For many organisations, software is a crucial component of their business as it provides a competitive advantage over other organisations. Hence, they are very keen to ensure that the software they employ is not only reliable and defect-free, but also cost-effective to produce and maintain. That is, organisations desire the highest quality of software, but at a minimum cost and time.

Research into software engineering has shown that by improving software development or maintenance processes, there can be an improvement to the software quality. This thesis describes how software process improvement has been implemented in a medium-sized company where software is used to automate manufacturing processes. Prior to this study, there had not been any form of software process improvement in the company, although they followed company standards and procedures that have been in place for over a decade.

The aim of this study is to use software development and maintenance projects in the company as cases for process improvement initiatives. Each case provides evidence in support of a hypothesis that is associated with it. For each case study, the problem is assessed, a diagnosis has been researched and actions have been taken based on that research. Each hypothesis is evaluated at the end of the thesis followed by conclusions of the whole research.

The principle outcome of this research is that software practices of a medium-sized company can be improved using small-scale software process improvement. Using the CMM as a guide, software process initiatives were implemented to address specific areas of software engineering, i.e. maintenance, testing, planning and control, requirements management, and testing. Efforts have been made to raise the company's maturity in the CMM with respect to these areas. Collectively, the case studies achieve improvement of software practices by way of applying software process improvement in a systematic manner - in this case the IDEAL framework from the CMM.

Acknowledgements.

I would like to express my immense gratitude to the following people who have given me their support and guidance to complete this research. Dr Cornelia Boldyreff, my academic supervisor, whose patience, advice and guidance has seen me through this research to the very end. Gary Brown, my industrial supervisor and I.T. Manager at Philips Components Washington, the man who had granted me the opportunity to undertake this research and who has constantly supported me throughout the two years.

I must also express my appreciation for support given to me by all my friends in the I.T. department, especially to Ross Mackenzie who has been my 'technical guru' and good friend throughout this research. Without his support, advice, guidance and expertise I would have not been able to complete the projects that I have worked on.

Finally, I must say a big thanks to my mother, father and sister for supporting me and giving me the motivation to work to the very end. I dedicate this thesis to them.

Contents

List of Table and Illustrations.

Statement of Copyright.

Chapter 1.

Introduction.

1.1 Introduction.

More and more organisations are becoming increasingly reliant on software in order to gain some business advantage over competitors. However, this advantage comes at a considerable cost, as software development is expensive, especially for large and complex systems.

Brooks[Brooks, 1995] has identified that software development is difficult because of its inherent property of complexity. This complexity means that products made from software cannot be written immediately. Instead, a development process must be undertaken where software is constructed through various stages starting from high-level list of requirements; followed by software design to satisfy the requirements; then the implementation of design; then testing of implementation to validate requirements; and finally, the acceptance by the user. This is a classic approach of software development and it is not dissimilar to other forms of development used in building construction or car manufacturing.

In software development, these stages are necessary to ensure that the specified requirements are realised into a complete software solution within budget and time. This means reducing the complexity of the software at early stages, raising the quality of the software at early stages, and preventing changes to the original specification being made at later stages in the development process.

It is strongly believed in the Engineering discipline that by improving the process of development, the quality of the final product will be raised. The classic approach of development (requirements, design and implementation) provides a very high level framework for creation of software. It is a generic model of *what* must be done, but the challenge for many practitioners is to understand *how* these activities should be

undertaken to make a software product effectively and productively, whilst maintaining the highest quality of the final product.

Over the last three decades, researchers have built on the foundations laid by the classic approach of development. There have been attempts to define *processes* that are prescriptions of how the stages of the classic model should be undertaken. These software processes have evolved because everybody attempts to design a system of working that will suit their own environment and meet the needs of their individual projects. In doing so they have furthered the work of the people before them and have incrementally progressed towards better software processes. This incremental progression to better software processes is known as Software Process Improvement.

Many organisations and institutions around the world are recognising the need to continue to improve software processes. By sharing this knowledge, businesses are gaining competitive advantage by focusing on tuning their software development and maintenance processes to achieving their business goals. Those that do not undertake process improvement are in danger of becoming less competitive by continuing to use their traditional, out-dated and unproductive software development processes.

This thesis discusses a project that aims to improve the software practice in a medium-sized company that uses software to run a competitive business. This is the basis for incremental software process improvement initiatives on different projects within the company. Each initiative will attempt to assess a problem area and then implement a software process improvement by researching a solution. In general, the research will attempt to raise the companies software practices based on the outcome of the initiatives.

1.2 The criteria for success.

The main objective of this research is to apply software process improvement theory and to evaluate their effects on the software practices of a company. The criteria for success are as follows:

a) to investigate the areas associated with software process improvement in order to gain background knowledge that will support initiatives.
b) to identify where the company will need to improve its processes in order to formulate hypotheses regarding the effects of software process improvement.
c) To undertake case studies in order to prove or disprove the hypotheses.

Each case study will be described in this thesis. They are based around the systems development projects that have been undertaken at the time of this research. Each case study is centred around one initiative that is carried out in consistent manner based on the IDEAL approach described in chapter 3.

The hypotheses are as follows:

- *Use of source code analysis tools will facilitate the process of re-engineering of existing software systems.*
- *A formal testing process can contribute to the production of more reliable software.*
- *Software process modelling will help to facilitate a more predictable or manageable systems development project.*
- *Introduction of new requirements process will allow developers to understand and satisfy user requirements for new systems.*

1.3 Outline of the thesis.

Chapter 2 sets the context of the research and provides background to the company involved. It discusses the purpose of the software process improvement initiatives with respect to the medium-sized company and explains why the company has taken action to improve its software practices.

Chapter 3 explains why software is difficult to work with and why software processes are necessary in industry. A brief insight into the history of software processes is also given. The chapter also discusses software process improvement providing two

examples of emerging standards that encourage businesses to improve their software processes. Chapter 3 also presents the opportunities that exist for software process improvement in the company.

Chapters 4 to 7 each discuss a case study for a software process improvement initiative. A standard format is used to explain each case study that is based on the principles of the IDEAL approach, which is discussed in chapter 3. Chapter 8 reviews each case study and presents the results. Then finally, chapter 9 draws a conclusion on the whole thesis and comments on further work.

Chapter 2

Background.

2.1 Introduction.

Emerging software process improvement standards are encouraging industry to undertake assessment of their processes as a drive to improve and achieve better business performance. Many companies whose software practices are far behind the more modern practices have long ignored this encouragement. Humphrey[Humphrey, 1999] comments on current software practices by saying:

"The general practices of industrial software engineers are poor by almost any measure. Their projects are typically late and over costs, they cannot predict when they will finish, and the final products frequently have many defects. While there are exceptions and software engineers are generally dedicated and hard working, this situation has existed for many years."

For software development companies, the need to examine their own methods of developing software is essential in order to produce high quality and defect free products mainly because the software is the core ingredient of the product itself. However, the need for software process improvement in manufacturing companies may be less crucial as software is a service to the production process. Here, the emphasis for making reliable software is still strong, but issues surrounding development and maintenance practices are not at the forefront of the company's improvement goals. It is left to the software engineers themselves to incorporate 'best practice' that they have learnt from their education and their own experience. However, these practices can be outdated or non-progressive in terms of producing better software. An example of such a manufacturing company is Philips Components Washington.

The next section discusses Philips Components Washington in more detail and explains why their software development and maintenance practices have deteriorated over the last decade, but more recently have become the focus of strong attention.

2.2 Philips Components Washington (P.C.W.).

P.C.W. is a company that is part of the Philips Components group that has factories around U.K. and Europe. The company is key part of the supply chain for television set manufacturing. They manufacture Deflection Units (D.U.) that fire the electrons onto the screen of a television set to provide the picture. The deflection unit is an essential component in the television set, hence, the company is under great pressure to produce very high quality products in the supply chain.

2.2.1 Software at P.C.W.

During the late 1980's, P.C.W., invested heavily into automating their production lines in order to reduce labour costs that were considerably high compared to countries overseas, particularly the Far East. The drive was aimed at reducing the product price in order to compete with other D.U. manufacturers. Automation also provided efficiency and productivity. These automated systems were developed in ANSI-C using UNIX operating systems. As more production line controllers were developed, with great urgency, the attention to software practice was given less attention as pressure mounted to meet deadlines.

The general design of the automated system was to transport the products on carriers to automated cells that process the product individually. The products are manually loaded onto the line and after they have been through all the processes they are manually off-loaded. The major advantage of these automated systems, known as 'Line Controllers', was that they collected production and process data. This was used to control the manufacturing process as information was available regarding product yield, cell downtimes and fault data, and efficiency times.

By 1995, there were seven lines with associated Line Controllers, each capable of producing approximately 30,000 deflection units in a week. By this time, software development had dropped considerably and the company enjoyed the benefits of automated production process for products that showed little technological changes.

Hence, P.C.W. began to reduce the number of software engineers, whilst the remaining engineers maintained the line controller systems software sometimes altering the code in parts to adjust to any small changes in the production process.

2.2.2 New threats to the company.

By the late 1990s, P.C.W. began to experience major threats to its business survival, namely dramatic changes in television technology and innovation, loss in customer confidence and the Year 2000 problem. These threats had a significant impact on the company.

The market for deflection became fiercely competitive. Major technological changes began to emerge as competitors invested into widescreen television manufacturing. The direction of the business turned to providing D.U.s for new widescreen television sets. However, the complexity of widescreen D.U.s is such that production processes would be extremely difficult to automate. Hence, P.C.W. decided to revert back to manual production processes for these new lines.

The loss of customer confidence became more apparent due to faults in products and cheaper pricing from competitors. Under severe pressure to increase manufacturing performance, the I.T. department was given the task to improve the flow of production and process information within the company. A project was proposeded to develop a Manufacturing Information System that would collect and store production and process data. This data would provide trend data that can be interrogated by production or senior managers so that they can monitor and control the manufacturing process. Within this project, there were small software development projects proposed to contribute to the development of the Manufacturing Information System.

By 1998, organisations across the world had become increasingly aware of the Year 2000 problem. P.C.W. began to focus its attention on its automated production systems and made preparations for a software maintenance project to overcome the Year 2000 problem. As a result of this problem, and the change in business direction, P.C.W. was left with seven legacy systems that had to be maintained to ensure

business survival beyond December 31st, 1999. This was by no means a cheap and easy task. These software systems were large and complex which meant that they would be difficult to maintain. Furthermore, documentation for each system that was written had disappeared and the little that remained were not updated. Today, tools exist that are designed to aid software maintenance tasks, however, no investment had been made into such CASE tools at P.C.W.

All three responses to business threats forced the company to focus more attention to the software from its legacy systems and to its software practices, in general. The next section discusses how P.C.W. invested in a scheme that aimed to support the company in its response to the three major threats discussed above.

2.3. The Teaching Company Scheme (T.C.S.).

The Teaching Company Scheme (T.C.S.) is a two year project between the University of Durham and Philips Components Washington (P.C.W.). T.C.S. projects are promoted by the British Government to encourage knowledge transfer from a university into a company, where clearly defined goals are set that will improve the company's business performance in some way. Such schemes have been devised between many universities and companies across the U.K. providing both partners with benefits. For this project, P.C.W. would achieve its goals and better business performance, whilst the university learns from the application of its research in a live environment.

The Computer Science department at the University of Durham was a suitable partner in this T.C.S. project because it was the founder of R.I.S.E. (Research Institute for Software Evolution) which is dedicated to research into software development and maintenance. An academic supervisor from this institute collaborated with the industrial supervisor at P.C.W. to co-ordinate the project. The author of this thesis was responsible for gathering knowledge from the university and applying it in the company. The academic supervisor provided support for the research whilst the industrial supervisor controlled the business aspects of the project. This thesis is a product of the research that has been carried out during this T.C.S. project.

2.3.1. The purpose of the T.C.S. project at P.C.W.

For this T.C.S. project, P.C.W. had the following three aims

1. to overcome the Year 2000 problem.
2. To improve the company practices in software maintenance, software auditing and software reuse techniques; and
3. to contribute to the development of a Manufacturing Information System (M.I.S.).

The first aim related to the issue of software maintenance and P.C.W. had proposed projects for the reengineering of each production line controller system. In general, each production line controller system required Year-2000 compliant hardware and platforms to replace old hardware and platforms. Consequently, the software would have to be upgraded to comply with hardware and platform changes, but must essentially retain the same functionality. The T.C.S. project would be directly involved in the re-engineering of production line controller systems and the university pledged to support the effort by allowing the use of their software maintenance tools.

The second aim was concerned with improving the software practices in the company. It was associated with the demand in the business to upgrade the software for Year-2000 compliance project, but its objective was to provide the company with a software structure that will facilitate software changes in the future, hence, software reuse. Other issues that relate to the software engineering field could be addressed as the T.C.S. project progressed. The next section discusses how this aim was the driver for software process improvement in the company.

The third aim addressed opportunities for projects that could be undertaken during the two years that would contribute to the development of the M.I.S. system. In phase 1 of M.I.S. development, the developers aimed to create a manual data collection process. As common database platforms would be used, it was proposed for phase 2 that, where possible, this data should be fed into the M.I.S. database automatically. It was intended that the author of this thesis would be involved directly in developing software solutions for this automatic data collection as part of the T.C.S. project.

2.4. Opportunities for Software Process Improvement.

The last section explained the role of the T.C.S project at P.C.W. and mentioned that one of its aims was to improve the company's software practices. This section explains how this aim has been supported by research into software process improvement which is essentially what this thesis addresses – software process improvement in a medium-sized company.

The T.C.S. project consisted of two large sub-projects: the re-engineering of production line controller systems for Year 2000 compliance, and the automation of data collection for M.I.S. It was decided between the academic supervisor, the industrial supervisor and the author of this thesis that issues surrounding the improvement of software practices can be addressed whilst undertaking these sub-projects. Subsequently, these projects could be used as case studies for software process improvement highlighting the essential problem in practice, the software process improvement solution, its application and its evaluation.

As the T.C.S. project progressed case studies emerged and software process improvement initiatives were applied. The first project was the re-engineering of production line controller systems that provided two case studies:

- Case 1: Re-engineering of production line system software to adapt to a new database platform for Year 2000 compliance.
- Case 2: Testing production line controller software.

The second project was to automate data collection and this provided one case study:

- Case 3: Automation of data collection from winding machines.

Between the two sub-projects, another case study emerged that was not related to the T.C.S. project, but the issue it addresses would greatly improve a specific software practice in the company. The industrial supervisor wanted the author of this thesis to address the problem of new systems requirements management. Thus, the following case study was observed:

- Case 4: Investigation of requirements gathering for new I.T. systems, primarily the MIS project.

A hypothesis for each case has been made to establish whether or not the initiative successfully leads to any improvement in the software processes of P.C.W. The following hypotheses are software process improvement proposals that were aimed to contribute better software practice. They are as follows:

For case 1: *Use of source code analysis tools will facilitate the process of re-engineering of the existing software systems.*

For case 2: *A formal testing process can contribute to more reliable software.*

For case 3: *Software process modelling will help to facilitate a more predictable or manageable systems development process for project.*

For case 4: *Introduction of new requirements process will allow developers to understand and satisfy user requirements for new systems.*

2.5. The hindering factors for software process improvement at P.C.W.

Software process improvement must be driven through a strong motivation for better business performance, better software quality and higher productivity. Whatever the goal, investment must be made in terms of finance and resources, and there must also be support from all levels of management. These are some of the factors that can promote or hinder software process improvement initiatives. [Kasse, 1998] provides the following list of factors that can affect process improvement:

- History.
- Financial resources.
- Human resources.
- Capabilities of software engineers and developers.
- Technology support available.
- Contractual obligations
- Scope.
- Customs and culture.
- Standards.

- Understanding and support from all levels of management.
- Corporate political pressure.
- Business objectives.
- Vision.

A comparison can be made between the proposals outlined in the previous section and the factors that contribute to hindrances of software process improvement. Table 1 below shows these comparisons. The following subsections discuss how the process improvement hindrances may be effected by the proposals outlined above.

Proposal	**Factors to overcome**
Use of source code analysis tools will facilitate the process of re-engineering of existing software systems.	• Financial resources. • Technology support. • Human resources. • Customs and culture. • Capabilities of software engineers and developers. • Business objectives.
A formal testing process can contribute to more reliable software.	• Business objectives. • Customs and culture. • Capabilities of software engineers and developers.
Software process modelling will help to facilitate a more predictable or manageable systems development process for projects.	• Customs and culture. • Technology support. • Capabilities of software engineers and developers. • Business objectives.
Introduction of new requirements process will allow developers to understand and satisfy user requirements for new systems.	• Customs and culture. • Standards. • Understanding from all levels of management. • Business objectives.

Table 1. A comparison between the proposals and the factors of software process improvement they must overcome.

Case 1 was the associated with the biggest project and one of the main drivers for the T.C.S. project itself. The Year 2000 problem forced many companies to perform maintenance on their computer systems that are not Year 2000 compliant. P.C.W. was no exception as many production lines in the factory were automated with computer systems and platforms that are not Year 2000 compliant. The main factors were overcome here were business objectives, financial resources and technology support.

The company had to invest in new, Year 2000 compliant computer systems in order to achieve the business objective. That is, to make certain that the business would continue as normal after the Year 2000. This resulted in an adaptive software maintenance task. The I.T. department had little resource and technology to accomplish such a large maintenance project. Hence, the T.C.S. project aimed to provide technology support by allowing the company to use CASE tools that were available from the University of Durham. Additional support was given as the author of this thesis performed software maintenance on one of the production line controller systems using the CASE tool.

Case 2, affected customs and cultures because it forced developers to allocate more effort, time and resources to the testing phase of the project. Often, testing is not properly executed due the urgent demand from the customer for the new system. Also, in a manufacturing environment there is little opportunity to implement thorough testing because it may require downtime on a production line and that costs money. This implies that the customers, as well as the developers, must understand the importance of thorough testing. Lack of thorough testing can result in failures and this can cause downtime to occur until the problem is resolved. Hence, testing could save time, money and reduce post-development rework effort in the long term.

For case 3, there was a strong impact on the customs and culture, and also on the capabilities of the software engineers because this proposal challenged engineers to 'take a step back' and design the process before it was undertaken. In manufacturing environment such as P.C.W., there is an extreme sense of urgency for systems that will support the manufacturing process. Under such high pressure, software engineers launch themselves into the project without serious planning and preparation. The concepts of software processes must be understood clearly for it to be effective. The company uses standards and procedures that can discourage the use of an independent process implementation because they have become a habitual practice. Hence, factors of culture and capabilities of software engineers in the company were affected.

At P.C.W., there was a procedure for information systems development and it specified that a user requirements specification (URS) must be written and agreed upon between the developers and customers before development starts. However, the procedure did not specify how the specification should be written. Hence, case 4 was in an attempt to implement a controlled requirements process that provided a URS. This would have to break through the customs and culture factors as members of the I.T. department would have to learn a new procedure and customers would have to be introduced to it. Heads of department who may also be on the management team usually approved new systems development. They have a role in the defining the URS. Hence, it is essential that all levels of management understand the concepts of the process and the improvements that it would make. The most important factor that this case affected was supporting business objectives. A new requirements process would underline what business objectives the new system would aim to achieve and how each requirement would contribute to supporting business processes.

2.6. Summary.

Many companies have identified the need for improvement of their software processes. These companies have found that there are many factors that may hinder software process improvement initiatives. Many of these have been outlined in this chapter. A manufacturing company had been used to study the effects of software process improvement on a medium sized business. Four proposals were made with respect to two different projects and they will be discussed, as case studies, in detail in chapters 4 to 7. Each proposal aimed to provide benefits to the business by way of improvement to the existing software practices in the company. To support these case studies, the following chapter presents an investigation into software process improvement.

Chapter 3

An Investigation into Software Process Improvement.

3.1 Introduction.

In the business world, no matter what the discipline, constraints are placed on projects - namely time and money. With these constraints imposed by the business, software developers must also build a software system with the complexities that software itself possesses. Humprey states the following:

"It is not easy to produce quality software on time and on schedule. And, as tough as this work is today, it will not get any easier. That, however, is our problem. From the customers' perspective, our dates are no better than guesses. Even leading software organizations regularly adjust delivery dates several times before first product shipment. For other products or services, a deal is a deal. If the supplier raised the price, you would not be happy; you might even refuse to pay. Except for software, we expect firm dates and prices. [Humphrey, 1999]"

No ad hoc approach will provide quality software that meets all the customer requirements within a given deadline and budget. A systematic approach is needed where control is maintained over the activities of software development and where predictions can be made to facilitate decision making. This approach of development is called a *software process*. This chapter attempts to explain the need for software processes and the need for software process improvement in order to improve the quality of the end product.

In the following sections, the first section is an explanation of the difficulties that developers must overcome when working with software. The next section is a discussion about the need for software processes, followed by a brief history of software processes. These topics then lead up to an explanation of software process improvement.

3.2. The Nature of Software.

It has been argued by [Brooks, 1995] that the inherent properties of software: *complexity, conformity, changeability,* and *invisibility* will always make it hard to build. These properties are said come from 'a construct of interlocking concepts: data sets, relationships among the data items, algorithms, and invocations of functions.' These properties are related to the difficulties that make software maintenance a challenging task for organisations. This section discusses the concept of 'no silver bullet' first applied to software engineering by Brooks.

[Brooks, 1995] believes that there is 'no silver bullet' providing a solution to the problems of software engineering. He believes that this is due to the nature of software itself that will always make software engineering a hard discipline. The 'inherent problems of software' have a particular effect on the maintenance of software. The following sub-paragraphs look at the problems of software identified by Brooks - complexity, conformity, changeability, and invisibility.

Brooks states that complexity is a property of software and that larger software systems require a larger number of different elements not an increase in scale of the components. The problem relating to software development is the complexity of the software structures that causes difficulties when extending programs without creating side effects [Brooks, 1995]. Hence, many errors and bugs can be created if there is no proper management over implementation of requirements. One of Lehman's five laws of software systems supports Brooks stating that program changes cause its structure to become more complex unless active efforts are made to avoid this phenomenon [Lehman, 1980]. There may be additional cost to the development project if the changes that are made, due to new, unexpected requirements, cause more errors to be introduced, hence, a further cost for their correction.

Software in any given system must provide functions that are specified by the user. Through time the requirements of the system may extend beyond its original specification and, hence, further effort is required to provide new functions to satisfy the users' new requirements. The property of changeability is a cause of software

development extending beyond the deadline and being above budget. The difficulty arises when changeability is not considered in the early phases of software development. Changeability effects not only different parts of the software, but also other components such as design and specification documentation, and testing documentation. Hence, changes made to the software can have repercussions on other components of the system.

Another inherent property identified by Brooks is the invisibility of the software. In textual form, source code is extremely difficult to comprehend. The complexity of the software can be reduced dramatically by visual, or graphical, representations. Many tools exist today that provide visual representations in the form of dependency graphs and control flow graphs. However, Brooks argues that while these graphs simplify the structures of the software, but they still "do not permit the mind to use some of its most powerful tools."

Conformity is another property that is inherent to software. Brooks maintains that there is no conformity in software because different people design different 'interfaces' of the software. This makes development hard because it is usually the case that more than one person will work on a project and each individual must write a piece of code that interfaces with that of another individuals' piece of code. Hence, there is a need for strong understanding of the users' requirements among all developers, one solution being the use of accurate designs as a communication medium.

3.3 The Need for Software processes.

One critical requirement for lasting business success is meeting customers' needs and doing it as well or better than the competition [Humphrey, 1999]. Customers expect the best possible solution within a given deadline and at a minimum budget. The following sections discuss the previous sentence in more detail.

3.3.1 'The best possible solution'.

The 'best possible solution' to a problem would be one that satisfies all the customers' requirements and with properties such as reliability and high quality. Ghezzi and Cugola describe why this expectation is not always met by saying that:

- customer requirements need to be gathered.
- customer requirements will change.
- customer requirements will have to be validated and verified [Cugola, 1998].

When requirements are gathered (or elicited) customers will often give informal requirements where little detail is given leaving the developers with an ambiguous specification. Sometimes a cause of this is that the customer doesn't really know what they want, perhaps due to their own lack of knowledge of the system domain. If requirements are unclear from the outset, then the continuing development will produce a system that will not satisfy the customers. As a result, there will be high post-production costs modifying and correcting the implemented requirements.

Once customers provide a set of requirements it is likely that some of these may change over the duration of the project. With unclear and informal requirements, there is continuous interruption to the development process as time is spent re-working causing delay. At times changes to a set of requirements will cause other requirements to change as relationships exist amongst them. Delay due to implementation rework, caused when new requirements are specified after the requirements gathering process is a result of additional implementation work associated with such new requirements. Changes to software and to documentation should be controlled, and communication between the development team and customer must be maintained through out the development [Cugola, 1998].

In order to determine if the 'best possible solution' has been achieved, there must be a way of validating and verifying the requirements. Primarily, various forms of testing are used to achieve validation and verification. The development team must determine whether all the given requirements have been satisfied and this requires a successful

acceptance test execution. However, this testing itself cannot be relied upon, as time will be absorbed looking for and correcting errors. Action must be taken from the outset to prevent the final implementation of the system from failing during testing.

3.3.2. 'Within a given deadline'.

The 'given deadline' is a time constraint imposed by the business for reasons such as response to market changes, time-to-market of a product or to gain market leadership amongst the competition. Imposing a time constraint on any type of project means that all the activities that are required to make the product must be planned accurately. That is, the period of time that is allocated must be used efficiently. An ad hoc approach to development will result in an incomplete system (where all requirements are not satisfied) at the agreed deadline, or a complete system beyond the deadline. In both cases, the consequences would be undesirable and costly.

3.3.3. 'At a minimum cost'.

A business must look at the economics of financing a software solution to a business problem. There is little point in a business undertaking a project that will not produce financial benefits. Hence, to ensure that the gains to the business are realisable, a budget will be set to restrict and control the amount of funding given to software development. The implications of limited money are limited resources or selective choice of hardware and applications for the software development environment. Where resources are concerned, developers must estimate the amount of effort required for achieving certain tasks. Referring to the point made in the previous section, adding new requirements or altering existing requirements will need extra effort at extra cost that will raise the budget above the original forecast.

With these constraints imposed on software development, it is no wonder that much research has been undertaken to learn and to understand how quality systems can be built within these boundaries. Especially, when the properties of software, identified by Brooks, can make it more difficult for developers to discover the software solution. It is, and has been for a long time, necessary for developers to define an efficient,

productive, cost-effective, predictive and controllable method of working. In [Boehm, 1988], Boehm & Papaccio state that one way of controlling cost is to optimize our software development and evolution strategy around predictability and control. Such strategies have been sought throughout the last three decades and they are the foundations of the software process technology of today. The following section provides a brief history of software processes.

3.4. The History of Software Processes.

Research into software engineering has shown that by improving the software process, there can be an improvement to the software quality. Apart from quality, there are other goals for improving the process and these are: to improve maintenance and control, reduce delays in responding to the user's needs, extend the software's life and to put the organization in a position to take advantage of the new and emerging technologies [Miller, 1983]. Cugola and Ghezzi [Cugola, 1998] provide a retrospective view of how software processes have evolved. They identify the following stages in software engineering over the last three decades:

- The software lifecycle.
- Methodology.
- Formal Specification.
- Automation.
- Management and Improvements.
- Programming processes.

Each of these stages will be discussed further in the following sections.

3.4.1 The Software Lifecycle.

The software lifecycle divides a software development project into predefined phases. These phases are sequential and are linked by documentation that is created from one phase and fed into the next. Examples of documentation are requirements specification, designs and test data. A good example of a lifecycle is the Waterfall Model [Royce, 1970] which is shown in Figure 1.

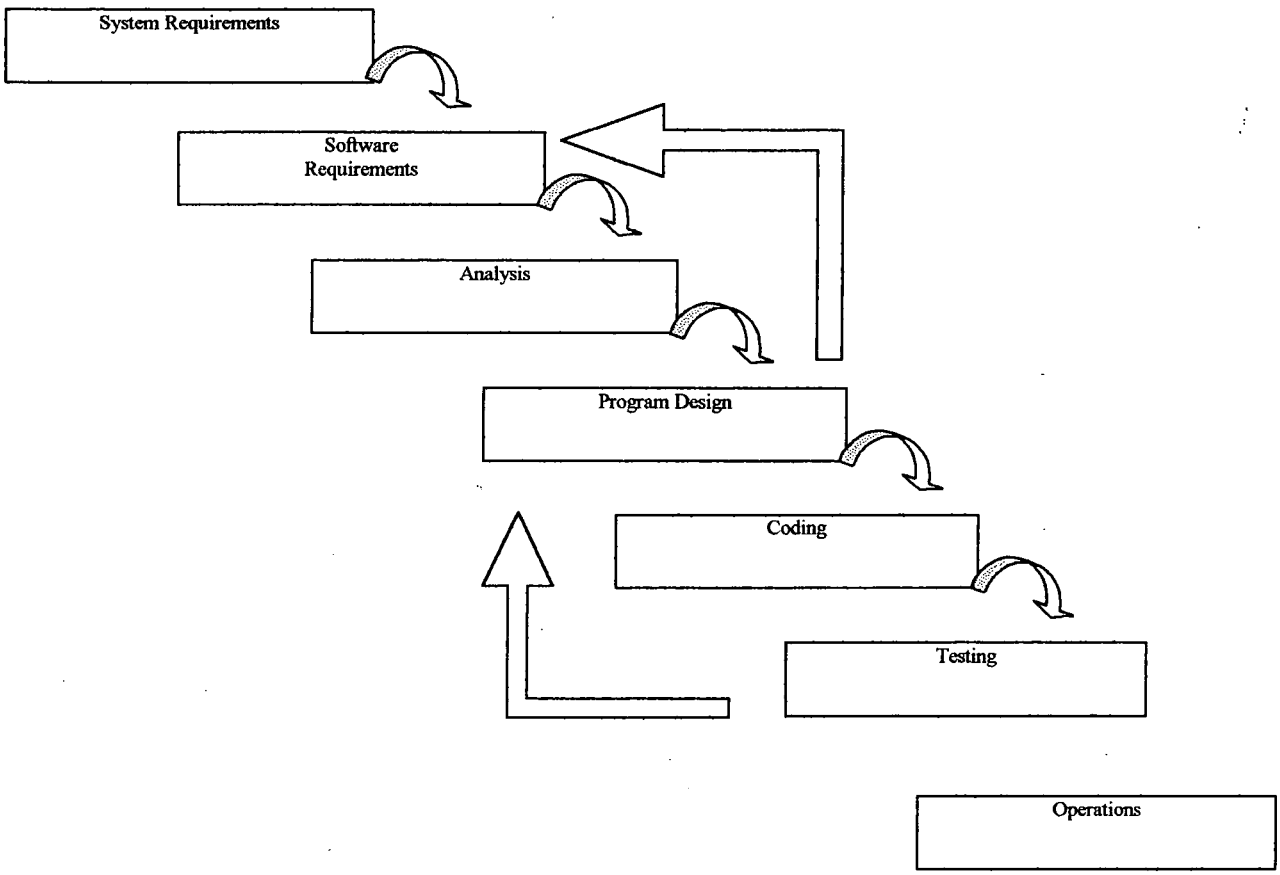


Fig. 1. The Waterfall Model.[1]

The Waterfall Model suggested that the development should proceed in a linear fashion, otherwise the process is difficult to control and to predict. In 1970, this was a considerable stepping stone in history of software processes because it went beyond ad hoc development and it recognised the need for management and documentation of requirements throughout the development process.

[1]taken from Royce, 1970 – figure 4 in the original paper.

However, there are fundamental flaws to the Waterfall Model [Royce, 1970]. Any changes that are requested late into the lifecycle manifests themselves in high maintenance costs, that is, post-production changes to the 'completed' system. The assumption that software development will proceed in a linear fashion is not true for most software projects. There is frequently the need to perform certain tasks in parallel to each other and there is frequently the need to make changes to decisions made in previous phases. Finally, one model cannot fit all universal projects. Different projects require different approaches to discover and implement the software solution. Hence, there certainly was a need for alternative approaches to software development.

3.4.2 Methodology.

Methodology provided more individual approaches to software development. That is, practitioners began to prescribe the phases (or activities) that should be taken to develop certain types of systems e.g. real-time systems, business information systems. Each methodology also prescribed notations that should be used to express requirements and designs. The main feature of all the different methodologies is that they are all born out of people's experiences [Cugola, 1998].

Despite being more applicable to a variety of problems than Software Lifecycles, there were still some failures in methodologies. Most developers using this approach misused it by applying a methodology in the wrong context. That is, some methodologies were used in business information systems development when they were actually created from and proposed for real-time application development. Developers also saw methodologies as recipes to solutions, but they are actually intended to be guidelines for developing software solutions. Methodology also required a lot of paper work for documentation of requirements and designs, that in turn, also cost a lot of development time. Notations for documentation were not always easy to master and documentation was not always be accurate, resulting in ambiguity.

3.4.3 Formal Specification.

Formal Specification is based on the assumption that programs are mathematical entities that can be formally specified, proven to be correct, and developed correctly by means of calculi for program derivation [Cugola, 1998]. It has contributed considerably to research into software programming, but failed as a general solution to the software development problem. The main causes for failure for this approach are that it assumes software functionality is known from the outset and that it does not consider non-functional requirements. For complex systems, the mathematical calculations can be time consuming and very intricate, requiring people who possess expertise in this field.

Formal Specification can be summarised by saying that it was useful for systematic programming, not software development.

3.4.4. Automation.

During the 1970s more tools began to emerge as part of the operating system revolution. Tools such as UNIX and UNIX workbenches provided better facilities for software engineers that improved programming methods. It was thought that these tools could be flexibly combined to achieve power and complexity. This lead to further studies into software development environments and language-based environments. Automation continued to dominate into the 1980s where business information systems were developed by users of 4th Generation Languages, and incidentally, not always by expert programmers.

Despite the advances that automation provided, there was seemingly little contribution to providing software processes that delivered better quality and more reliable systems. Automation, too, had limitations. For instance, automation only automated simple steps that facilitated programming. However, the need to master complexity still exists. That is, understanding the problem, defining the algorithm, and verifying the implementation of the solution are all human-orientated tasks. In terms of software processes, automation does not assist in requirements acquisition, specification and

critical design decisions. There was still the need to integrate the technical with the non-technical aspects of software processes, which automation failed to address.

3.4.5. Management and Improvements.

During the 1980s, the fundamental issue was quality as software began to play a larger role in businesses throughout the world. At this time there was a focus on the Japanese who were renown for their attention to processes that would guarantee quality products. It became more apparent that the quality of software can be increased if the process of development was improved. Consumers of software began to look for software developers who could guarantee quality in their products. As a result, international quality standards were defined and institutes began to certify organisations that would pass certain criteria that were needed in their software development processes.

Once organisations have been certified, they may continue to use the same processes over and over again and cease to look for improvements. For this reason more effort had been made in introducing standards that meet the needs of organisations around the world where process evaluation and improvement are continual. Examples of such emerging standards are CMM and SPICE (developed as an ISO standard). These are discussed in the subsequent sections.

3.4.6. Programming Processes.

During the late 1980s, researchers became aware that every software project was unique and individual for every organisation, even within the organisation itself. It was accepted after many years of experience that there is no unique, ready-made software development process.

Software developments depend on the problem to be solved, the culture of the development and customer organisations, and their environments. Hence, each project had to have its own process and these processes have to be designed, agreed upon and

communicated. This lead to the creation of the Software Process Models. [Dowson, 1985] describes them as

"A purely descriptive representation of the software process. A software process model should represent attributes of a range of particular software processes and be sufficiently specific to allow reasoning about them."

Like software itself, process models can be validated, verified and executed, to provide run-time support. That is, they are designed and implemented much the same as software where errors are identified and removed on each 'run-time', hence, software process improvement. Process Programming tailors the method of working according to the following:

- the problem domain,
- the environment for development of the solution,
- the customer/developer relationship,
- the interaction of people,
- the places in the process where automation will be used [Cugola, 1998].

This brief history of software processes shows how software processes started as very general prescriptions (life cycle models) for software systems development, eventually leading to very specific prescriptions for individual projects (process programming). Humphrey has also taken the work further by promoting personal software processes (PSP) [Humprhey, 1995] where developers aim to improve their roles in a software project.

These methods of software process improvement require the developers to design software processes. As in any discipline, software process design requires some form of technique to express and document ideas. Software process modelling is used for this purpose and is discussed more detail in chapter 6.

3.5. Software Process Improvement.

Many organisations that develop software are now facing the challenge of improving the quality of the software. Customers demand better software because their businesses will rely on it. There are organisations that will not tolerate defects and failures because they cannot afford to lose business performance. In some cases, failure of safe-critical systems can place people's lives at risk. Software developers are continuously under pressure to produce better software, whilst still keeping down production costs and achieving the specified delivery time.

Large and complex systems require detailed planning and control, and from this comes time, budget and resource constraints that developers must work within. The question is how the developers overcome the properties of software within the given constraints to produce high quality software. This is the problem faced by many organisations that develop software for their own businesses or for external customers. Organisations need to take a critical look at their software development processes and make evaluations regarding how they can improve their processes.

Two important software process improvement frameworks have been developed in the 1990s for organisations to co-ordinate software process improvement. The first of these is the SPICE standard, which is an international standard that consists of a 'suite' of documents. The second and earlier framework, CMM, has been used by organisations to understand their level of process maturity. Each framework guides organisations for implementing different aspects of software process improvement. Both frameworks are discussed in more detail in the following sections.

3.5.1. SPICE – Software Process and Capability dEtermination.

The description for SPICE as given in [ISO/IEC, 1995] is:

"This International Standard provides a framework for the assessment of software processes. This framework can be used by organizations involved in planning,

managing, monitoring, controlling and improving the acquisition, supply, development, operation, evolution and support of software."

The clear need for a standard for software process improvement became more apparent at the start of the 1990s. In 1991, a study report was approved by the ISO/IEC JTC1/SC7 that investigated the need for such a standard. Two years later the SPICE Project Organisation was established as a result of the study. A technical report was published under the title Software Process Assesment [ISO/IEC, 1995] that was divided into the following parts referred to as the 'document suite'.

Part 1: Concepts and introductory guide
Part 2: A model for process management
Part 3: Rating processes
Part 4: Guide to conducting assessment
Part 5: Construction, selection and use of assessment instruments and tools
Part 6: Qualification and training of assessors
Part 7: Guide for use in process improvement
Part 8: Guide for use in determining supplier process capability
Part 9: Vocabulary

The basic concept behind the SPICE standard is to assess organisations' processes using a framework that is both repeatable and comparable. The result of this assessment is then used to identify ways of improving the processes to support the organisation's goals. In general, there are three main core activities, which are process assessment (Parts 2,3,4,5,6), process improvement (Part 8) and process improvement (Part 7).

Unlike the CMM framework, SPICE provides more detailed and stringent guidelines for software process improvement allowing little flexibility for compromises. Such a standard would be suitable for large and ambitious software process improvement projects where benefits will be seen in the medium- to long-term future of the company. Implementing such a standard properly would require considerable resource,

expense, time and co-ordination that perhaps does not make it appropriate for this project.

3.5.2. The Capability Maturity Model (CMM).

[Humphrey, 1989] has proposed a maturity framework for software process improvement. This framework has five levels that can be used to identify the level of process control in an organization and they have the following characteristics:

Level 1. *Initial*: Process is under statistical control, but there is a need to achieve predictability of schedules and costs.

Level 2. *Repeatable*: Project management by commitments, costs, schedules and changes are in place in the organization that can be used to manage other projects.

Level 3. *Defined*: The is a defined process in place for implementation and better understanding.

Level 4. *Managed*: The defined process is analysed and measured for improvement.

Level 5. *Optimised*: There is now a foundation for continual improvement from the measures and statistics of each project.

For an organisation that aims to progress from level 1 to level 5, progress will be gradual as continual changes to processes will occur from one project to the next. For each level Humphrey has defined the controls that need to be in place for the organisation to progress to the level. For example, in level 3, one of the controls that an organisation must have in place is a defined process architecture. The CMM provides five levels of maturity that an organisation can climb, each maturity level stresses the managerial aspects.

This framwork is very flexible in that there are no strict guidelines to abide by. An organisation of any size can use it to apply improvements to their software processes. Thus, it was decided that application of the CMM would be suitable for these small, incremental software process improvement initiatives such as those proposed at P.C.W. in this project.

3.6. Application of software process improvement at P.C.W.

So far this chapter has provided information with respect to software, software processes and software process improvement. At this point it is necessary to understand how this information can be used for application to P.C.W. in a manner that will improve the software practices of the company. This section provides the background for an approach that was used to implement software process improvement to each case study mentioned in the previous chapter.

A systematic approach was required to undertake software process improvement and to prove the validity of the hypotheses mentioned in the previous chapter. Such an approach was essential so that each case study was carried out uniformly in order to achieve some degree of consistency between them. In other words, an aproach was required that was simple to understand, practical, methodical and showed a significant relationship to software process improvement. The IDEAL [Paulk, 1995] approach was discovered during an investigation into the uses of the CMM and has been chosen for this study.

3.6.1. The IDEAL approach.

IDEAL is an approach developed by the Software Engineering Institute. It is the overall framework that describes the necessary phases, activities, and resources for a successful software process improvement effort [Paulk, 1995]. IDEAL is an acronym of the five stage cycle. That is, *Initiating, Diagnosing, Establishing, Acting* and *Leveraging*. Each of these stages are described below.

Initiating. This stage involves creating the awareness for and gaining support for process improvement. The feasibility for implementing software process improvement will be identified in this stage where sound business benefits will be argued in order to gain the necessary support from management. The scope for software process improvement are also set here, possibly outlining the basic structure for the improvement programme.

Diagnosing. This stage is a two-part activity - study the current practice, and make recommendations. Process improvement instigators must have considerable knowledge and understanding of the current methods of software practice in the company. A study will show where processes are failing to achieve their goals, or if they can be achieved more effectively and productively. Recommendations can then be made to what is required to improve the process in order to achieve its goals, and these are used as a basis for the next stages.

Establishing. At this stage the strategies for how the improvements will be implemented are defined and efforts are made to secure financial and human resource support. Training may have to be given to learn new practices, methods, techniques and tools as part of the preparation to complete process improvement.

Acting. This is the implementation of strategies where new practices and techniques are used in place of old practices. Business benefits are not experienced immediately, but the wheels for long-term good are in motion at this stage. Acting can be viewed as the execution of design, but where errors are found small changes are made to initial plans. These changes are lessons learnt and are useful for future successes in process improvement.

Leveraging. This is the final stage and leads back to the beginning of the cycle. Leveraging will allow for lessons learnt throughout the cycle to be documented and revised. A retrospective view of measurement and results can show where goals have been met. Any failures in the process will result in running through the cycle again i.e. initiating, diagnosing, etc.

3.6.2. Application of IDEAL.

The IDEAL approach provides five phases for implementing software process improvement during the progression of each project. The following chapters will discuss the implementation of process improvement initiatives on each of the case studies. Each case study has been described in terms of the five phases of the IDEAL approach - initiating, diagnosing, establishing, acting and leveraging.

For each case study there will also be an outline of the business case and the technical case. The former will argue the benefits that the software process improvement proposal will provide to the business. The latter will argue the technical benefits from efficiencies gained from implementing the proposal.

3.7. Conclusions.

This chapter has explored various aspects of software process improvement. The research has been useful because it has shown how software projects require control and organisation. Hence, software processes are a means of establishing those factors when software is developed or maintained. Once an organisation is committed to devising software processes, it can further improve efficiency and productivity by improve on the processes it has already experienced. Small incremental steps can be taken to apply software process improvement at Philips Components Washington. This will lay the foundations for further improvement efforts and will also educate the I.T. staff at P.C.W. in new software engineering practices and concepts.

The company has always implemented a classical approach to software development (requirements, design, implementation, testing and installation) whilst abiding by corporate procedures. These practices are outdated and should be upgraded in the light of rapid changes in the business that will require much tighter control over software development and maintenance. Hence, the software process improvement initiatives discussed in the following chapters show how software practices can be upgraded in an incremental, step-wise fashion using the CMM as a guide and the IDEAL approach as a framework for each initiative.

Using the research that has been presented in here, the software development practices of the I.T. department at Philips Components Washington can be improved by:

- understanding the difficulties that are experienced in software development and maintenance,
- utilising software modelling to design better development processes, and

- aiming to gain a higher level of maturity on the CMM.

Chapter 4

Case 1:
Re-engineering of production line system software to adapt to a new database platform for Year 2000 compliance.

4.1 Introduction.

In recent developments in software maintenance, many tools have emerged that can support software evolution. Such tools allow maintainers to understand the functionality of the software, as well as the effects of implementing a change. The case study in this chapter demonstrates how one such tool was used at P.C.W. to improve software maintenance on production line control systems as part of a Year-2000 compliance project. The incorporation of tools in the maintenance process is considered with respect to supporting the re-engineering process.

The hypothesis for this case study is:

Use of source code analysis tools will facilitate the process of re-engineering of the existing software systems.

4.1.1. The business case.

In manufacturing there is a deep sense of urgency for any kind of systems maintenance whether it is mechanical or software malfunction. If such failures cause production downtime or are likely to in the near future, then maintenance must be undertaken swiftly and effectively. Swiftly, in order to reduce the amount of downtime because money is lost when production schedules are broken. Effectively, in that new errors must not be introduced in the process of amending existing ones causing interruptions to future schedules and, hence, further losses. It is essential in such a fierce business environment that customers are not let down because of inadequate maintenance operations.

The business case for using software maintenance tools is the fact that considerable time can be saved when the rate of program understanding is increased. This can be

achieved through a process of re-engineering where design recovery provides abstract views of the code that supply the maintainer with sufficient information to become familiar with the code's functionality. Such tools support the business needs by reducing the maintenance time with automated facilities and ensuring that changes to the systems software will not result in anything untoward.

4.1.2. The technical case.

Automated re-engineering tools can increase productivity where high level views of code and rapid navigation through different part of the code in HTML fashion can allow the maintainer to understand the "whole picture" of the code. Use of such automated tools can raise the level of maturity in the CMM [Humphrey, 1989], and so effect a process improvement.

At P.C.W., the technical case for using such a tool - especially when the cost of using them is zero - is very strong. Without the tool, maintainers must read through different source code files searching for the relevant lines of code using a standard text editor. This is an extremely time consuming and tedious task leaving the maintainer with a large margin of error. Automated tools have facilities that remove the tedious tasks allowing the maintainer to confidently apply more effort in analysing the problem and creatively solving it.

4.2. Initiating.

At P.C.W., the Year 2000 problem was the one, and only cause of this very large, time-consuming re-engineering project. If one of the original developers of the production line controller systems had not remained, the company would have had to invest much more to re-engineer all the systems software. This is because software maintenance is such a complex task to achieve successfully if an engineer is not familiar with the software or the functionality for which it was designed and implemented. This background section explains the Year 2000 problem at P.C.W. and how research into software maintenance has been used to improve the process applied in the Y2k project.

4.2.1. Year 2000 Problem.

In a previous survey of the factory, the I.T. department declared that the Year 2000 problem would have an impact on the production line controller systems at P.C.W. The four areas of impact for each system were the Image databases that store the production data and performance data; the HP 9000 Server; the operating system; and the application software. The impact to the business was critical.

Hewlett Packard had stated that the HP 9000/835S server would not be tested for Year 2000 compliance and would not be supported after the 01/01/00. The Image database has been obsolete for five years and Hewlett Packard had stated that it would not be supported or tested for Year 2000 compliance. The existing operating system and the application software (ANSI C) are also not Year 2000 compliant.

To resolve the Year 2000 problem on the production line controllers systems, the company had to invest in new HP D320 servers running the HP-UX operating system 10.2 with Year 2000 patches and an Oracle 8 database management system (the current market leader). Changes to the line controller systems are shown in table 1.

Line Controller.	*Model*	*OS Version*	*DBMS*
Existing version	HP 9000/835S	HP-UX 8.0	HP Image DBMS
Year 2000 Compliant Version	HP D320	HP-UX 10.2	Oracle 8 DBMS

Table 1. Summary of changes to Auto-YAMA-2 line controller.

Using Oracle 8 database required a new interface that would allow production staff to communicate with the line controller. To reduce the effort required to develop a terminal-based interface, the company invested in Oracle Developer 2000 in order to provide a true client/server architecture. The client end provides interaction with the server using forms and reports. Consequently, the terminals will have to be replaced with new personal computers capable of providing an improved forms and reports based interface.

The greatest challenge to the company in the project was that the implementation of a new database system would have a major impact on the software. In the existing system, the software communicates with the database by complex C functions. The Oracle 8 database system provides the utility to embed SQL statements into the software. The complexity of the code would be reduced, as more comprehensible SQL statements would perform the same operations.

However, as a consequence of using embedded SQL, the entire code for all production line controllers would have to be re-engineered to replace existing database functions. The essential functions that are affected by SQL statements are those that directly interact with the database tables. These functions read, write or update data in the tables; hence, these are the areas that will require embedded SQL statements.

4.2.2. The process of re-engineering.

The Year 2000 problem at P.C.W. has meant that all the production line controller software had to undergo *adaptive* maintenance. That is, the source code had to be changed to adapt to the new, Year 2000 compliant Oracle Databases. Such a maintenance task requires changes to be made to the code in order to adapt to changes that are external to the system whilst the functionality must remain exactly the same. This task is extremely difficult and expensive the following reasons:

- All six production line controller systems were large and the software is very complex.
- No design documentation existed that showed each of the system's functionality.
- Functionality of each system was specific to each line and had to be extracted using the code.

An approach that was suitable for this type of maintenance situation was 're-engineering'. The definition for re-engineering given in [Frazer, 1992] that states:

"Re-engineering is the process of reverse engineering a subject to a chosen level of abstraction and then reconstituting the system by means of forward engineering. Very

often it also involves introducing modifications to the system functionality prior to forward engineering."

The re-engineering process is presented in [Frazer, 1992] and detailed in fig.1 below.

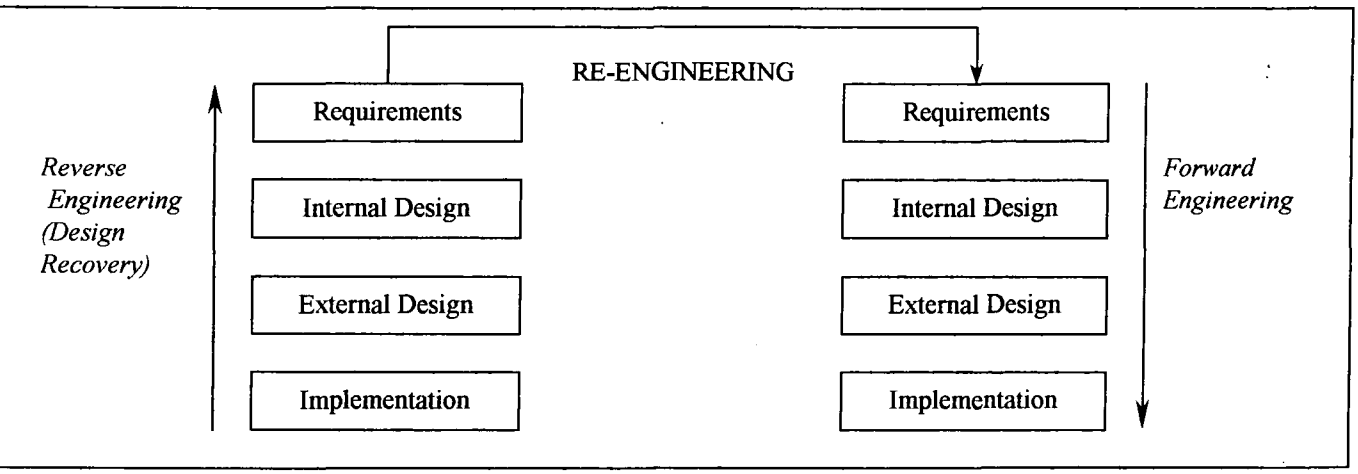


Figure 1. The re-engineering process adapted from [Frazer,1992].

Figure 1 shows how source code (implementation) is used to extract the design that is then used to obtain the original requirements. This is reverse engineering where representations of the code at a higher level of abstraction are obtained and used for analysis and documentation. Whilst this is a systematic and methodical approach, there were problems associated to software maintenance that had to be overcome when implementing reverse engineering. These problems were:

- The complexity and magnitude of the production line source code made analysis extremely time consuming and laborious.
- The learning process was gradual due to difficulties in program comprehension.
- Documentation was very tedious to generate and would not be very accurate.

Forward engineering uses the requirements that are obtained from reverse engineering to generate new designs (if alterations and new requirements are added) and to implement the new source code. However, this part of re-engineering also had problems that were associated to software maintenance. The problems that had to be overcome here are that:

- The effects of making a change to one part of the code on other parts were unknown.

- Using a standard text editor (VI Editor), making changes would be arduous due to the large number of files and lines of code.

The problems that were associated with both reverse engineering and forward engineering made this is a very expensive process for the company. This was due to the time lost because the original functionality is difficult to reproduce and the task of implementing changes must be accurate so that new errors were not introduced into the code. The next section discusses how using automated tools that were developed to facilitate the re-engineering process would alleviate these problems.

4.3. Diagnosing.

The company had to meet the Year 2000 deadline otherwise production will cease on the lines that have not been re-engineered for Year 2000 compliance. One of the lines that is not Year 2000 compliant, was assigned for re-engineering to the author of this thesis who had no prior knowledge or experience with these systems. Whilst the re-engineering process was understood, there are still many difficulties (discussed in the last section) that had to be overcome. This section discusses how CASE tools facilitated the process of re-engineering and how they were used to assist in the software maintenance process at P.C.W.

4.3.1. Facilitating the re-engineering process.

The first part of the re-engineering process is reverse engineering that involves detailed study of the source code and high level documentation to be written in order to understand the system. There are many tools today that can alleviate code comprehension in reverse engineering and code modification in forward engineering, although Brooks argues that they only help to an extent and do not allow the mind to use its most powerful conceptual tools. The A.U. tool [Boldyreff, 1995] that was available for use at P.C.W. is one such tool, but there are other examples that are discussed in [Tilley, 1998]. A commercial example is the SNIFF+ tool (see Appendix C1 for full description and diagram) that was also available for use at P.C.W. for a

fortnight. The advantage of such tools were that they automated the tasks that made re-engineering very time-consuming, as follows:

- They presented the relationships between data structures, identifiers, procedures and functions in a high-level, graphical format.
- They accelerated the navigation between files, for instance, using a HTML format a maintainer can click on a function call to see its full implementation.
- They generated documentation for the source code automatically.

Tilley [Tilley, 1998] says that software engineers must spend an inordinate amount of time creating representations of a system's high-level architecture from analysis of its low-level source code. Code analysis is a tedious process where the code must be thoroughly studied and simplified for the maintainer to gain sufficient amount of knowledge to implement the changes that are required for the maintenance task. The more sophisticated and accurate method is to use computer-aided tool and techniques that extract a high level information from the code automatically and instantly.

Tools such as the A.U. Tool and SNIFF+ provide facilities to 'jump' from file to file that allowed the maintainer to rapidly navigate through different parts of the source code. These tools also present the parts of the code that share data structures, procedures, functions and identifiers and allow easy access to them which means that changes to the code could be made more accurately.

By linking the different aspects of the system the maintainer could navigate between the code and the documentation. The Application Understanding (A.U.) tool contained this traceability feature in the tool set, hence, the links between modules were presented and allowed faster accumulation of information and understanding.

4.4. Establishing.

There were six out of the seven production line controllers at P.C.W. that required re-engineering for the new Oracle 8 Databases. One of the production line control systems was used for this software process improvement case study, whilst the others

were re-engineered by a software engineer who is already very familiar with the software. The assignment to re-engineer this production line controller system (known as Auto-YAMA(2)) was given to the author of this thesis and a project duration for 19 weeks was allowed for completion, including installation onto the production floor. Appendix C2 has a complete breakdown of the project in the form of a Gantt chart.

University of Durham permitted the use of the A.U. Tool [Boldyreff, 1995] which was available on Sun systems within the Department of Computer Science at the University. This tool had the capability of analysing ANSI-C software. Another commercial tool - SNIFF+ - was available as a trial version for two weeks. Both tools were designed for software development and software maintenance. There was no expense in obtaining these tools as the former was freely available from the university and the latter was licensed for two weeks as a free demonstration.

As part of the planning stage, a Gantt Chart was created to represent the time and resource allocation for the project that was accompanied by a process model that showed the activities for the whole project. This process model is shown in Figure 2 below.

4.5. Acting.

Figure 2 shows the activities undertaken for the re-engineering of Auto-YAMA(2). This is a detailed prescriptive model that represents the process of re-engineering of Auto-YAMA(2) from collecting the requirements to installation. This model was used to explain to production staff and managers how the project specifications detailed in the Gantt Chart will be satisfied. Figure 3 shows a process diagram that generalises the re-engineering process of Auto-YAMA(2) based on the re-engineering process shown in fig. 1. The subsequent paragraphs discuss the activities shown in Figure 3.

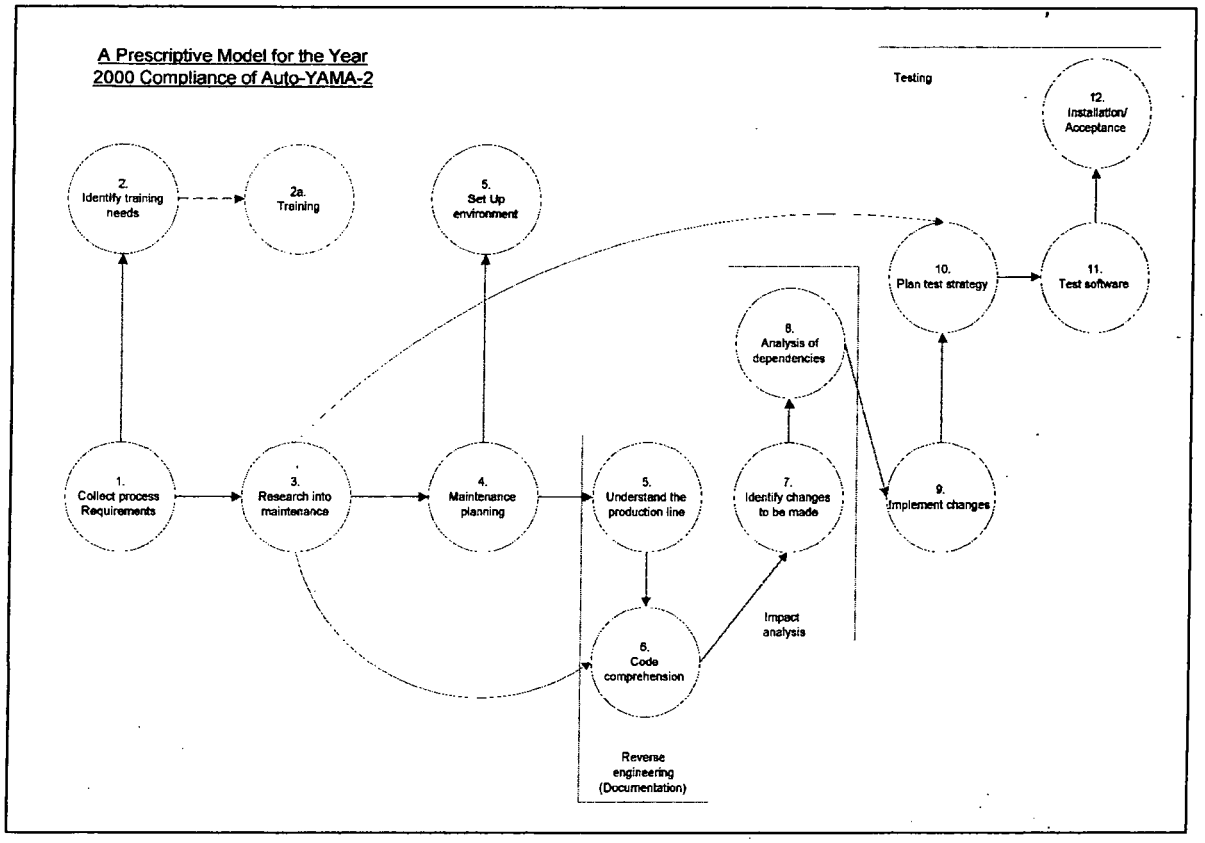


Figure 2. The overall process for re-engineering Auto-YAMA(2).

Activities 1 ,2 & 3.

An understanding of the functioning of a system is required before the maintenance process can begin. Frequently, the maintainer has no such knowledge of the system and obtaining this knowledge is both time consuming and expensive [Humphrey, 1989].

Figure 3 shows the general process of re-engineering for the Auto-YAMA(2). For this project, both a study of the environment that the system supports and the existing source code of the system were performed together. Activities 1, 2 and 3 are related to tasks 2 to 7 in the Gantt chart that is shown in Appendix C2.

In activity 1, a document was written that described the Auto-YAMA(2) production line process and how a product is assembled by the numerous automated cells on the line. Little documentation was available and most of the information was gathered by talking to production staff and process engineers. During this period, there was an

increase in the knowledge of the role played by the production line controller in the whole manufacturing process.

In section 6.3.1., the principles of re-engineering were said to be an aid to program understanding whereby the original design (or functionality) is recovered and documented. The A.U. Tool was used in activity 1 to assist in understanding the general functionality of the source code. This was achieved by running the source code through the A.U. tool and using the HTML facilities to navigate though various parts of the code. Rapid navigation provided a faster rate of understanding than manually reading through the code because relationships between different functions can be grasped, and also, implementations of function calls can be seen instantly allowing the mental model of the code to develop with less confusion. Appendix C3 shows an example of the function calls in the function "rec016". The document shows other functions and procedures used by rec016.

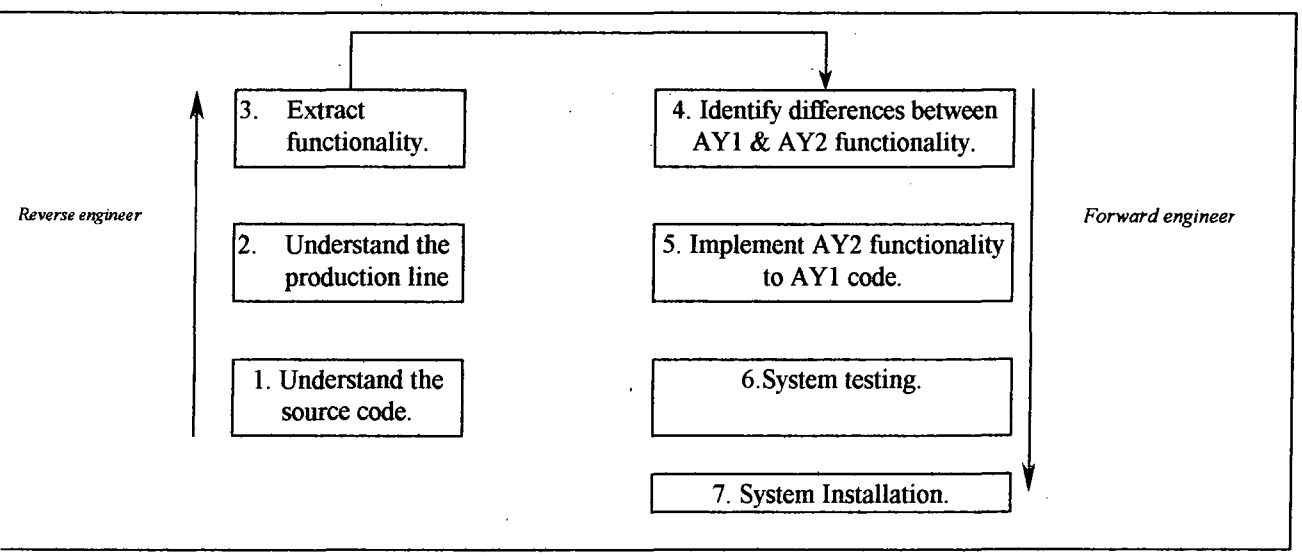


Fig. 3. The process of re-engineering specific to the Auto-YAMA(2) project.

The SNIFF+ tool (see appendix C1) also had these capabilities, but was more supportive because different colours were used to represent different attributes of the source code. For example, comments embedded in the source code were coloured red and variables were coloured blue. When such attributes are made more obvious then code understanding is made considerably easier. It was used in conjunction with the A.U. tool.

After the code and its environment were studied the next stage was to decide how the existing source code would be re-engineered so that the database communications could be changed to handle the Oracle 7 database.

Activity 4.

Auto-YAMA(1) is another production line that is, in principle, very much the same. The code for this production line controller had already been re-engineered and it seemed sensible and efficient to use this code as a base for the new Auto-YAMA(2) code. Two steps were involved as follows:

1) the differences between the lines of the code in the two systems were identified (using SNIFF+ and diff), and
2) the code for Auto-YAMA(1) was changed to meet the specifications of the existing Auto-YAMA(2) code.

The first part of this activity was to identify the differences in the actual production processes of the two Auto-YAMA lines. It is relates to the tasks under "Migrate code", tasks 8 to 17, in the Gantt chart (Appendix C2). Any differences here would explain relevant differences in their respective code. There were differences in the production processes because the products that are made on the two lines are slightly different. Many of the configurations were stored in the database that was updated by the code using C data structures.

The SNIFF+ tool was very useful in this activity as it showed which parts of the code used these structures that were declared in the .h files. These structures were displayed in a similar manner to the call graph shown in Appendix C3. Where ever these structures differed, there would be an impact on any part of the code that used that structure. Hence, with SNIFF+ it was immediately known which files were affected, and with HTML, the exact locations of any references to the structures were instantly accessible.

The A.U. tool was used to show the relationship between those functions that would be affected by differences in the data structures. Call graphs (Appendix C3 shows one

example) that showed these relationships were generated from each source file and printed out. Such documentation is essential for future learning and training of new staff. It also facilitates maintenance in the future.

As an extra conformation, the *diff* function that is standard in UNIX was used to find the differences between the old Auto-YAMA (1) and Auto-YAMA(2) source code files. Again those files that required changes were identified and this verified those changes identified through SNIFF+ and the A.U. tool.

Activity 5

Having identified exactly where the re-engineered Auto-YAMA(1) code must be changed to mimic the functionality of the existing Auto-YAMA(2) code, preparations were made to implement those changes. First the database was created using the scripts that were already generated from database creation for other previous systems. Again, the changes found using SNIFF+ and the A.U. tools supported database creation as non-master tables were of the same structure and type as the data structures in the .h files.

After the database was created, the changes to the code were implemented. The changes were made using the VI editor on UNIX because SNIFF+ was PC based tool whilst the A.U. tool was offsite. However, they still provided significant assistance to ensure that all the changes were made.

Activity 6.

Once all the changes had been implemented and compilation of the code was successfully complete, testing was the next stage. It was performed in a methodical manner, first each function was tested individually. Gradually functions were tested together and then the system was tested as a whole in order to validate the system's functionality accurately. Test scripts and results for each function can be seen in Appendix B. Examples show the procedure for the test, followed by the scripts that were used and the results outputted.

This testing ensured that the production line controller performed the functions exactly as the existing system performed it. However, Chapter 7 will discuss this in more detail, as it became one of the case studies for the software process improvement initiative on testing.

Activity 7.

After the testing phase was complete, there was still one final test left to pass. The system was installed onto the production line where it replaced the existing system. A two months trial period was proposed which in effect was the 'live' test of both the re-engineered source code and the front-end interface. Any faults or errors that were identified within these two months would mean that corrections would have to be made.

Errors were found in the front-end part of the software, which were mainly related to how the data was represented. They did not require any changes to the database structure, which subsequently, would require software changes. Once the two months has passed, the production line controller system was signed off as accepted and has remained to support the production exactly as the previous system ever since installation.

4.6. Leveraging.

Many lessons have been learnt from this case study. This was a new method of working for P.C.W. and new technology had been used to perform a task that had been undertaken by one engineer who used knowledge, expertise and experience without any specific software tools. The following outlines the important issues that characterise this case study:

- documentation was generated automatically during the re-engineering process,
- the rate of program understanding and learning was increased using the tools,

- tools provided sufficient understanding to a new software engineer allowing him to change source code as confidently as an experienced software engineer who had developed the system, and
- to understand the code, the new maintainer must learn a considerable amount about the process (the purpose) that has been designed to support.

Documentation is a valuable asset to any company that develops and maintains software. There was clear need for documentation when this project was started as no previous knowledge of this particular system was retained. The striking benefit was the automation of the documentation that can take weeks if accuracy and detail is required. A system of this size (130 Kloc) would require much detail to explain the functional aspects of the code and the environment it supports.

Previously, another production line controller system was documented manually where source was translated into pseudocode. This took approximately seven weeks and there was still no guarantee of accuracy and consistency. Automated documentation provided these elements from source code with speed, accuracy and maintainability. The Gantt chart (Appendix C2) shows that the reverse engineering was planned for only five days because trial source code analysis on these tools proved to be very fast. Hence, more time can be spent analysing design recovery documents than on design recovery itself. Task 24 on the Gantt chart was already performed during reverse engineering, hence, a further 15 working days was saved.

Such tools as SNIFF+ and the A.U. tool were very effective for program understanding and learning. HTML style navigation provides rapid references to different parts of the code. When a function call is encountered, it is possible to see the implementation of the function instantly to find out what part it plays and how the parameters passed to it are handled. With out this facility, the maintainer would have to exit out of a file and find the relevant file that contains the function's implementation. These manoeuvres are tedious, especially, when using a VI editor. The A.U. tool can generate call graphs (Appendix C3) that are similar to data-flow diagrams where relationships between different procedures and functions are depicted with boxes and

arrows. Unfortunately, it does not show the variables that are passed between them; however, it still provides an effective visual representation of the source code. This can also be useful for identifying reuseable software when patterns can be identified between different call graphs. Software reuse is recognised as an extremely effective method to reduce maintenance costs by minimising the coding effort [Karlsson, 1995]. Again, such tools as the ones used in this project can enable and facilitate technologies as software reuse that would provide long-term benefits.

Performing re-engineering without the use of tools can only be done if the maintainer has a deep understanding of the software. Such people are usually the developers of the software and, hence, their experience provides them with a unique insight that allows them to make changes to the software whilst knowing the 'knock-on' effects. This case study has shown the code analysis tools can provide considerable assistance to a maintainer who wants to know the consequences of making even a minor change. A good example from this case study are .h files that declare data structures used for passing data to and from the database tables. If the database tables are change, the data structures must be change and this requires many changes to numerous parts of the code. The SNIFF+ tool (see appendix C1) clearly shows which functions and procedures that share these data structures, so that they can be clearly identified instantly when a change is required. Rapid navigation takes the maintainer to the very spot that will require the change.

Finally, it appears in this case study that the code was orientated very closely to the production line process. Identifiers and structures were named using the names of objects in the database. Sometimes embedded comments in the code explain how the functionality supports that process. It was vital that some considerable effort was made to learn and understand the process and the environment that the software was designed to support. Malfunctions and errors may occur on a re-engineered system if the maintainer does not understood how a change to the process will affect the code. From this lesson, it can be argued that tools may not always provide complete support. Sometimes, a degree of knowledge and understanding of the whole production process and its strategy is required by the maintainer that cannot be obtained by simply using automated code analysis.

4.8. Summary.

This chapter has described the case study for using software analysis tools on a manufacturing project. The problems of software maintenance (namely complexity and changeability) are relieved by automatic facilities that support design recovery and navigation to different parts of the code. Two tools were used - the A.U. Tool from the University of Durham and the SNIFF+ tool available as a demo - on a re-engineering project on production line controller systems.

The tools provided facilities for identifying which parts of the code had to be changed and call graphs that served as pictorial documentation for the software. The line was re-engineered well before the deadline and this allowed more time for another case study based on improving the testing process.

Chapter 5

Case 2:
Testing line controller software.

5.1. Introduction.

In the development life cycle, testing is a crucial element if a customer is to receive a high quality product. Testing is the process of executing a program with the intent of finding errors [Myers, 1979]. It is the validation of the requirements that must be realised in a system that is delivered to the customer just as they have specified. However, testing is sometimes overlooked or truncated when deadlines are imminent. In other cases, developers have claimed to have applied testing before installation, but post-production maintenance may contradict such claims suggesting a lack of thoroughness.

The re-engineering case study discussed in Chapter 4 also provided an opportunity for software process improvement with respect to testing. For such an essential system, where little (if not no) post-production maintenance can be afforded, an adequate and methodical testing approach seemed vital before installation into the live environment. This chapter explains a case study that formalises the process of applying testing techniques to the system resulting from re-engineering project at P.C.W. (described in Chapter 4).

The hypothesis for this case study is:
A formal testing process can contribute to more reliable software.

5.1.1. The business case.

In a manufacturing environment, the pressure for fast computer solutions for business problems often cause testing to be limited to 'as long as it can be demonstrated that the basic requirements are met'. Failures occur when circumstances arise that cause the

solution to fail because these circumstances were not accounted for in the testing. This can lead to disruptions to the manufacturing process that lead to financial losses. Controls are the means used by organisations to minimise risk of undesirable events, and software testing is a form of control. The highest business risk should receive the most test resources [Perry, 1995].

Further losses are claimed when maintenance of the system is undertaken and testing can reduce these costs. Post-production work on a system also causes instability to the business as the users lose confidence and are incapable of fulfilling their roles.

Hence, from a business perspective there must be sufficient testing to avoid failures, but it must be efficient enough to meet the project deadline whilst also validating that all the user requirements are completely satisfied.

5.1.2. The technical case.

For the technical case, testing is good software practice and is recognised in the CMM as means of achieving Level 3, the Defined Process. The following diagram shows a defined process described by Fenton [Fenton, 1995] that shows how unit testing and testing plans are used in the development process.

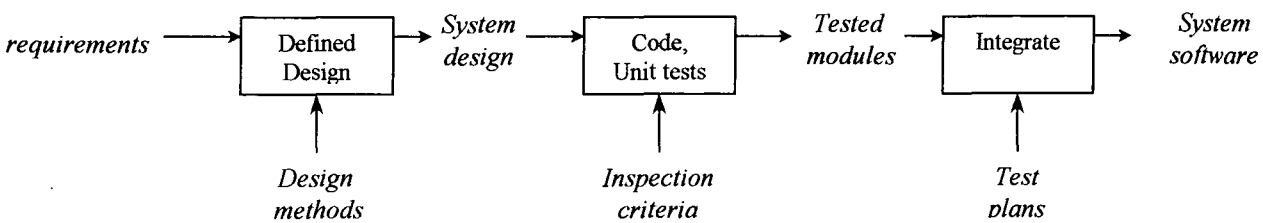


Fig.1 An example of a defined, testing process.

Developers who create very large systems within a team will, without a doubt, have written software that contains defects. For some business, these defects can harm the business, whilst for others they can be life-threatening. It is in the interest of the developers that a system should undergo testing before it is delivered to the customer.

Testing can also be viewed as a method of validating that the requirements have been satisfied. After all, the developer is aiming to meet the users' requirements, but they must be sure that even though requirements are implemented, they must function as the user has specified. Thorough testing of implemented code can reveal flaws in the previous phases and will force a development team to make improvements, thereby reducing the existence of defects in the final system at earlier stages.

However, for business reasons, there are time and budget constraints on a development project. [Humphrey, 1989] states that the question is not whether all the bugs have been found but whether the program is sufficiently good to stop testing. It is here that a compromise must be made for testing between the technical arguments and business arguments for testing.

5.2 Initiating.

A system that is essential to the business must provide a useful, efficient and productive service. It should facilitate a business process. However, as the demands for such systems are always urgent there is a tendency to reduce the period of testing, perhaps give it less priority.

In the re-engineering of a production line controller system (chapter 4) there was a clear need for ensuring that code functionality of the new system was exactly the previous system. P.C.W. rely on these systems for Deflection Unit manufacturing and it was vital that no time was lost during the installation stage or that this stage was postponed. Reducing any 'business risk' to production was essential.

Hence, there was a clear need to look into the field of software testing in order to find ways of building confidence into the system made by the I.T. department before it is delivered to production. The next section discusses software testing.

5.3. Diagnosing.

At P.C.W., the methods that were used to ensure that any newly developed systems were fit for the live, production environment were up to the individual developers associated with the system. In terms of company procedure, it was said that a system must enter a trial period where it was allowed to function in the live environment for an agreed duration after which it was 'signed off' by users, if they are satisfied with its functionality.

There was no defined testing process that developers must implement to validate and verify their systems at P.C.W. Nevertheless, testing was very much prevalent in all aspects of the company and it was essential in such an environment where system failures are costly to the business. Software developers in the past have used common testing methods such as 'functional testing' and 'installation testing'. However, the testing was not formally defined in a form that can be used again the future for process understanding or improvement. This section discusses concepts that are common to the testing discipline.

5.3.1. Testing Methods.

Myers [Myers, 1976] proposes seven types of testing:

- Unit or module tests.
- Integration tests.
- External function tests.
- Regression tests.
- System tests.
- Acceptance tests.
- Installation tests.

These different type of tests verify the lower levels of the programs (individual units or modules) and progress to a higher levels. Here, all the modules are tested as one (integration, external functions and regression tests), until full simulations of the

system's functionality as a whole are tested (system, acceptance and installation tests). Unit testing will most probably be done during development where programmers will verify that their functions perform as intended. Integration, external function and regression tests will be done in collaboration with other members of the development team, perhaps involving the customer to validate the functionality. System, acceptance and installation will most certainly involve users and customers. Acceptance and installation tests may be run through a trial period which will be enough time for the system to experience all real-world conditions.

5.3.2. Black box and White Box Testing.

Humphrey [Humphrey, 1989] explains the two basic ways of constructing tests - 'White box' and 'Black box' testing. The former is performed with knowledge of the program constructs to test the internals of the programs functionality. Whilst the latter requires knowledge of the program structure but aims to test that the program does exactly what it was supposed to do.

In terms of the seven types of testing, the white box testing will be performed during low level unit, integration and regression testing. It would inappropriate to white box test the system as whole. For this it would be more practical to do black box testing to verify that the general functionality meets its specification.

5.3.3. Overtesting and Undertesting.

The dilemma that is frequently experienced by software developers is the amount of testing that should be undertaken on a piece of software. Business provides many constraints, namely, deadlines, budget and resources. Testing must be economical enough to provide a reliable system that both the customers and developers will have confidence in, but within the constraints of business. Perry [Perry, 1995] discusses the economies of software testing in terms of undertesting and overtesting. He states the following:

- The risk of undertesting is directly translated into system defects present in the production environment.
- The risk of overtesting is the unnecessary use of valuable resources in testing computer systems that have no flaws, or so few flaws that the cost of testing exceeds the value of detecting system defects.

Both undertesting and overtesting involve risks to the business. The trade off should consider the probability of finding more bugs in test, the marginal cost of doing so, the possibility of the users encountering the remaining bugs, and the resulting impact of these bugs on the users [Humphrey, 1989].

5.4. Establishing.

The main task here was to define a strategy for testing that would make use of this time within the resources available. Most production line controller systems are based on the same message passing model, hence, there was some opportunity for re-use in the testing especially since the functions and procedures were taken from a common production line controller software base.

A previously tested production line controller system had been documented, but no effort had been made to have reusable test data that could have been used for testing of other systems. It seemed essential that test data should be documented and available for reuse because it can be used as evidence to prove that adequate testing had been undertaken. Also, it can show where testing was insufficient and where failures had occurred, bugs were present.

Section 7.3.1. shows the methods that should be used for thorough testing, however, with the time that is available it would not be practical to try to implement each method. There was a compromise in the process where some methods are grouped together. Also, there was the opportunity to automate some of the unit testing by using UNIX shell scripts where the message passing between modules could be simulated. Again, as many systems worked on the same message passing concept, there was

scope for re-use of these test scripts for testing other systems after maintenance has been performed on them.

The prescriptive process for testing is shown in Figure 2 below.

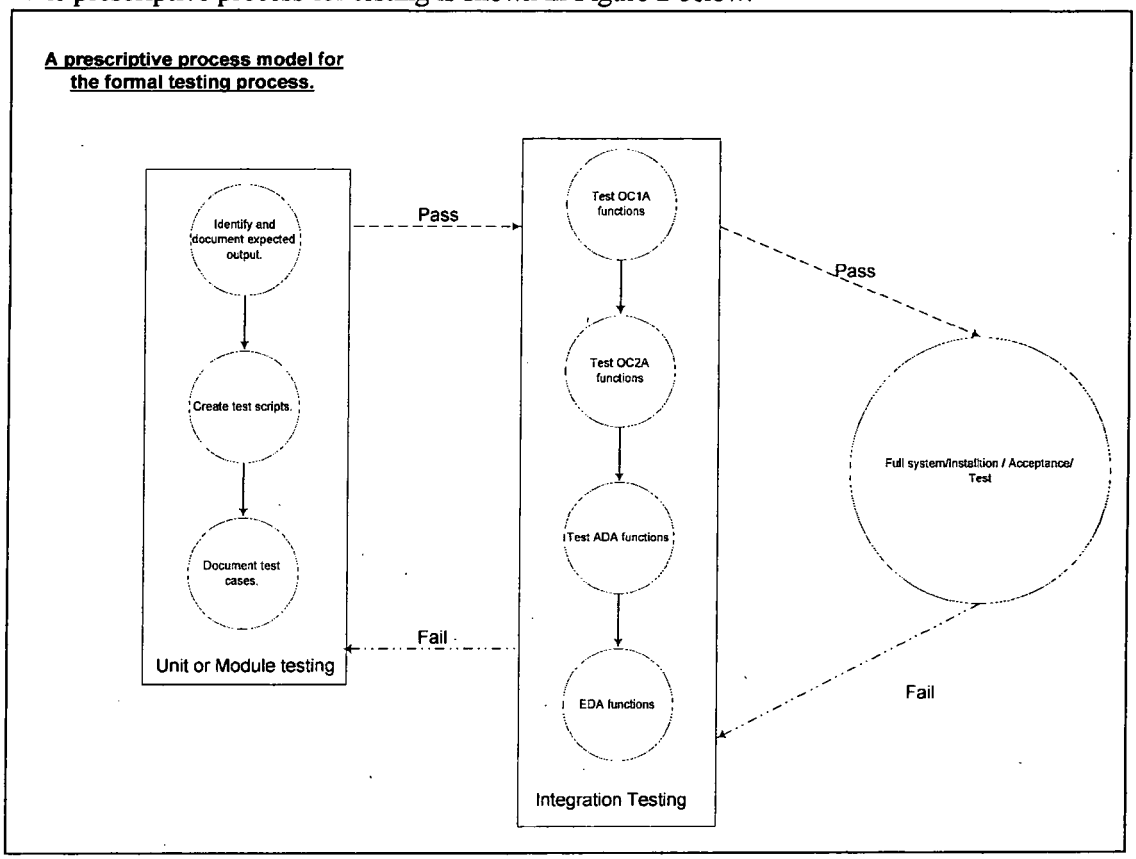


Figure 2. The overall testing process that should be executed to test the production line controller system discussed in chapter 6.

5.5. Acting.

The system uses a message passing principle where data is sent from an automated cell to the server where it is placed on a message queue. Each message would be prefixed with a number that is used by a message handler to remove the message and to process it. Hence, the message handler had a function for processing different messages (see figure 3). The diagram below shows a simple model of the system.

During the re-engineering of the production line controller system, each function of the message handler was re-engineered individually. Very basic white box testing was performed to ensure that each function performed as intended. Once all the functions

had been re-engineered (the end of the implementation stage), the focus turned to testing the functionality of the system as a whole before it was installed in the production environment.

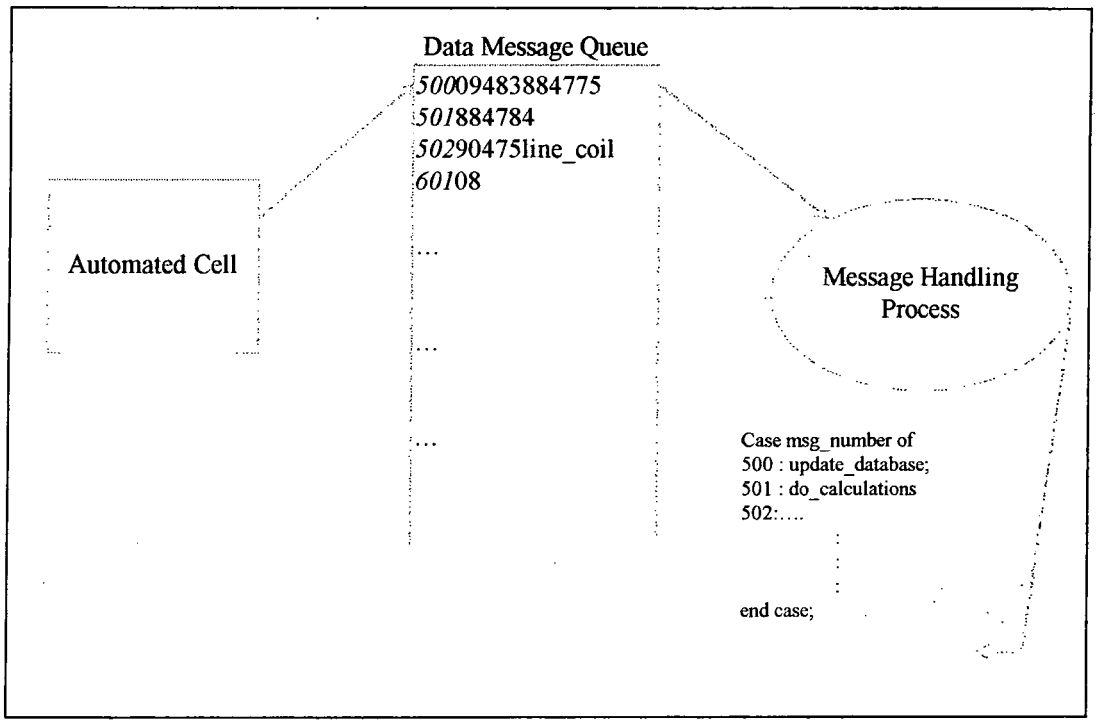


Fig.3. The message passing model of the production line controller system and the winding data collection system.

Creating test cases.

The message passing system contained many functions that were all called in a case statement. Within these functions there were other functions and procedures that processed the data sent by the automated cells as parameters. Such a system could be tested with some degree of reusability and automation. Placing the messages on the queue with shell scripts simulated the message delivery from the automated cells. These were regarded as 'test cases' where 'test data' would be used to test the behaviour of each function. The following statements are taken from the axioms of testing [Humphrey, 1989] and the apply to the case study:

- A necessary part of every test case is a description of the expected output.
- Write test cases for invalid as well as valid input conditions.

Appendix B shows the procedures and scripts that were written to document and automate the testing phase. The procedure provides a description of the test, it's purpose, the data input and expected output. Attached to the description is a print out of the evidence that the test has successfully passed.

Function testing

With shell scripts, each function can be tested individually (module or unit testing) - instantly and consistently - until any errors that are present are removed. These simulations mimic the conditions of the production environment making the possibility of failures in the live environment minimal. Appendix B shows the function test of "rec009" that validates a product type and checks that the product is currently in production. The following line is executed from a shell script

```
msgsnd 71 "0090001616" 202
```

where *msgsnd* is a function that sends the data "0090001616" to channel 202. The data string can be decomposed to

- 009 the message type,
- 0 – return flag,
- 001 – the identifier of the carrier that the product is transported on,
- 616 – the product type.

The cells that exist on the production line would send this message to the production line controller system. The system would identify that the message is of type 009 and would have a message handler that performed a specific function. In this case, the function searched the database to verify that product 616 is valid and in use, updated the database to record that the product carrier 001 was at the cell 71 and sent a 800 message back to cell 71 with data specific to product 616.

The evidence in Appendix B show that database tables that were updated and also show that cell 71 had been sent a 800 message as the script simulated cell 71 with

"msgrcv" function that took it off the message queue. There is also an output of the error logs that show the error messages that were logged from the test.

Performing this task manually would have taken a considerable amount of time, but automating this facility reduced testing time, whilst also making it accurate, documented and most importantly, reusable. Shell scripts also facilitated the integration testing that followed.

Integration testing.

The next stage was integration testing where shell scripts were used to simulate several automated cells all sending messages as they would in the production environment. Again, it is another step closer to a live environment, however, the inputs to the functions were all predefined and predictable. At this level, it was important to find errors that occurred due to one function that caused others to fail. For example, each function interacted with the database and if one failed to commit its transactions then all other functions would fail. Database integrity would be lost. In a live environment, this would cause a failure and harm the production process.

Full system test.

What was now required was a full system test. That is, all the re-engineered code, the front-end interface (developed by another engineer), and communication with automated cells. However, this can only be achieved by running the system in the production environment. One shift had been allocated for such testing. This meant that in this case, the system test, installation test and acceptance tests were merged as one. The acceptance test would ran under a trial procedure for a period of a month.

No errors or failures occurred during the installation of the re-engineered production line controller system. Each automated cell was tested individually and the message queue was monitored at first and then the system was left to run throughout the remainder of the shift. Another experienced software engineer was present during this test as were production staff. Everyone was satisfied with system's performance and it

was agreed that it should remain in the production environment to run the line in place of the old system.

Acceptance test.

A trial form was signed and a one month period was allowed for live system testing. This was sufficient time for the system to experience real-world conditions as opposed to the simulated data that was inputted during module and integration testing. Some minor changes were requested by the user to alter the system's functionality. These alterations were made to remove errors that were data reporting issues, but they were not undermining the production process. The system was accepted formally through a change proposal (a change from the existing to the new, re-engineered system) which was signed and agreed by production staff and members of the other departments at the end of the trial period.

5.6. Leveraging.

The following points outline the lessons that have been learnt from the testing process improvement case study:

- A testing strategy was influenced by the time remaining to the project deadline, the cost of removing defects and the risks of any remaining defects to the user.
- Testing is an essential aspect of development of manufacturing systems, but other software process improvement initiatives must be undertaken to avoid defects from emerging in the early aspects of the project.
- When systems share the same principle design, a defined and automated testing process can allow for reusability.

The strategy of testing is very difficult to plan at the early stage of a project. For the two projects in this case study, time was allocated when initial plans were made, but it was not until the testing stage actually arrived that a testing strategy was formulated. Due to the general nature of development projects, changes are inevitable under changing circumstances. Hence, there may be less time for testing if implementation

has overrun, or management may declare a shortage of resources and budgets. All these conditions affect testing, but they are factors that must be considered to remove the defects that are of the highest risk to the users (or customers). Essentially, what has been learnt was that more than sufficient time should be allocated for testing. Objectives should be defined that must be achieved within the allocated tested period so that the developed system will possess minimal risk to the user.

In an environment such as P.C.W., project managers will be pressured to have all requirements implemented within the deadline, relying completely on the testing phase to remove all risks of failures. For software process improvement, the drive for more reliable software should come from the very start of the project. Kit [Kit, 1995] suggests that testing must start as early as possible. He states "test early and prevent defect migration [Kit, 1995]". That is, if there is a defect in the requirements definition then it will be designed and then implemented leaving the testers to spend time and effort amending not only the code, but also all documentation. For this project, automating the re-engineering process as, explained in chapter 4, eliminated the risk of producing an error. Hence, there is evidence of improvement to the software practice.

For the winding machine data collection system project (Chapter 6), the same scripts were used but were altered – in terms of test data - to meet the testing requirements of the system. The same testing process was used, that is, module or unit testing was conducted in the same fashion as the previous project, however, the full system tests were undertaken in a three-phase approach:

1. Two winding machines were connected to the data collection system to prove the concept. This is regarded as the full system test and installation test.
2. One whole production area consisting of twenty-five winding machines were connected to prove networking reliance and to resolve any reporting issues. Production staff co-operated to make the reports as accurate as possible.
3. Factory-wide implementation of the winding data collection system then followed. This was regarded as the acceptance test by the production staff where the system is formally 'signed off'.

Both systems – production line controller and winding machine data collection - continue to successfully supply the MIS system with production data through automated processes. Production staff have access to production level reports and managers have instant access to higher level trends-analysis reports as never before.

5.8. Summary.

For this case study, one project was chosen for a software process improvement initiative that focused on better testing techniques. The systems tested were essential to the manufacturing process as the risk of a defective system would be high causing harm to the production process. Hence, the business case for implementing testing thoroughly was very strong.

Testing was possible at a low level unit testing where functions are verified internally. Then functions were integrated and their behaviour was closely monitored to find any defect that could cause interruption to the production process. This was also preparation for the full system testing where the systems were executed in a 'live', production environment. This approach to testing proved successful as the system remained in the production area providing process and production data to the production staff.

One of the benefits of investing in testing on this project is that defined practices and test cases could be 're-used' in subsequent projects. Over time, this would lead to a more efficient testing process. In this case study, UNIX shell scripts have been used to automate the testing and whilst it is time consuming to create them, they will save considerable amounts of time when they are re-used.

Chapter 6

Case 3:
Automation of data collection from winding machines.

6.1 Introduction.

Research into software engineering has shown that by improving the software process, there can be an improvement to the software quality. From a business point of view, the goal is to reduce the cost of software development and maintenance. In [Boehm, 1988], Boehm & Papaccio state that one way of controlling cost is to optimize our software development and evolution strategy around predictability and control.

Defining a software process for a project allows developers to take a 'step back approach'. That is, they can question the essential technical issues that are otherwise omitted in Gantt charts or classical approaches to development. Processes can be designed to plan the project in terms the activities that need to be undertaken to achieve the given project specification.

The hypothesis for this case study is:

Software process modeling will help to facilitate a more predictable or manageable systems development project process for projects.

6.1.1. The business case.

Humphrey [Humphrey, 1989] has proposed a maturity framework for software process improvement that can support Boehm & Papaccio. This framework, the CMM, has five levels that can be used to identify the levels of process control in an organisation and has been in described in Chapter 3.

There is no particular level on the CMM model where P.C.W. can placed. At P.C.W., company standards and procedures have allowed for some areas of software development to show level 1 characteristics such as the use of source code control for change management. Gantt charts are used for planning and communication for project issues.

The use of Gantt charts has been a long-standing custom and its use is fundamental for any project that is undertaken in the company. This shows a repeatable characteristics, that is, level 2. Other examples of Level 3 characteristics are the use of methodologies and tools, plus a new systems creation procedure has been defined. However, there is little to suggest that P.C.W. can be placed in the Level 4 or 5 category, as there are no process measurements and there are no efforts for continuous progress through software process improvement. Foundations for this managed stage can be laid by defining the processes that will achieve the project specification. For example, in level 3, one of the controls that an organization must have in place is a defined process architecture. Humphrey states that

'While software processes models may be constructed at any appropriate level of abstraction, the process architecture must provide the elements, standards, and structural framework for refinement to any desired level of detail [Humphrey, 1989]'.

There are many types of models that are used in software engineering. Software lifecycle models provide an overview for the development process, whilst maintenance models provide a framework that isn't covered in most lifecycle models. However, it is recommended that organisations develop and continuously monitor their software processes by using models that are specific to their process requirements.

6.1.2. The technical case.

In chapter 2, programming processes was discussed. It provides a means of tailoring software processes to suit the needs of an individual project. Like software programs, programmed processes can be 'compiled' and 'executed'. Any errors at 'run time' can

be amended, or 'debugged', in the next execution of another project. Software process modeling is a technique that enables process programming.

Process modeling allows the developers to understand how they intend to approach a development project. That is, they can be used as part of the planning process. It is also useful for documenting previous approaches to projects that might explain their factors of success and failure to the project team. This contributes to software process improvement.

6.2 Initiating.

For this case study, the specific details of the data collection project will not be discussed, but more importantly, the issues surrounding the use of process models at P.C.W. will be illustrated. The following statement briefly explains the purpose the proposed data collection system:

The system must automatically load the winding data from all the winding machines into one common database that will be used by the MIS system to generate higher-level production information.

The proposed winding data collection system aims to support MIS in providing higher level production and performance information. The customer for the system is MIS and the users are production staff and management. The system will economize data collection by way of automation where data is extracted directly from the winding machines and stored into a database. There will be no human interaction in the process, it will be 100% automated. The data that is sent in must meet the MIS requirements, which in turn, are the user requirements. For MIS the system must collect and store the as much of the winding data that is possible automatically.

The data will be stored in a relational database that will communicate with the MIS front end. MIS will be responsible for providing the reports that are currently produced by the line controller [Mistry, 1999].

The above paragraphs show how important the winding data collection system is to the daily production process. It will supply production staff the necessary information to control winding coil production, hence, this system is both urgent and it must be reliable. At P.C.W., it is routine to prepare Gantt charts and financial estimates before the projects start. However, such documents are only useful for showing what is to be done i.e. the project specification. To ensure that the project will meet its specification, efforts must be made to understand and design how the project will meet its specification.

The next section discusses an approach that shows promise for designing a software process that will enable developers to devise a strategy for undertaking a project, such as the winding machine data collection project.

6.3. Diagnosing.

Software process modeling allows the developers to express how project objectives can be achieved through a series of predefined activities. The following section discusses software process models and how they can be developed using a hierarchy to present process designs.

6.3.1. Software Process Modeling.

Software processes should be dynamic to accommodate changes that need to be made if the process is improved. Organizations should have a model of their process that is tailored to show what should be done or how a process should have been done.

6.3.1.1. Descriptive and Prescriptive Models.

There are two types of models that can be used in software process modeling: a *descriptive* model and a *prescriptive* model [Humphrey, 1989]. A descriptive process model explains the activities that are undertaken for a process. This is then used to identify where improvements can be made and a new prescriptive process model is drawn that shows how the process should have been done. This is particularly useful

for future maintenance tasks in terms of time, cost and effort analysis, and also training. As maintenance tasks are undertaken the processes should be continuously reviewed for improvements there by increasing productivity and effectiveness of the maintenance process.

6.3.1.2. Software Process Model Hierarchy.

Dowson Dowson, 1985] suggests that models should be viewed critically and questions must be asked, such as, do they correctly describe how software is really built? Does it really correspond to how the software should be built? Does it describe the whole process and is it useful? After all, the model will be used to communicate what will be done in the software development or maintenance project to other people. Hence, it is important that the model is a concise yet useful to members of a software team.

Dowson suggests that a hierarchical structure to process models where a meta-model is at the top of the hierarchy and is the basis used to express the process models. The meta-models below are the various software process models that are different representations of a software process. Then each software process model will have different approaches to undertake the activities in the process i.e. a methodology. An example given by Dowson [Dowson, 1985] is explained below.

Figure 1 shows a very simple process model. Note that there is no indication of how the transformation will occur.

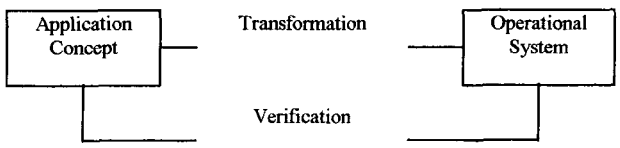


Figure 1. A simple process model.

There is no definition of what the rectangles and arrows mean and it is not particularly clear what the developers must do to transform the application concept to an

operational system. An approach must be taken for the transformation from application concept to the operational system. A simple approach would be as follows

1. to represent the application concept in a language (e.g. a requirements list),
2. to produce a design of the requirements for the operational system,
3. to Verify the design to ensure that the requirements of the users are met in the final system.

Figure 2 shows a more refined software process model.

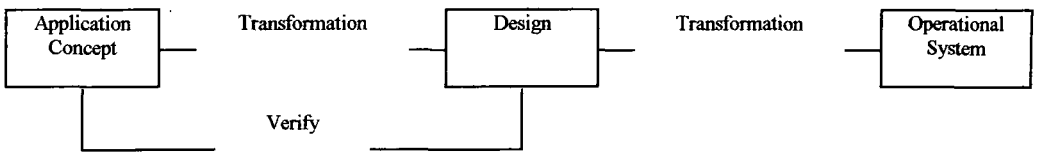


Figure 2. Next level hierarchy of the simple software process model.

The model in figure 2 shows a more detailed process than the model in figure 1. However, this model does not conform to the model in figure 1 because there is no verification from design to the operational system. Perhaps, there are other approaches that need to be explored that will transform the application model into the operational system. Hence, Dowson shows that a hierarchy can evolve as in figure 3.

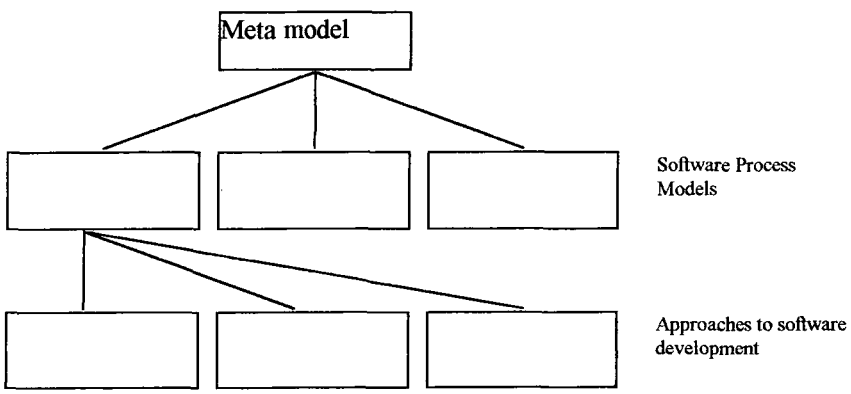


Figure 3. Model Hierachy.

The hierarchy shows that from one meta-model (at the top) there should be several representations of a process. A decision has to be made as to which process model to use for the transformation of application concept to the operational system. Then, once a process model has been chosen, an approach has to be chosen that will complete the process productively and cost-effectively. This suggests that several attempts at

modeling the software process will be required and that the choice of which process to use will be decided by several members of the software team.

6.4 Establishing.

Humphrey [Humphrey, 1989] has shown that improving the software process can improve the product. There are five maturity levels that each require certain level of control and measurement. In terms of the CMM, P.C.W. could be classed as a Level 2 (Repeatable) organization where Gantt Charts were used to plan and monitor projects.

By defining a process and modeling it, software engineers would be able to communicate and coordinate the technical aspects of a project. P.C.W. would then show Level 3 (Defined) characteristics where process models provide a better understanding of the process. Once the processes had been defined, creating a descriptive model (to show what has been done) and then a prescriptive model (to show how that process can be improved) could improve them. It is suggested in [Finkelstein, 1994] that the process model should be of an imperative nature, but more generally it will be less stringent, describing the tasks that need to be carried out and the constraints that should be observed.

Several models may be created, out of which, one is chosen that best describes a process. This model is then used to understand what approach must be taken to accomplish the activities that are defined. In order to move to Level 4 (Managed), the company would have had to analyze and measure the defined process to make improvements for future projects.

6.5. Acting.

A software process model had been designed for the winding machine data collection project. The final draught is shown in figure 4. This model shows the activities that were necessary to meet the project specification. It is a prescriptive development model and was been used to discuss pre-project issues related to planning with the I.T.

Several draughts of software process models were drawn in the course of discussions with users and the I.T. Manager. The final draught given here has been drawn up using a documentation tool and was used in conjunction with a Gantt Chart for the duration of the project.

Several observations were made with respect to the use of process models. Production and senior managers were only interested in the project specification, therefore, only using the Gantt chart. They are not interested in the technical issues unless they directly related to managerial issues .i.e. extra cost, time saving, etc. These issues are communicated through Gantt charts. Users of the system were the same, having no interest in technical issues. In general, process models were of no use to non-technical people, only to developers of the software system.

Activity 10 in the model shown in figure 4 – "Test Software" – was broken down into another process model. The testing process that was used for testing the production line controller software (Chapter 5) was reused and implemented again for this project. This was possible because the two systems work on the essentially a common model, which meant that a successful process was "executed" again to ensure that this system would not pose any risk to the business.

6.6. Leveraging.

Defining and modeling software processes for this particular case has taught the following lessons:

- process models provide coordination in a project, but must be used in conjunction with Gantt Charts.
- process models are dynamic. That is they must evolve with changing circumstances.
- process models can be executed again in a different context.

Companies such as P.C.W use Gantt Charts to communicate project information. With automated tools, short and long term plans were created effortlessly allowing very

detailed information to be presented. At P.C.W., most project meetings are rarely held without the presence of Gantt Charts to monitor progress. They were used for complete co-ordination throughout the project duration. Software development projects were no exception. Hence, implementing process models provided a fresh approach to project preparation, but its usefulness could only be fulfilled if managerial issues were also considered. [Finkelstein, 1994] supports the notion by stating that a process model not only describes that technical activities to be carried out in the development of a system, it also describes the managerial meta-process as well.

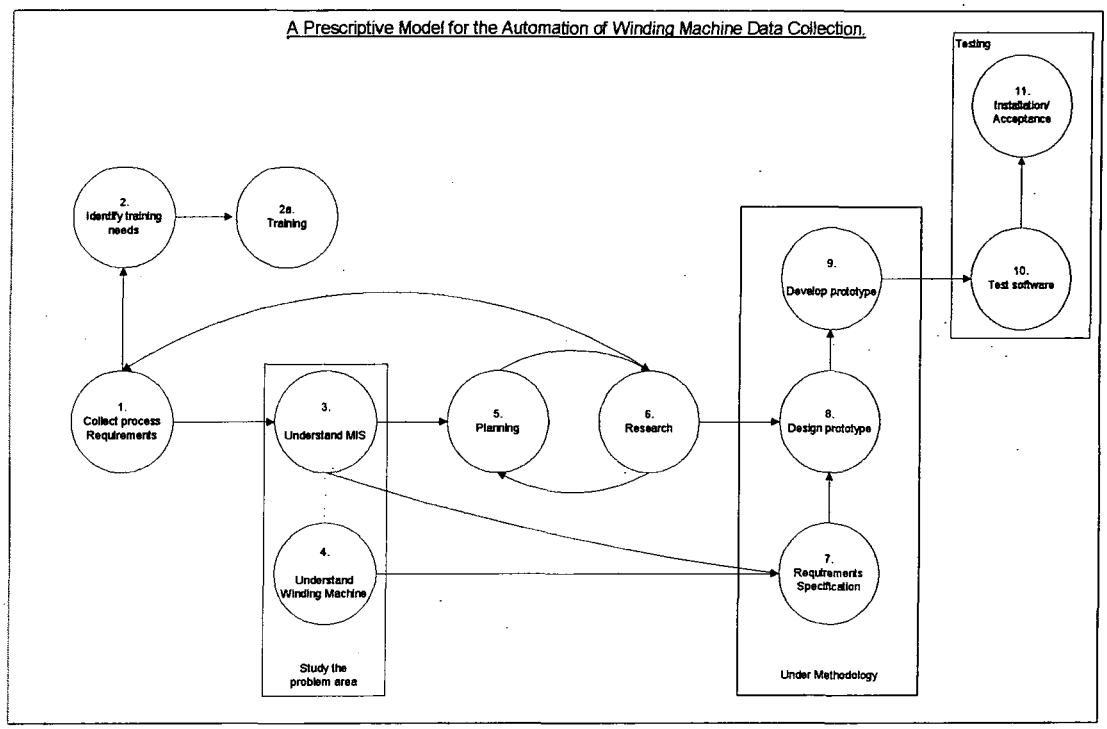


Figure 4. The overall process for development of the Winding Machine Data Collection System.

The testing process used for testing the reengineered production line process (chapter 5) was used again as part of this process. Time was saved because a process did not have to be created from scratch. Furthermore, this testing process has already proved to be successful and so it seemed appropriate to apply it again. It has been demonstrated here that process models can be used within other process models. The process for this data collection system could be used again as part of another development project, or it might be used as a basis for a new process design. In either

case, the development team would have better control over a project, as the process model will allow for more predictability and better management.

6.8. Summary.

Software process models must be tailored for individual maintenance project needs. Before models can be drawn there must be a clear goal for the development project. The activities defined within the process should aim to achieve that goal collectively.

This case study has demonstrated the usefulness of process models and how one specific model can be used in a different context. Developers will be more prepared for circumstances as a project progresses because descriptive process models will present a previously executed process. Process improvement may be initiated here as developers can change a descriptive process model to remove any failures experienced before, whilst keeping the successes, in a new prescriptive model.

Chapter 7

Case 4:
Investigation of requirements gathering for new I.T. systems, primarily the MIS project.

7.1 Introduction.

When a customer requests a new system for whatever the purpose, it is absolutely essential that their requirements are completely understood and accounted for during the development process. Failure to meet the customers requirements will result in an incomplete system and further costs in post-production maintenance.

Research conducted by Taylor [Taylor, January 2000], supports the fact that failure to appreciate the importance of good requirements management will cause I.T. projects to fail. In a survey of 1027 projects only 130 were successful. When managers were asked at which stage had their projects failed the highest frequency answer was the 'requirements definition' stage. One of the key problems in these projects was the assumption that once requirements were documented then there would be little significant change. Taylor suggests that management of expectations is poor and the approach of full requirements definition followed by a long gap before those requirements are delivered is no longer appropriate.

Requirements definition is an essential process because it is the preliminary stage of systems development. Hence, if errors exist in this process then they will be present throughout the development cycle. The case study presented in this chapter discusses the software process improvement initiative at P.C.W. by way of designing a new requirements process.

The hypothesis for this case study is:

Introduction of new requirements process will allow developers to understand and satisfy user requirements for new systems.

7.2 The business case.

By improving the requirements process, the systems development can also be improved as the deliverables are clear from the outset and measures can be taken to control any changes that are specified by the user. Time, money and resources are essential to a business, especially in manufacturing where delays in production schedules can result in losses. Hence, if a system is delayed by ad hoc changes to requirements or late implementations of new requirements then the rest of the manufacturing process can suffer.

Also, there is little expense in devising such a process, unless automation tools are purchased to facilitate requirements documentation and configuration management. Time will be spent in researching new methods and techniques, followed by implementation of the findings of the research.

A new process will result in a new procedure to ensure that requirements are dealt with a controlled and consistent manner. Such procedure will lead to effective and efficient business control that, in turn, will lead to better business performance.

7.3 The technical case.

A new requirements process can affect the development of new systems. Developers must learn a new method of working and must follow new procedures that will contribute to understanding the customer's requirements.

It is often not the developer's fault when systems deliverables are late or over budget. They must deal with the changing needs of the customer who sometimes do not know exactly what they want. This can lead to requirements change at later stages of the development or new requirements being specified, both cause considerable disruption and delay. Developers need to be more prepared for such circumstances to enable their role in the development cycle to more productive. Ad hoc changes and implementations of requirements should be replaced with control and management in order to deliver what the customer desires within the budget, time and resources.

7.4 Initiating.

The Manufacturing Information System (MIS) project had faced considerable problems that caused schedules to be extended. A fundamental reason for this was the lack of agreement over the MIS requirements between the developers and the customers. The main causes of prolonged schedules had been:

- Extremely vague and general description of the required system.
- Requirements were interpreted differently between customers and developers.
- Customers of the new system had changed during system development that caused requirements to change or be redefined.
- Customer were not involved enough from outset.
- Data collection and entry logistics had not been considered.
- Conflict between requirements that were also 'volatile' i.e. extremely susceptible to change.
- No clear understanding of the inputs and outputs of the system in the business process.
- The required systems end-users and functions had not been clearly defined.

In general, for such a large-scale system the requirements of the system were clearly insufficient. The Rapid Application Development (RAD) approach was used. This encouraged system development with user input, but it was found that the customers gave insufficient feedback for the technique to work effectively. Hence, there should have been more collaboration between developers, customers and end-users. What was needed was a mechanism that improved the way requirements were gathered, the way they were documented and agreed upon, and the way they were managed when requirements of a proposed system changed.

This case study aimed to provide a standard for requirements process that could be used for the development of any new system. There were three main issues for discussion in this case study and they were as follows:

- requirements collection: The method of collecting the requirements of the desired system.

- requirements documentation: The qualities that had to be considered when documenting the user requirements specification.
- requirements evolution: The management of requirements of a system after its installation phase.

7.5 Diagnosing.

The Capability Maturity Model (CMM) has made a considerable impact on software engineering since it was introduced by Humphrey [Humphrey, 1989] in the late 1980s. It defines five successive process improvement levels – initial, repeatable, defined, managed and optimise. Somerville [Sommerville, 1997] has incorporated the CMM into the requirements engineering to explain how an organisation can mature its requirement process. The following paragraphs describe the first three levels of CMM based on good requirements process practices.

Level 1 (Initial Level) – At this level ad-hoc requirements process is employed where no methods or techniques are used to produce good quality requirements documentation. The organisation relies on the skills and experience of their engineers' requirement elicitation, analysis and validation.

Level 2 (Repeatable Level) – At this level the organisation has defined standards for documentation and management. Some tools and techniques are used in the requirement process that produces better quality documents than level 1 organisations.

Level 3 (Defined Level) - At this level the organisation has employed a model of the requirements process that achieve good requirements engineering practices. They will continue to assess the processes and improve them.

It is likely that the level 3 organisation will produce high quality requirement documentation on time. However, it must not be assumed that it will not experience requirements engineering problems because it still relies on the ability and experience of people involved. There are also systems development issues such as time, cost and complexity of the problem. At the lowest level, the organisation will experience

problems due to ambiguous, incorrect and complex requirements that have been collected and documented without control and standards.

It is important for an organisation to implement standards and control over how the requirements are collected, documented and used throughout a systems development project. Without control and standards the system developers will have a different interpretation of the system from the customers (or users) resulting in customer dissatisfaction. Extra time and money that is spent on changing requirements at later stages of the development can be saved if all persons involved can understand and agree on the requirements before development starts.

7.6.1. Requirements Collection.

This process does not just involve asking people what they want; it requires a careful analysis of the organisation, the application domain and how the system is likely to be used. Effective requirements elicitation is very important. If the analyst does not discover the customer's real requirements, the delivered system may not be acceptable to customers or end-users [Sommerville, 1997].

The statement above is an extract from Sommerville [Sommerville, 1997]. Sommerville also states that in requirements elicitation the following problems must be addressed:

- Stakeholders often do not really know what they want from the computer system except in the most general terms.
- Customers express requirements in their own terms and with implicit knowledge of their own work.
- Different stakeholders have different requirements and they may express these in quite different ways.
- Organisational issues and political factors may influence the requirements the requirements of the system.
- The economic and business environment in which the analysis takes place is dynamic.

The following paragraphs prescribe guidelines for requirements collection taken from [Sommerville, 1997] that should be considered in order to devise a standard, requirement collection pro forma.

Identify and consult system stakeholders. These people are the end-users, customers, managers, and developers of the system. Talking to the stakeholders will provide several viewpoints of the system, will involve them in the requirements process and will provide specific, and perhaps essential, requirements to be implemented.

Record requirement sources. The source of a requirement may have to be contacted in order to clarify or validate a requirement. There must be some way of finding the source of the requirement because they are the people that must be satisfied through out the system development process. It is important that the role of a person is recorded because the people may leave or change jobs in which case their names are particularly unhelpful. Where a group of people are the source of a requirement one of them should be designated as the principle source.

Define the systems operating environment. Here there should be a record of the systems, platforms, software and hardware that will interact with the proposed system. This should lead to fewer installation problems and reveal some constraints that may be considered during design.

Use business concerns to drive requirements elicitation. The requirements will aim to satisfy specific business needs. Hence, these needs should be recorded in order to understand how the system will contribute to the business.

Look for domain constraints. These are requirements that will impose restriction on the system. They must be identified because there may be regulatory, health and safety or political reasons for the constraint to take place. Further time and effort will be spent on collecting them late into the development process.

Record requirements rationale. It is important to understand why a requirement is specified, i.e. the link between the requirement and the problem. The rationale justifies the requirement in terms of the problem that the system is trying to solve.

Collect requirements from multiple viewpoints. Requirements should be collected from many perspectives of the system. This may mean the end-users to managers so that a wider coverage of the requirements will be obtained.

Define operational processes. If the proposed system is to support a business process then this process should be defined to reveal the process requirements and the requirements constraints. This can be a surprisingly complex task especially of the process that the system must support is new to the business.

One way of collecting this information consistently over a number of projects is to use checklists. However, developers must address not only system requirements, but also system development requirements to ensure a smooth development process with the customer/user. In the requirements collection process it is essential that both parties agree on such topics as customer, size of the system, time scales, customer training, etc. A checklist should be used by the developers before or during the requirements collection stage to list the system development and system requirement issues. Questions that address those issues should be dealt with and an agreed set of terms should be 'signed off' by both parties.

After the system requirements have been gathered they will be analyzed in order to remove conflicts, inconsistencies, to prioritise and most importantly to create a draft of the requirements documentation. The next section discusses requirements documentation.

7.6.2. Requirements Documentation.

A standard document structure encapsulates what the organisation thinks is the best way to organise a requirements documentation. If a standard is to be useful, it must reflect the best practice in the specific organisation for requirements documents

[Sommerville, 1997]. This section describes the purpose of the requirements document and the qualities it should possess.

From a development perspective, an requirements document should serve to [Davis, 1993]

- communicate requirements among customers, users, analysts, and designers,
- support system-testing activities, and
- control the evolution of the system.

A requirements document is used to express what the system is to do - its functions and operation – to all parties that are involved in the system development. These are the users, customers, the analysts and the designers who will all use the requirements document for their part in the development project. Hence, the requirements document must be precise in its interpretation of the requirements, but flexible in the notation that is uses to express how the system is to behave. It is important that all parties must have the same interpretation of the system requirements so that the end product will satisfy all the users' requirements.

Once the system is built, or prototyped, it will be tested to ensure that all functions work correctly. The validation and verification of the system comes from testing to see if the requirements have been met. Another reason for the requirements document to be accurately interpreted is so that they can be tested properly.

Once the system has been tested and accepted it will be maintained. That is, its requirements will change due to environmental changes and new user needs. Hence, the requirements of the system will change and when this happens, the requirements document is referenced to find out if it is a new requirement or if it is an update of an old one. The requirements document must be kept up to date for it to be useful through out the system's lifetime.

7.6.2.1. Attributes of a well-written Requirements Document.

Although this research is concerned with a requirements document that is not specific to software systems projects only, this section about the attributes of a requirements document can be applied to any generic form of requirements specification. The following paragraphs explain the attributes of a good requirements document as defined by Davis [Davis, 1993]. He has listed the following attributes that would make a perfect requirements document. They are as follows and are explained in the following sections:

- correct
- unambiguous
- complete
- verifiable
- consistent
- understandable by the customer
- modifiable
- traced
- traceable
- design independent
- annotated
- concise
- organised.

Correct.

A correct requirements document should satisfy the user's needs. If the user requests for a summarised weekly report, but the requirements document insists on giving seven daily reports for quicker implementation then the requirements document is incorrect. The requirements document of a new system should set out to meet all user requirements and should be implementation independent.

Unambiguous.

The requirements stated in the requirements document should have one interpretation only. The developers and the customers must have the same interpretation of the requirement otherwise a system will be designed to meet what the developers 'thought' the users wanted.

Complete.

In a complete requirements document, everything that the software must do should be stated. There must be no missing requirements such as 'To be determined' or 'not yet been clarified'. There must also be completeness in that all data flows and outputs must be defined.

Verifiable.

An ambiguous or non-measurable requirement cannot be verified. An example of both would be 'must have a user-friendly interface'. How can this be verified, what determines a 'user-friendly interface' and how will it satisfy the user? Hence, it is important that all the requirements in the requirements document must expressed in a verifiable form.

Consistent

The requirements must be consistent with each other and any other document, or rather, there must not be any conflicts in requirements of the system. Davis [Davis, 1993] has stated four types of inconsistencies:

Conflicting behaviour: where one action made by a user is specified as having two different responses.

conflicting terms: where two different terms are used to mean the same thing.

conflicting character: where two parts of the requirements document demand the product to exhibit contradictory traits.

temporal inconsistency: where the requirements document demands the product to obey contradictory timing characteristics.

Understood by Customers.

Requirements documents may be written in various notations in order to facilitate communication of requirements with other engineers. However, these notations are not suitable for communications with customers who may be operators and management and will be less likely to be familiar with anything other than natural language. Where computer science notations are used there should be a suitable method of translating it into language for communication with the customers.

Modifiable.

An requirements document will undergo many changes in order to make it complete, consistent and valid. It is essential that the requirements document is written such that the changes to it are made easily, completely and consistently. The requirements document must be structured so that when a requirement changes the parts of the requirements document that have to be updated can be easily found. This is achieved by a contents page, an index and cross-references where necessary.

Traced.

It may be useful or necessary for the some to know the requirements origin. It may be that if the requirements were ambiguous then the engineer would want to talk to the person who specified the requirement. Other documents that are related to the requirements should be referenced so that the requirements can be traced.

Traceable.

The requirements document must be written such the requirements are traceable from the proceeding phases of system developments. For example, when the system undergoes testing the software engineers will want to know which requirements they are trying to fulfil. Hence, the requirements document will be referenced, so it is

important that there exists a method for finding the requirements. Traceability can be achieved by numbering the requirements or the paragraphs that contain the requirements. The effort taken in making requirements document traceable will facilitate the use of a requirements traceability matrix (RTM) which is used during design and testing to ensure that each requirement is satisfied by design and testing.

Design Independent.

A requirement in the requirements document should capable of being satisfied by several design alternatives. There should be little or no constraint on how a requirement is designed or implemented. However, there may be a case where a system is intended for a particular platform or hardware and that indicates that some requirements will have restrictions.

Annotated.

All the requirements listed in the requirements document will have the relative amount of importance over each other. For example, the requirement 'system must shutdown immediately if fire is detected' will be a more critical requirement than 'system must display time with am/pm indicator'. Hence, in the requirements document the requirements should be labelled as being essential, desirable or optional. An optional requirement would be a function that was not critical to the users but would be useful to have. By annotating the requirements the developers can place some order of priority when designing, implementing and testing the system.

Concise.

The requirements document should be simple in its layout and notation. Note that an requirements document will be read by non-technical people and so there must be some leniency over complexity. The requirements document should state the software requirements concisely.

Organised.

The requirements document should have the requirements of the system organised such that they are easy to locate. This is important particularly when the system is large with many requirements to list.

There cannot be a requirements document that can satisfy all these attributes because trying to abide to one attribute will disobey another. If the requirements document is tailored to the unambiguous and consistent attributes then the reduction in natural language notation will make it less understandable for the customers. Hence, there have to be compromises made when writing an requirements document. The conclusion reached in Davis [Davis, 1993] is that 'There is no such thing as a perfect requirements document'.

7.6.3. Requirements Evolution.

During system development users may suggest new requirements or existing requirements will change. Requirements evolution is concerned with managing system requirements as these changes occur in order to keep the documents updated and to understand the impact of the changes to the project. If requirements are not managed properly then the project could extend beyond the intended schedule resulting in extra cost and effort. The following guidelines are prescribed by Sommerville [Sommerville, 1997].

Uniquely identify each requirement. Each requirement should be uniquely identifiable making it easier to reference and distinguishing it from any other requirement. With unique identifiers there can be considerable reduction on ambiguity between requirements and greater traceability of requirements from later stages of development. Configuration Management principles may be applied for improved management of requirements. Such an application would place an organisation at level 3 of the CMM model of Requirements Engineering.

Define policies for requirements management. Standards and procedures should be used to discipline people and to ensure that everybody involved in the project maintains the policies that effectively manage the requirements. Procedures will help people to understand the current process and allow developers to maintain control because each defined step will be followed consistently by all persons.

Define traceability policies. Requirements will have relationships between one another. Many will actually depend on others and this can prove to complicate requirement evolution because changing one requirement may have an impact on others. Hence, it is essential to have a system that will interpret these relationships in order to ensure higher quality and stronger control over requirements evolution that will result in a complete system implemented for the users.

Use a database to manage requirements. This is particularly useful for large-scale system development that concerns high volume of requirement data. Applying unique identifiers and maintaining links can be facilitated with a database.

Define change management policies. These polices are concerned with controlling and authorising change to requirements through a formal mechanism. By controlling change there is a control over costs and time because both factors are monitored as part of the process.

Identify global system requirements. These requirements influence the whole system, as opposed to a requirement that may affect only one function of the system. Hence, global requirements are susceptible to high cost if they are changed. By identifying such requirements the analysts can pay considerable attention to them in order to understand the scale of impact in terms of effort and cost if there were changes to the requirement. High costs are due to the fact that global requirements are complex because they will effect several parts to the system that would require extra effort and time to change.

Identify volatile requirements. Volatile requirements are those requirement that are most likely to change. If these requirements are identified then development can be

tailored to prepare for possible changes i.e. software modules may be written to possess loosely coupled properties so that a change will not effect the rest of the system.

Record rejected requirements. It is common for rejected requirements to be proposed again during system development and by saving the rejected requirement and recording the reason for the rejection analysis time can be saved.

These guidelines can be implemented more effectively if the requirements documentation show as many of the attributes that were mentioned in the previous section as possible.

7.7 Establishing.

The previous section discussed the three issues that were the focus for the requirements process improvement - requirements collection, requirements documentation and requirements evolution. Consequently, the following tasks were undertaken:

- a pro forma was formulated for requirements collection with user guide (both in Appendices A1, A2 and A3),
- a template was created for documentation of requirements with a user guide (in Appendix A4) and,
- a procedure was defined to manage requirements (given in Appendix A5).

Developing a standard set of forms promoted a consistent survey where all customers/users would be asked the same questions. In a business environment such as P.C.W., many people do not appreciate the importance of system requirements and treated the matter with low priority. Hence, it was essential that the forms were easy to fill out and presented little bureaucracy. At the same time, it had to guide the customer/user in providing the right information that showed his or her view of the desired system and its role in the business.

A standard template had to satisfy as many attributes as possible, but most importantly, it had to satisfy the developers as they must use it to understand the desired system as conceived by the customer/user. A template simply allowed reusability and consistency throughout the requirements process.

For requirements evolution, a procedure had to be developed such that a standard practice could be ascertained by those involved in project. If everyone followed the same, predefined steps to control the changes that were made to requirements then there would be a greater chance of producing a desired system accurately. The procedure had to be useable, that is, if it was too specific then it would be difficult to control. If it was too vague, then control would be very loose and mistakes were likely too have occurred.

User guides and instructions for each part of the new requirements process were also written in order for developers to fully understand and appreciate the structure of the process (shown in Appendix A).

7.8 Acting.

The forms and templates discussed in the previous section were been created in a "prototype" manner. That is, the forms that are presented in Appendix A are final versions of the several draughts that were sketched. Members of the I.T. department contributed ideas and requirements that were implemented in each new draught until the final version satisfied all members. The next few sub-sections briefly show how the forms had evolved.

Using the guidelines presented in section 7.6.1., an initial draught of the requirements collection form was sketched. Members of the I.T. department stressed that this collection form aimed at obtaining the various perspectives of the desired system from the different users. These perspectives would allow the I.T. department get a clearer understanding of what the different users actually want and why, as each user has requirements that support their own role. The first version of the form (Appendix A1) asked from the user:

- The user's name and a description of their position in the company.
- A list of requirements each justified by the contribution it makes to the business process.

Further discussion led to more changes as members of the I.T. department wanted to understand the context of the desired system, what data requirements it will have and any constraints that must be imposed. The following were applied to the form to support these requests (shown in Appendix A2):

- A context diagram that allowed the user to show the other entities that the desired system will communicate with i.e. other systems, other computer systems, departments, people, etc.
- A box to list the items of data that must be stored and calculations that must be made to them.
- A box to allow the user to list the constraints imposed on the system i.e. security and access.

More discussions and trials of the form within the I.T. department lead to more revisions of the requirements collection form. Suggestions were made about performance of the system, the business benefits that must be obtained, and the process that the system must support. Thus, the final revision contained the following (shown in Appendix A3):

- A box to allow the user to specify performance requirements i.e. data must be updated every five minutes.
- A box to allow the user to show the business benefits to be obtained, or to explain the feasibility of the desired system.
- A box to allow the user to draw or discuss the business process that must be supported.

More revisions could have been made, but the I.T. manager was satisfied that the latest revision would facilitate the collection of new system requirements. It was also decided that new revisions could be made after the forms were used in real, business situations, as this would most probably highlight any significant problems.

The requirements documentation template was created in a similar fashion, except that the members of the I.T. were less involved with this form. Hence, the form was created using the attributes for a well-written requirements document discussed in section 7.6.2.1.

The next stage was for implementation of the requirements process on a systems development project. Such a project was proposed and it was intended that the new requirements process would be used to understand the needs of the customers.

However, discussions about the project between customers and developers caused considerable delay. Customer requirements began to emerge through these discussions but the forms and templates were not used at all. It seemed that the urgency to consolidate and settle the general issues surrounding the new system discouraged the I.T. department to run a 'live' requirements test on the process. There was little chance of an opportunity to implement the new process. Following the submission of the completed draft of the requirements process, it was becoming apparent that the company would not be investing in new systems development. In fact, the process was not implemented at all during the TCS programme.

7.9 Leveraging.

The following points outline the lessons that have been learned from this case study:

- Requirements process improvement must be tuned to the needs of the business.
- Implementing new procedures is difficult without sufficient support from peers and superiors.
- Medium sized businesses do not appreciate new methods and techniques to alleviate systems development problems.
- The feasibility of a specific software process improvement initiative must be determined before it takes place.
- And changing circumstances on a company may invalidate earlier feasibility studies.

There can be no standard way of collecting and managing requirements. Every development team must define a process that will provide sufficient information for

understanding and sharing the customers vision of the desired system. Guidelines can be used when designing such a process. However, they must assist the process creators to improve their ability in requirements engineering, blending it with the company's working culture and business objectives. For example, at P.C.W. the manufacturing culture expected results immediately and provided very vague, very general requirements leaving the I.T. department to their own devices until the system was delivered. The requirements process was designed to encourage the customer to express more by asking more people more questions with little bureaucracy. Other companies may be able to afford more time and can use it to negotiate with the customers at a more interpersonal level.

Great support is required from peers and superiors when any new procedure is submitted for approval. Procedures can change the way people work, often affecting traditions, standards and habits. People are known to resist change. Hence, it is essential that people that are involved in promoting new techniques and new ideas must share the same vision, goals, and enthusiasm with all persons affected. This includes superiors as well as peers. With this support, the new procedures would not only have be approved, but applied with a high degree of verve allowing the process to be improved undeterred.

Companies such as P.C.W., that use software as a service to a business process do not fully appreciate the value of new techniques and ideas. Perhaps, with more persuasion through training and teaching these ideas, the management would provide greater momentum for them to be used in the factory. However, this requirements process improvement was implemented at a time when it became apparent that little software development would occur in the near future. The nature of the business has changed where new, complex products will be manufactured through a manual process.

Finally, it is important that a software process improvement program must be evaluated for future use and effectiveness. That is, its feasibility must be determined and the benefits to that can be gained need to be identified. This process improvement did not cost the company in terms of direct finance, but resource time that could have been utilised elsewhere. If a strong commitment from superiors has been made for the

requirements process to be implemented after it was designed then this project would have been feasible at least.

7.10 Summary.

It can be concluded that the requirement process can potentially be improved if the company applies standards and policies to record and administer the requirements for system development. Many of the problems that had been experienced during MIS development could have been avoided by following a new process created by the guidelines above. Prolonged schedules could have been avoided because: requirements collection would have involved many customer perspectives, requirements documentation would have been structured in a manner that was understood and agreed by the customers and the developers; and change of requirements could have been controlled by both developer and customers.

A new process was designed where standard pro formas would be used for requirements collection, a template would be used for documenting requirements and a matrix (Appendices A1-A5) would be used to monitor and control requirements change. These documents would be used to bring the customer closer to the developer and to give them more involvement in the system development process.

The main lesson to be learned from this is that strong commitment and support is required from the superiors and peers in order to have a new procedure approved. Once it is approved, it must be undertaken by all those involved with enthusiasm and with a sense of purpose for the procedure to improve the software process.

In this case, another significant lesson is that what appears to be a feasible software process improvement initiative may fail because of changes to a company's core business i.e. no longer carrying out major software development projects.

Chapter 8.

Results.

8.1 Overview.

In chapters 4, 5, 6 and 7, four case studies have been discussed. Each has a specific process improvement initiative and an associated hypothesis. This chapter reviews each case study and presents the evidence that supports each hypothesis. It will also address the issue of improved software practices, which is one of the main aims of this T.C.S. project as stated in chapter 2. In summary, the four hypothesises are as follows:

Hypothesis 1: Use of source code analysis tools will facilitate the process of re-engineering of existing software systems.

Hypothesis 2: A formal testing process can contribute to more reliable systems.

Hypothesis 3: Introduction of new requirements process will allow developers to understand and satisfy user requirements for a new system.

Hypothesis 4: Software process modelling will help to facilitate a more predictable or manageable systems development project.

8.2 Hypothesis 1.

Use of source code analysis tools will facilitate the process of re-engineering of existing systems.

Overview

The re-engineering of the production line controller systems for the Year 2000 problem was an extremely high objective in the factory. There were six systems that required re-engineering where all database transactions in the code had to be replaced with embedded SQL statements for new ORACLE 7 & 8 databases. Each system consisted of approximately 130 KLOC of code. For P.C.W., this was to be a very challenging software maintenance task especially with such limited resources and time.

All systems were re-engineered successfully and this was largely due to the effort made by one of the remaining developers of the original systems. With his in-depth knowledge of the system's software structure and functionality, very little time was lost in program comprehension and implementing modifications. The only tools that were used by this engineer were those supplied with the UNIX system i.e. VI editor and standard functions.

However, it was not possible for this single engineer to complete all the re-engineering work by himself and the author of this thesis was assigned to assist with re-engineering task. Research into software maintenance has allowed numerous tools to emerge that can facilitate such re-engineering projects. The case study for this hypothesis employed to such tools that were used to re-engineer one of the production line by the author of this thesis who was unfamiliar to the software.

Evidence to support the hypothesis.

This case study showed a successful employment of CASE tools that supported the process of re-engineering the production line controller system. The source code analysis tools facilitated the re-engineering process (see figure 2 in Chapter 4) because:

- the time to obtain and understand the functionality of the existing system was considerably reduced (facilitation of reverse engineering),
- changes that were required were easily identified and changed (facilitation of forward engineering).

One of the problems faced by software maintainers is the lack of documentation, or updated documentation that can provide much of the information about the system's functionality before any work is undertaken. Extracting the original system functionality and generating design documentation requires a lot of detailed code analysis that is very time consuming and expensive to produce. Evidence to support this came when another production line controller software was used to produce documentation manually. It took seven weeks to translate all the source code for

Auto-YAMA-2 took into high level structured language statement that expressed the code in a more meaningful format.

The source code analysis tools used in the case study allowed graphical representations of the code to be printed out which meant that updated, concise and accurate documentation was produced for any future maintenance, or training. Using this documentation, the maintainer can now also understand the functions of the production line controller's software. The whole reverse engineering process was facilitated because of the introduction of maintenance tools.

The tools have facilities to graphically represent the code i.e. Call graphs (Appendix C), and this allows the maintainer to understand the effects of making changes to the code. It also confirms which parts of the code are required to change to meet new or altered functionality. Without the tools the maintainer would have to open and close numerous files of source code in order to understand the details of one particular function. Text editors, such VI, can cause a lot of time to be spent searching for relevant the files, then searching for the relevant lines in the code. This slows down the time needed for the maintainer to develop a mental model of the code. With hyper-text style, rapid navigation the maintainer can see the relationships between procedures, functions and data structures more clearly. This breaks down the complexity often found in such large systems. HTML also increases the maintainer's confidence for identifying and making changes, but without this facility the consequences of altering code in a large system is very difficult to foresee. Source code analysis tools do considerably alleviate the re-engineering problems that are presented by the properties of complexity, changeability, and invisibility (discussed in Chapter 2) that facilitates the forward engineering process.

Conclusion.

Time and effort are considerably reduced due the automation of tedious manual tasks such as reading though source code files, writing documentation, and searching for possible modifications that are required. The tools that were used allowed more

reassurances that the all the required modifications did not alter the functionality of the system, nor did they cause further errors.

Hence, for this hypothesis, it can be argued that the use of code analysis tools by P.C.W. can facilitate the re-engineering of systems, especially for new staff. This use of CASE tools also contributes to the aim of improving software practices. At P.C.W., the I.T. department has made no investment into new software engineering environments. New environments that have automation facilities, graphical representations of source code, or navigation tools do improve productivity, efficiency and accuracy. This has been the outcome of this case study, which means that the T.C.S. project aim has been achieved here.

8.3 Hypothesis 2.

A formal testing process can contribute to more reliable systems.

Overview.

The main drive for software process improvement is to improve the end product and in this case study, the process aimed to remove any risk of failure before the user accepts the re-engineered production line controller system. Software testing is regarded as a crucial phase in development at P.C.W., but the practice has always been up to the developer. There was no defined process that could be used for controlled and methodical testing of other systems.

This case study focused on finding a suitable test strategy for a production line controller system at P.C.W. The aims of the strategy were as follows:

- to avoid any risk of interruption to the production process that will result in a loss in manufacturing time and money and,
- to create a process that can be reused, as all the software systems at P.C.W. were implemented using a common principle.

The constraints imposed on this strategy were the time remaining before the project deadline, the cost of removing the defects, and the risk of the remaining defects to the users. A bottom-up approach was used where each function was individually tested, then functions were gradually brought together in stages until a full system simulation was created. This approach ensured that any risk of failure to production can be identified and removed before the system is placed in the 'live'; environment.

Evidence to support the hypothesis.

Testing was a formality in all areas of manufacturing at P.C.W. where systems had to be verified and validated before they are placed in the live, production environment. The reason for introducing a formal process was to ensure that the testing phase successfully removed the most high-risk defects within the time and budget that remained in the project. A formal testing process would contribute to a more reliable system because:

- all the defects were removed in order of highest risk to lowest risk,
- testing was performed from the lowest level (unit testing) to the highest level (full system testing).

By eliminating all the high-risk failures, (i.e. where the system completely fails to execute or where essential processes malfunction) the harm to the production process was removed. Small defects would cause minor problems to production such as reporting wrong process data, but this, at least, allowed production to continue. By prioritising defects, the developers have the confidence to install the system knowing that small defects can be removed later without causing delay to the production process. Hence, this meant that a more reliable system had been developed and where defects still existed, they would cause minimal damage.

By testing from unit testing to full system testing, the developers will have performed thorough and methodical testing. The functional requirements are tested from a detailed white box to a general black box approach until a full simulation of the system is performed.

This process was also reused in another project (chapter 6 or hypothesis 3) as both systems work under a common model. Time was saved, as a new process did not have to be written. Automated testing used in this test strategy proved to be productive as shell scripts were used to speed up the rate of functional and integration testing. Reusability was promoted because the shell scripts were used for the testing of the data collection system. Only the test data had to be changed and this takes relatively less time than rewriting new shell scripts or performing each test manually. The next section will discuss the results of this in more detail.

Conclusion.

The strategy that was employed proved to be successful as no problems were experienced during the installation stage when the system was executed the production environment. The project deadline was not exceeded because the re-engineering process (chapter 4 or hypothesis 1) was implemented so soon before the deadline that sufficient time remained to execute the testing process. These results may not prove that all defects were absent. Some defects could show their presence in a specific situation where a failure could cause an error that was negligible.

The important conclusion that will summarise this case study is that for high profile systems, the focus of testing should be to remove the risk of failures in order of their severity where the most severe failure is removed first. As the deadline for installation approaches, the developers must assess the level of risk of the remaining defects and decide whether it is cost-effective to remove them. This method will also involve the users, keeping them informed of the state of the desired system before it is officially released.

It can also be concluded that defining a generic process such as "testing" will have benefits in the future. New systems or legacy systems that have undergone maintenance will have to be tested before they are submitted to the customers. A defined or prescribed process allows developers to follow a proven method. Furthermore, such a defined process contributes to the upgrading of software practices and raises the maturity in the CMM to level 2. Software practices have been improved

here, not for the individual, but for the I.T. department as a whole. Some members of the department could argue that their own methods of testing have been thorough and reliable in previous projects. However, a formal, defined process ensures that all developers follow the same activities in the process and this may inject a degree of discipline to some individuals.

8.4 Hypothesis 3.

Software process modelling will help to facilitate a more predictable or manageable systems development project.

Overview

Development projects at P.C.W. are co-ordinated in a manner that is very business orientated. Time, budget and resources are the key factors by which progress is measured. This is the nature of business communication where managers want to know *what* needs to be done, leaving the freedom of *how* it is done to the developers. Management use time, money and resource as project boundaries within which developers must build the desired system. Gantt charts express a project with respect to these boundaries and are used as a means of co-ordinating the project between management and developers.

Whilst Gantt charts support the business aspects of systems development, the technical aspects are often neglected as management is not involved with such issues. Developers must address technical aspects of a project such as choice of hardware and implementation of software. Such issues are not addressed by Gantt chart and, thus, require another form of co-ordination to meet the project specification.

The case study discussed in Chapter 4 aimed at implementing a defined process, whereby a modelling language is used to describe the strategy employed to build the desired system within the boundaries of the project specification. Software process models have been used in all the case studies in order to describe the strategy for each project. This case study illustrated how process models could be used for more than

descriptions. It aimed to illustrate that process models can facilitate a more predictable and manageable systems development project.

Evidence to support the Hypothesis.

Implementing software process modelling at P.C.W. proved to be a useful exercise because it encouraged the developer to focus on technical issues from the outset, rather than at the later stages in the project. Software process modelling supports the hypothesis because they:

- present the technical aspects that make the development project more manageable in terms of activities that are associated to actual system development;
- describe a project in a manner that will make the current and future projects more predictable.

Process models show that details of how the goals presented in a Gantt Chart can be achieved. For example, the Gantt chart for a project may present a bar 'Design Stage' that is allocated a period of three weeks to two designers. There is no indication of how these designers will achieve the designs within those three weeks. Hence, manageability is only achieved to an extent. By using software process models software engineers can show the activities that should be undertaken to fulfil the criteria for the 'Design Stage'. For the data collection project, the activities in the process model explained how the project should be executed to achieve the goals and deadlines set in the Gantt Charts.

As process models are documented, they can be used again for reference to understand, and predict aspects of another project. Essentially, the key benefit of documented process models is their ability to show what mistakes were made before that should not be repeated again. This is the promotion of process improvement in itself. For the data collection Project, one of the key learning points was that resource was underestimated for cabling of the machines and this delayed the project. For future project, an activity for the process models can be 'Consult resources' and this will be placed before implementation activities.

The strongest evidence to support the hypothesis is present in the case study where a testing process was defined and executed in a previous project, but was executed again. As the data collection system and the production line controller system used a common software design, the testing strategy itself was also common. Shell scripts that were saved on the line controller server for future use were changed to meet the specification of the data collection system. The changes were small in comparison to re-writing the scripts again, as only the test data had to be changed. The benefits of automated testing that were encountered in the re-engineering project presented themselves in this project – reduced testing time, more accuracy and documented test procedures.

Conclusion

For people who have been working at P.C.W. for many years these process models still lacked the business-orientated information such as time and resources. It is clear that Gantt charts must be used in conjunction with process models to convey complete information. There is a danger here of further paper work that can lead to distraction from actually *making the product* to spending more time *designing how to make the product*.

Note that one of the key lessons learnt in the case study was that software process models must change accordingly with real-world changes. The same applies to Gantt charts where initial plans and forecasts must be revised when circumstances change. Time and effort is required in maintaining all these documents, and at P.C.W., resources are too little for this concept to work properly. This problem discourages the use of software process modelling and can make it a software process inhibitor, not promoter.

Nevertheless, a foundation of process modelling was laid by this work, and this enabled process modelling to be used in the other process improvement initiatives. It has contributed to the improvement of software practice because a defined process was used, and more importantly, reused. The benefits of this reuse are supported by

Karlsson [Karlsson, 1995] who suggests three major cost-savings in software development:

- working faster (through a better tool set),
- working smarter (through better process for software development and better control of the process by estimation, planning, assessment and improvement),
- work avoidance (through reuse).

No investment had been made into a testing tool set, however, it can be argued that shell scripts that were used do provide an automation facility. The scripts are, to some extent, a tool set for testing of similar systems in the future. A defined process provides a more manageable and predictable process that was also reused. Hence, work smarter and work avoidance were achieved.

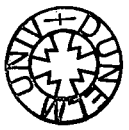
It is well understood that progress in capability maturity must be incremental. Before the research described in this thesis, P.C.W. did not employ any defined models of software development, so with this initiative, the level 3 foundation (defined process) for managed process at level 4 has at least been laid.

8.5 Hypothesis 4.

Introduction of new requirements process will allow developers to understand and satisfy user requirements for a new system.

Overview.

Requirements management and satisfaction are essential to systems development in P.C.W. Developers are pressed to produced a desired system as fast as possible and customers/users brief them with very general requirements. When developers and customers do not share the same perspective of a desired system, extra time, money and resources are spent on realising the same vision through implementation of extra requirements and altered requirements.



There was no formal process for collecting and managing user requirements during maintenance or development at P.C.W. A procedure does exist for new systems development where it is specified that both parties (customers and developers) must accept a User Requirements Specification (URS). However, there are no guidelines that state how this URS should be written. A new formal requirements process can provide a more defined, consistent and controlled approach to systems development where the user specification is the focus throughout the life cycle. The case study for this hypothesis aimed to define a full requirements process that would allow developers to

- collect user requirements in a standard and consistent manner.
- document user requirements.
- manage the requirements to control change during maintenance and development.

Evidence to support the hypothesis.

Whilst there is no practical evidence of improved understanding and user satisfaction as a result of the process supporting the hypothesis, the possible effects that it may have had can be considered. The process in theory would allow the developers to understand the user requirements because:

- More users would provide more input and involvement through a standard pro forma.
- More information would be available to developers in order to understand the user's vision of the system.
- Requirements would be documented in such a manner that developers would have an understanding of what each requirement would do and why it must do it.

The standard pro forma allows developers to ask the same question to different users. Hence, they will receive different opinions from different users. This also means that developers have much more information than they would get if they were talking to designated users who are chosen to represent all users. With more information developers can process the requirements list, validate it, prioritise it and document it when both users and developers are agreed. Without the process, this level of

understanding takes more time, as users are not as involved and a complete understanding is not obtained due to vague and one-sided requirements. The process provides a bigger picture where developers can share the same vision of the desired of the system as the users.

The new process would allow developers to satisfy user requirements because:

- Documented and agreed user requirements reduces the chance of errors.
- Users are involved and updated throughout the process.
- Matrix system provides a check that all requirements are implemented and tested.

Users should be more satisfied because they would be much more involved throughout the process than they would normally be. Also, the user would agree (or "sign off") to all the documented requirements, so during the implementation phase there would be no misinterpretation.

The most effective method in the process for satisfying users would be the requirements matrix whereby each requirement is given a unique identifier when it is documented. During design, implementation and testing each requirement is "ticked off" to show that it has been address at each stage, respectively.

It can be argued, therefore, that employing this defined requirements process during new systems development or software maintenance, could have effects to support the hypothesis 4.

Conclusion.

A new process was designed, however, no conclusions can be made with evidence to prove or disprove the hypothesis. However, when the possible effects of the process are considered, the initiative would have had positive consequences had the process been implemented. At the time of its creation, however, P.C.W. changed its strategy to reverting to manual production processes, which meant that very little, if any, software development would be undertaken in the future. Nevertheless, the process defined here

could be the basis for requirements gathering for I.T. systems that may not require software development such database applications or financial applications i.e. Microsoft Access and Microsoft Excel.

Discussions with the I.T. manager about the defined requirements process, lead to the conclusion that this process may be used in the future to some extent. For example, the requirements collection form may still be used for I.T. new projects as it can work independently of the rest of the process. Even, if the process is not used exactly as it was intended, the underlying principles can be used in some manner to improve requirements management in the future. It can be argued that this case study has made I.T. staff at P.C.W. very aware of the need for such a process. This, in turn, could lead to improved software practices at some later point in time when software might play a large role in the company's strategy again.

8.6 Summary.

This chapter has discussed the results of the four case studies that were initiatives to apply software process improvement as a drive for improving the software practices in the company. Each case study has addressed specific areas of software engineering, i.e. maintenance, testing, planning and control, requirements management or testing, and efforts have been made to raise the company's maturity in the CMM with respect to these areas. Collectively, the case studies aimed to achieve improvement of software practices by way of applying software process improvement in some manner.

The results have shown that there is evidence to show that software process improvement has been achieved as each case study has laid foundations for maturity by defining processes. In terms of maturity levels, this could place P.C.W. in the level 3 - the *defined* stage. However, it is left to the company to maintain their position by using defined processes and laying the foundations for level 4 – the *managed* stage.

Chapter 9

Conclusions and Further Work.

9.1 Introduction.

A two-year T.C.S. project was undertaken at Philips Components Washington, which is a medium-sized company that manufactures components for television sets. The scheme aimed to support projects for reengineering production line software for Year-2000 compliance, contribute to the development of a Manufacturing Information System and to upgrade the software practices of the company. This research has described how efforts were made to upgrade the software practices by way of implementing small-scale software process improvement initiatives.

Four case studies were undertaken within two large, ongoing projects. Using the IDEAL approach, each case study identified an area for process improvement. Through diagnosis the current state of practice at P.C.W. was surveyed and recommendations are made to solve the problem. The actions were taken to implement the recommendations and then the actions were evaluated to understand what could be learnt from the initiative.

The results discussed in the previous chapter presents evidence to suggest software practices were upgraded through small-scale software process improvement. The CMM framework was used as it provides flexibility when implementing software process improvement. It can be used as a guide as each maturity level requires some form of process improvement. For example, defining a formal process for projects is step away from simply costing, scheduling, planning and other repeatable characteristics that are found in level 2. Once processes are defined and improved, an organisation can begin to use measurements from the process definitions to monitor and control processes. This would not have been possible with the SPICE standard because specific guideline must be observed that would have been difficult to observe

in an organisation such as P.C.W. where resources are scarce and funding is very minimal.

9.2 Evaluation of the criteria for success.

Chapter 1 presented the criteria for success which were:

a) to investigate the areas associated with software process improvement in order to gain background knowledge that will support initiatives.
b) to identify where the company will need to improve the processes in order to formulate hypotheses regarding the effects of software process improvement.
c) to undertake case studies in order to prove or disprove the hypotheses.

The first criterion was successfully completed as Chapters 2 and 3 presented material that was obtained from the investigation into software process improvement. Most of the software theory here was based on Frederick Brooks' "No Silver Bullet [Brooks, 1995]", a paper that is praised in the software engineering field for its concise explanation of the inherent properties of software. It is from this paper that further study was undertaken into the problems encountered for software maintenance that was used in the case study 1 for the re-engineering of production line controller systems. In general, it was important that a full appreciation of the nature of software was gained because it was the underlying cause of the research.

The research then progressed from 'Software' to 'Software Processes'. It was necessary to ask why software processes have been so widely researched, and also to see how they have evolved in the last three decades. The most useful part of this research was studying the need for software processes because it underlined an issue that has been overlooked in P.C.W., and that is, how to plan for the construction of a software solution given the constraints of time, money and resources. When a project was set at P.C.W., managers agreed on project specifications with Gantt charts and the developers were left to decide how the specifications should be met. However, planning the *'how'* is seldom achieved and developers simply jump into the phases specified in the Gantt chart. An insight into the history of software processes provided an understanding of how people have attempted to plan projects in the past and how

these methods have evolved to present day where processes are programmed and executed like software programmes.

Furthermore, an investigation into software process improvement was carried out. The focus was on the current frameworks that are available for implementing software process improvement. The information found on the CMM was used extensively in this research because it was used as a guide to measure the level of maturity that each software process improvement initiative achieved.

The second criteria for success was achieved and the work is presented in chapter 2 and more specifically in each chapter, as these discussed the areas in which P.C.W. needed to improve in order to upgrade their software practices. Chapter 2 discussed the 'problem domain' and set the context of this research, explaining why the company proposed the T.C.S. scheme. The chapter also looked for possible obstacles that may have to be overcome when trying to implement software process improvement initiatives in the company. This provided an appreciation of the issues that must be addressed when implementing new methods with people who are not familiar with them.

The most important part of this work is the discussion of the IDEAL approach that actually identifies a specific problem area, diagnoses a solution and implements it. It is this systematic and uniform approach that addressed the software process improvement initiative for each specific problem area. Consequently, each initiative was given a hypothesis that was proved or disproved. Hence, the second of the criteria was completed.

The third of the criteria was also achieved successfully and the work for each of the case studies is presented in chapters 4, 5, 6 & 7. Each case study used the IDEAL approach to identify, diagnose and resolve a specific problem with respect to software projects at P.C.W.

These case studies provided an opportunity to implement software process improvement initiatives in a live, industrial environment. The advantages to this

opportunity are that business issues had to be addressed. Such issues might have been ignored if they were implemented in a laboratory. The case studies have shown that implementing much of the theory that has been researched can be very difficult in a live environment. This is especially true for case study 4 (chapter 7) where a requirements process was designed but not implemented due to sudden changes in the business strategy at the time.

The case studies related to the projects all had a definite business purpose and were not merely experiments that would not be implemented. They all had time scales and budgets which meant that the software process improvement initiatives were under the same constraints. For case study 2, this meant that the testing time was reduced because of the time spent researching and formulating a process to execute. However, the process improvement initiative proved to be a success as the production line controller system that was tested was free from high-risk defects. This was a good example of how investment into software process improvement, however small, can sometimes bring considerable benefits to the business.

9.3 Further Work.

Software process improvement at P.C.W. can be furthered in many areas. The I.T. department will have very few members in the future. New business policies dictate that contractors will be used for I.T. work in the future. This has major implications for the maintenance of existing systems as all the knowledge from the developers that will be taken with them. Maintenance is a costly process. Hence, further work can be done to extend process improvement to ensure that maintenance is undertaken in a controlled manner that will be effective, productive and efficient.

Much of the process improvement initiatives involved very few people. Software process improvement would be more of a challenge if the development team consisted of a wider variety of skilled people. Humphrey has discussed 'Team Processes' [Humphrey, 1999] that encourages developers to formulate a process where individuals can work most effectively as team in a development or maintenance process. Perhaps much of this work would have been more relevant during the 1980s

when the production line controller software was being developed in the company with teams of software engineers.

In the future, the I.T. department at P.C.W. will have very few permanent staff and contractors will be used to perform development or maintenance tasks. Whilst software process improvement cannot be supported, the company can support the individuals to use a disciplined approach to their work. Humphrey suggests Personal Software Processes (PSP) [Humphrey, 1995], that encourages the individual developer to understand and improve their role in a development or maintenance project. The company can look to train any contractor to that is new to the department, or they can introduce this factor in their selection criteria when new contractors are introduced.

9.4 Final Words.

The most satisfying aspect is that this research has been carried out in a live, industrial environment where there are many other issues that effect software process improvement, and some of these would not have been appreciated by simply sitting a computer laboratory. Process improvement initiatives can be devised in a laboratory, but they must be applied in an environment where all 'real world' issues present themselves and make an impact on the initiatives. Finally, working with the I.T. department has been a very enlightening and educational experience. All the staff supported this project with complete co-operation. They contributed with their knowledge and expertise throughout the project that made it interesting and enjoyable to undertake.

References.

Boehm, B. and P. Papaccio (1988). "**Understanding and Controlling Software Costs.**" IEEE transactions on Software Engineering: 1462-1477.

Boldyreff, C., E. Burd, et al. (1995). "**The AMES approach to application understanding.**" Proceedings of the International Conference on Software Maintenance, IEEE Press.

Boldyreff, C., E. Burd, et al. (March 1996). "**Greater understanding through maintainer driven tracebility.**" Proceedings of the International Workshop in Program Comprehension, IEEE Press.

Brooks, F. (1995). **The mythycal man-month**, Addison-Wesley. Chapter 16.

Cugola, G. and C. Ghezzi (1998). "**Software Process: Restrospective and a Path to the Future.**" Software Process - Improvement and Practice **4**(3): 103-112.

Davis, A. (1993). **Software requirements objects, functions and states.**, Prentice-Hall: Chapter 3.

Dowson, M. (1985). **The Structure of the Software Process.**, International Workshop of the Software Process and Software Environment, IEEE Computer Press: 55-60.

Fenton, N. (1995). **Software Quality - Assurance and Measurement.**, Thompson Computer: Chapter 3.

Finkelstein, A., J. Kramer, et al. (1994). **Software Process Modelling and Technology.**, John Wiley and Sons: Chapter 1.

Frazer, A. (1992). **Reverse Engineering - Hype, Hope, or Here?**, Chapter 10 in "Software Reuse and Reverse Engineering", Chapmen-Hill: 209-243.

Humphrey, W. (1989). **Managing the Software Process.**, Addison-Wesley.

Humphrey, W. (1995). **A discipline for software engineering.**, Addison-Wesley.

Humphrey, W. (1999). "**The Changing World Of Software.**" Software Engineering Institute: Carnegie Mellon University (www.sei.cmu.edu/publications/articles/watts-humphrey/).

Humphrey, W. (1999). "**Why don't they practice what we preach?**" Software Engineering Institute: Carnegie Mellon University (www.sei.cmu.edu/publications/articles/watts-humphrey/).

ISO/IEC (1995). **ISO/IEC Software Process Assessment - Part 1: Concepts and Introductory Guide.**, SPICE Project Organisation.

Karlsson, Even-Andre. (1995). **Software Reuse: A Holistic Approach.**, Wiley & Sons.

Kasse, T. and P. McQuaid (1998). "**Entry into the Process Improvement Initiative.**" Software Process - Improvement and Practice 4(2): 73-80.

Kit, E. (1995). **Software Testing in the Real World.**, Addison-Wesley.

Lehman, M. and L. Belady (1980). "**Understanding Laws, Evolution and Converstation in Large Program Lifecycle.**" Journal of Software Systems. 1(3).

Miller, K. and C. Huntz (1983). **Software Improvement Program - A solution for software problems.**, Software Maintenance Workshop, Addison-Wesley: 120-123.

Mistry, S. (1999). **URS - Winding Machine Data Collection System.**, Philips Components Washington.

Myers, G. (1976). **Software reliability: principles and practice.**, New York, Wiley.

Myers, G. (1979). **The art of software testing.**, John Wiley & Sons.

Paulk, M. (1995). **The Capability Maturity Model: Guidelines for improving the software**

process., Software Engineering Institute, Addison-Wesley.

Perry, W. (1995). **Effective methods for software testing.**, John Wiley & Sons: 7.

Pigoski, T. M. (1997). **Practical Software Maintenance.**, Wiley & Sons.

Royce, W. W. (1970). **Managing the development of large software systems.**, 1970 WESCON Technical Papers Western Electronic Show and Convention, pp.A/1-1 - A/1-9. Reprinted in: Proceedings of the 9th International Conference on Software Engineering, March 1987, pp 328-337.

Sommerville, I. and P. Sawyer (1997). **Requirements Engineering - A good practice guide.**, Wiley & Sons.

Taylor, A. (January 2000). **IT Projects:Sink or Swim?** The Computer Bulletin. **BCS Magazine:** 24-26.

Tilley, S. (1998). "**Coming attractions in Program Understanding II: Highlights of 1997 and opportunities of 1998.**" CMU/SEI-98-TR-001, Software Engineering Institute: Carnegie Mellon University.

Weihrich, H. and H. Koontz (1994). **Management: A Global Perspective.**, McGraw-Hill.

Appendices

Appendix A1

The first draught of the Requirements Collection Forms.

Description of your role in the company:

Requirement code	Requirement description	Priority (Low, medium or high)

Need to break all this down otherwise we'll just have a long list
Address-data
- Security
- Performance

STILL TOO VAGUE.
ASK SPECIFIC QUESTIONS.

More boxes required as users will ask for a lot more

Too textual - get user to draw a picture of what they want

Appendix A2

The Second draught of the Requirements Collection Forms.

User's name:

Description of your role in the company:

List and describe the constraints of the desired system.

↑
MAKE THIS LIKE REQUIREMENTS LIST.

List and describe the items of data that the desired system must store.

↑
Break down - more specific

Context Diagram. Draw the entities that will interface with the desired system e.g. departments, other systems, people, etc.

Do they know what a Context diagram is?
Draw an empty one. They can fill it in

Requirement code	Requirement description	Priority (Low, medium or high)

Description of your role in the company:

Requirements Collection Form

1a. Project:

1b. Date submitted to the Customer:

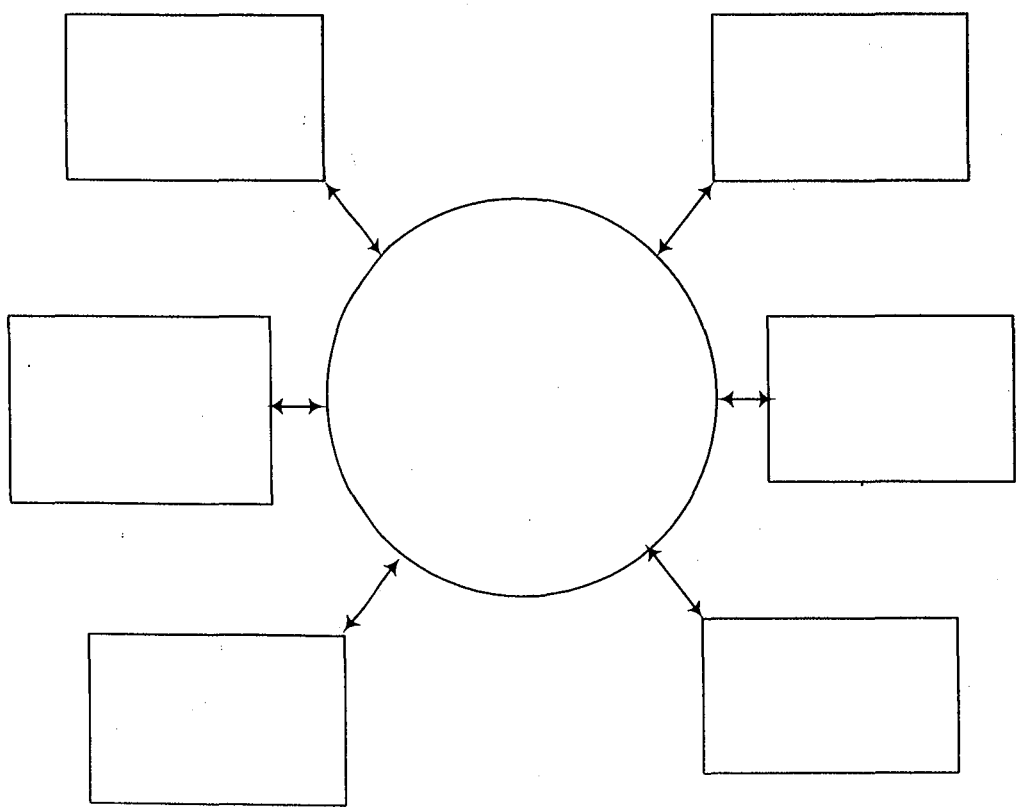
1c. Date received from the Customer:
(Developers use only)

2a. Customer/User name:
2b. Contact details:

2c. Description of your role in the company:

2d. Description of your role with the proposed system:

3a. Context of the proposed system.
Write the name of the proposed system in the circle and then write the entities (staff roles, department name, other systems, etc.) that will interact with the system. If more boxes are required then draw them and label them freehand.



4a. Business Concerns:

Describe the **ultimate business benefits** that the proposed system should aim to provide.

List all the different **items of data** that the proposed system must store or process. Provide calculations if necessary.

List the **performance expectations** that the proposed system must meet.

5a. Operational Process Concerns:

What are the **boundaries of the proposed system**? Will there be users external to Washington? Which departments will use the system?

What **existing manual processes** will the proposed system automate?

What are the **security and access** measures imposed by the proposed system.

6a. Requirements List:

In the table below write your requirments for the proposed system. You must mention why the requirement is needed, how it contributes to the business process and it's priority grade. Your explanation should be concise, but detailed. If your requirement comes from source other than yourself then write the source down so that it can be traced at a later period.

Desired requirement (what the proposed system must do)	Reason for the requirement (Why the requirement is requested)	Contribution to the business process.	Priority (high, medium or low)

Requirements List:

In the table below write your requirments for the proposed system. You must mention why the requirement is needed, how it contributes to the business process and it's priority grade. Your explanation should be concise, but detailed. If your requirement comes from,source other than yourself then write the source down so that it can be traced at a later period.

Desired requirement (what the proposed system must do)	Reason for the requirement (why the requirement is requested))	Contribution to the business process	Priority (high, medium or low)

7a. Constraints List:

In the table below write the your requirements in terms of what the proposed system must not do. You must mention why the constraint is needed, how it contributes to the business process and it's priority grade. Your explanation should be concise, but detailed. If your requirement comes from source other than yourself then write the source down so that it can be traced at a later period.

Desired constraint (what the proposed system must not do)	~~Reason for the constraint~~ (why the requirement is requested))	Contribution to the business process	Priority (high, medium or low)

8a. Outputs of the Proposed System.

Present below the output of the proposed system. Show the format of reports, charts and graphs that the system is expected to produce.

9. What are the interface requirements for the proposed system. Describe how the users will interact with the system.

PHILIPS

Requirements Collection Form:

User Guide.

Introduction.

This user guide explains how the 'Requirements Collection Form' should be filled in. As a customer or user of the proposed system it is essential that you provide your perspective of the proposed system. The information that you provide will be used to generate a user requirements specification (URS) which is then used to build the system. The more accurate you requirements written then the more likely that the system will meet your needs providing that they are feasible to implement.

Section 1.

a. *Project.* The developers will write the name of the project or the system that the form is submitted here. Please verify that it is correct if it is not then correct the mistake.
b. *Date submitted to the Customer.* Enter the date that you received the form if it has not already been filled in.
c. *Date submitted to the Customer.* (For developers use only).

Section 2.

a. *Customer name.* Enter your name here, unless the developers have already written it.
b. *Contact details.* Enter you contact details such as extension number, email address, department name, etc.
c. *Description of your role in the company.* State your title and very briefly explain the role you play at Philips Components Washington.
d. *Description of your role with the proposed system.* Briefly describe your role with the proposed system.

Section 3.

a. *Context of the proposed system.* This section aims to define the surrounding environment of the proposed system. The circle represents the proposed system and the rectangles around it are the entities that interact with it. The entities can be roles of staff, departments, other systems, anything that must input to or receive output from the proposed system. Hence, enter the name of the proposed system and then enter the entity names.

Section 4.

a. *Describe how the proposed system will contribute to the business.* Here you should explain the purpose of the system and the benefits it must provide to the business. Mention any performance improvement measures that is expected of the system.

Section 5.

a. Describe the business process that the proposed system will support. Briefly describe the overall business process that the system will play a part in. Mention

the inputs and outputs of the process. Use diagrams to support your answer or for calculations write the formula that the system will use.

Section 6.

a. *Requirements List.* The table aims to collect your specific requirements of **what the system must do** from **your own perspective of the** system. It is essential that fill out the whole row for each requirement so that the developers will understand each requirement, why each requirement is necessary, and the contribution to the business process each requirement will make. The priority section should state how important a requirement is to the overall function of the system. It is important that each requirement is expressed concisely, but with sufficient detail to avoid ambiguity. More *Requirements List* sheets can be obtained if there are fewer rows for your requirements.

Section 7.

a. Constraints List. This section is that same as Section 6, but you must express **what the system must not do**. More *Requirements List* sheets can be obtained if there are fewer rows for your requirements.

See the example form provided before you fill out the form.

Appendix A4

Requirements Documentation Template and User Guide.

Requirements Matrix and User Guide.

User Requirements Specification.

Document Identifier:

System:

Customer(s) :

Developer(s):

Statement of Purpose:

System Environment

Date of Agreement.

Signiture of Principle Customer.....Date.....

Signiture of Principle Developer.....Date.....

Volatile Requirements:

Global System Requirements:

User Requirements List

Requirement Code	Requirement Description	Requirement Source	Priority (High, Medium or Low)	Requirement Design Code	Requirement Test Code

User Constraints List

Constraint Code	Constraint Description	Requirement Source	Priority (High, Medium or Low)	Contraint Design Code	Contraint Test Code

PHILIPS

User Requirements Specification:

User Guide.

This user guide explains how the 'User Requirements Specification' should be written. The guidelines below should be followed in order for all requirements documentation to be consistent. This documentation will be the final draft of the users' requirements of a system and it will be used in the design stage and the testing stage.

1. *Document Identifier.* This is a code that will uniquely identify the document. One system will have one requirements document.
2. *System.* This is the name of the proposed system and it will be used as a reference to the requirements specification also.
3. *Customers.* The name(s) of the customer(s) and their department(s) are listed here. Note: the users are the customers they are not stated here.
4. *Developers.* The name(s) of the developer(s) and their department(s) are listed here.
5. *Statement of Purpose.* This is a brief, textual statement of the overall function of the proposed system. It should describe the main function of the system without mention of cost, design or implementation.
6. *System Environment.* A context diagram should be used to show the entities that will interact with the system. The diagram should consist of a circle that represents the system and rectangles around it that represent the entities. An entity is a physical object that will give input to or receive output from the proposed system, i.e. I.T. Department, Production Manager, Line Controller, etc. Hence, an entity can be a person, a group of people, another system. Arrows must be drawn to show the direction of interaction between the system and an entity.
7. *Date of Agreement.* This document will be 'signed off' to signify that the developers and the customers are agreed on the requirements of the system and are satisfied that they will be implemented. After the document has been signed the requirement changes to the document will be made in a controlled manner and the document will be used throughout the design and testing process to ensure that each requirement is satisfied.

<u>User Requirements List.</u>

8. *Requirement Code.* This code uniquely identifies a requirement and is used for tracing the requirement during later stages of the development.
9. *Requirements Description.* This is a textual description of the requirement. It must be concise and understandable to both customers and developers. There must be no ambiguity and one requirement must represent one function.
10. *Requirements Source.* The origin of the requirement must be stated. It can be a person, in which case a contact number must be given and the role of the person should be stated, not their name. Or it can be source such as a regulations manual, in which case the reference must be given.

11. *Priority.* This is the level of priority that a requirement has in relation to other requirements. The highest priority requirements should be listed first and then the medium priority and followed by the low priority ones.
12. *Requirement Design Code.* This is used during the design stage. When a requirement has been satisfied in the design stage the design has a code allocated to it. Note that one design may satisfy several requirements, hence, the same code may appear on several rows of the requirements list.
13. *Requirement Test Code.* This is used during the testing stage. When a requirement has been satisfied in the testing stage the test has a code allocated to it. Note that one test may satisfy several requirements, hence, the same code may appear on several rows of the requirements list.

User Constraints List.

Steps 8 to 13 applies to the Constraints List but the requirements are concerned with what the system must not do.

14. *Volatile Requirements.* These requirements are those that are most likely to change. Identify and list those requirements that are most likely to change and adjust the development process to accommodate for these changes to reduce the delay in the schedules.
15. *Global System Requirements.* These requirements are those that influence the whole system. Hence, if a change occurs here then many dependent requirements will be affected. Prepare for such changes in the development process. These requirements should be analysed carefully before design takes place.

Requirements Tracebility Matrix (RTM)

Document Identifier: System:

Requirement Code	Test Code											

PHILIPS

Requirements Traceability Matrix:

User Guide.

Introduction.

This user guide explains how the 'Requirements Traceability Matrix (RTM)' should be used. There are two forms of RTM – requirements/design and rrequirements/test. Each maintain a link between the requirements to their designs, or requirements to their tests, respectively.

The Requirements/Design RTM.

During the design phase of a system, this form of the RTM is used to show which design satisfies the requirement(s). The requirements codes are listed in the left-hand-side column and the design codes are listed across the top row of the matrix.

From the User Requirement Specification list all the requirements codes in the gray boxes in the left-hand-column of the RTM. During systems design, a unique code must be allocated to the design. These codes are written in the gray boxes at the top of the matrix. Once the design has been signed off it is then ticked off against the requirement(s) that it satisfies. The RTM is then placed under source code control once all the requirements have been designed.

The Requirements/Test RTM.

During the test phase of a system, this form of the RTM is used to show which test satisfies the requirement(s). The requirements codes are listed in the left-hand-side column and the test codes are listed across the top row of the matrix.

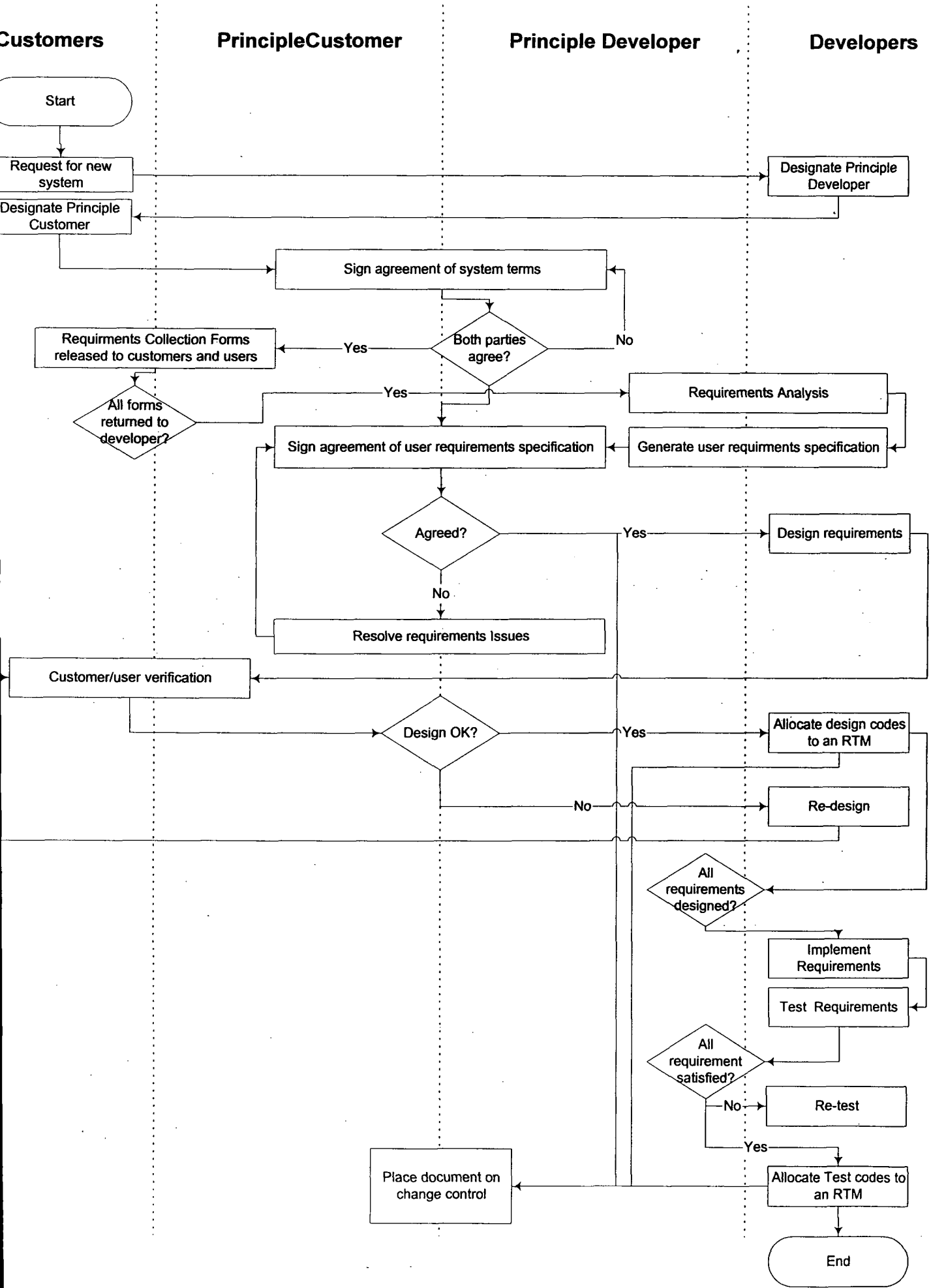
From the User Requirement Specification list all the requirements codes in the gray boxes in the left-hand-column of the RTM. During systems testing, a unique code must be allocated to the test. These codes are written in the gray boxes at the top of the matrix. Once the test has been signed off it is then ticked off against the requirement(s) that it satisfies. The RTM is then placed under source code control once all the requirements have been tested.

Appendix A5

System Requirements Process Diagram and Procedure.

Procedure.

Flowchart: **System Requirements Process**



Document Title	:	Requirements Procedure.
Author	:	Sanjay N. Mistry
Date	:	16 June, 2000
Project	:	T.C.S.

PHILIPS COMPONENTS

REQUIREMENTS PROCEDURE.

Introduction.

This document defines the procedure for requirements management for systems that are developed at Philips Components Washington. The procedure sets a standard method of collecting, documenting and managing requirements for any proposed system.

Procedure.

See the flowchart of the *System Requirements Process*.

Description

1. Request New System. The customer proposes a new system to the developers.
2. Designate Principle Developer. The developers will assign one person to represent the development team that will communicate with the principle customer.
3. Designate Principle Customer. The customers will assign one person to represent them to communicate with the principle customer.

It is the Principles responsibility to then communicate any issues to other members of their party.

4. Sign agreement of terms. To commence the project, the principles must first agree on some of the basic systems development issues. The basis for their discussion is the **Systems Development Checklist** that asks some questions about various topics regarding the proposed system. After the checklist has been filled in it is signed by the principles of both parties as a mark of agreement. Progress should not be made until the checklist has been agreed and signed.
5. Requirements Collection Forms released to customers and users. **Requirements Collection Forms** will be handed to all customers and users of the system. All sections of the form must be filled out and ALL forms must be returned to the developers to continue development.
6. Requirements Analysis. The developers must analyse and interpret the requirements obtained from the Requirements Collection Forms. The sources may be contacted for clarification. (see Requirements Organisation).
7. Generate the User Requirements Specification. The standard **User Requirements Specification** must be used to create a requirements document abiding by the user guidelines.
8. Sign agreement of user requirement specification. The **User Requirements Specification** must be signed by the Principles if both parties are satisfied with the requirements specification of the system. If the parties are not agreed, then the issues should be resolved, a new **User Requirements Specification** should be created until it is signed. Otherwise development must not continue. Once the **User Requirements Specification** has been signed then it should be placed under the Change Control System to control requirement change.
9. Design Requirements. The developers design each requirement.

10. Customer/User verification. The developers must verify each design with the customer or users of the proposed system.
11. Redesign. If the design of the requirement(s) is not satisfactory to the user then the developer must redesign the requirement. Note: if a requirement changes then it must be implemented at the **User Requirements Specification** that is under change control.
12. Allocate design codes to the RTM. Once a design has satisfies a requirement, it should be given a unique identification code that will be used to trace the requirement(s) that it satisfies. The RTM **(Requirements Traceability Matrix)** must be filled in to show which requirements are satisfied by design. The RTM for requirement/design traceability will be placed under change control in order to keep the **User Requirements Specification** and the RTM consistent.
13. Implement requirements. The implementation of the system must only take place unless ALL requirements are satisfied by design i.e. all the requirements codes in the RTM have corresponding design codes.
14. Test Requirements. During system testing it is essential that the requirements are satisfied.
15. Re-test. If the test is not satisfied, then it must undergo re-testing until the requirement is completely satisfied.
16. Allocate test codes to an RTM. When a test satisfies the requirement it must be given a unique identifier that will be placed in the RTM with corresponding requirement code. Hence a traceable link will be created between the test and the requirement. The RTM for requirement/test traceability will be placed under change control in order to keep the **User Requirements Specification** and the RTM consistent.

Appendix B

Test Procedures and Scripts.

Document Name	:	Test Procedure – OC1A/rec009
Author	:	Sanjay N. Mistry
Date	:	15 April, 1999
Project	:	Auto-YAMA-2 – Year 2000 Compliant.

Function : REC009.c
Module : OC1A
Purpose : Test LOCAN Communications.
Description :

The rec009 message is called from the PIT Cell. It validates the product type and checks that it is currently in production. It is also responsible for the box data validation.
If this message is a retry, then it is ignored.

Focus of Test :

The rec009 function has passed acceptance test for Auto-YAMA-1. The functions that it must perform are identical between the Auto-YAMA-1and Auto-YAMA-2. Hence, the test must prove that:

1. When valid product types are simulated no error messages must be generated.
2. When invalid product types are simulated error messages must be generated.
3. Ensure that carrier records are correctly updated, i.e. carrier status, carrier to_position, and carrier product.
4. Ensure that an 800 message is sent to OC2A.

Test Data.

In the tests, carriers 1 to 4 will be used and the fault code will be UNIT_NOT_FAULTY(0). Only the PIT cell uses this message so the cell address will be 71.

Test	Product Code	Status	Expected Resulted.	Pass/Fail
1	616 (defined, not in use)	Production Good	Error issued because product is not in-use. Carrier at_position should become 71. Carrier product should become 616. 800 message should be received by OC2A.	PASS
2	111 (defined, in use)	Production Good	No error messages should be issued. Carrier at_position should become 71. Carrier product should become 111. 800 message should be received by OC2A.	PASS
3	154 (defined, in use)	Production Good	No error messages should be issued. Carrier at_position should become 71. Carrier product should become 616. 800 message should be received by OC2A.	PASS
4	010 (not defined)	Production Good	Error issued because product is not in-use. Carrier at_position should become 71. Carrier product should become 616. 800 message should be received by OC2A.	PASS

See evidence attached:
Test data, 801 messages, carrier tables, and error messages.

Test data is in file *009testdata.*

```
# ********************************************************************************
# Filename : 009testdata.
# Author   : Sanjay Mistry
# Date     : 14/4/98
#
# Purpose  :
#
# This test data simulates a rec009 message.
# The message sent in this script tests that a undefined product and not in_use
# product issue a error statement. The rec009 must also update the at_position
# and product code for the carrier record. It must send a 800 message to OC2A.
# correctly if the calling cell is other than the yoke shift cell (68).
#
# ********************************************************************************

# TEST 1
msgsnd 71 "0090001616" 202;
msgrcv 71 203;

# TEST 2
msgsnd 71 "0090002111" 202;
msgrcv 71 203;

# TEST 3
msgsnd 71 "0090003154" 202;
msgrcv 71 203;

# TEST 4
msgsnd 71 "0090004010" 202;
msgrcv 71 203;
$ exit
```

```
SQL> @select009.sql

CARRIER_ID PRODUCT_CODE      STATUS FAULT_CODE CURRENT_POSN ALLOCATED_POSN
---------- ------------ ---------- ---------- ------------ --------------
         1          616          2        908            8             10
         2          111          1          0            8              0
         3          154          1          0            8             18
         4                       1          0            4             10

SQL>
```

```
$ /home/sanjay/testscripts/009testdata
msgsnd: send done
msgrcv: message type:   71
        message:   800001111000
msgrcv: receive done
msgsnd: send done
msgrcv: message type:   71
        message:   800001541000
msgrcv: receive done
msgsnd: send done
msgrcv: message type:   71
        message:   800000102907
msgrcv: receive done
msgsnd: send done
msgrcv: message type:   71
        message:   800006162908
msgrcv: receive done
$ cdl
/ay2/logs
$ more F_ERR*
1999/04/15[08:59:41]@%Z%oc1a %I%@rec009@%C%@Product type (10) not defined in MDBP
1999/04/15[08:59:41]@%Z%oc1a %I%@rec009@%C%@Error updating carrier record   4
1999/04/15[08:59:41]@%Z%oc1a %I%@main@%C%@REC009 function error (node = 71 msg = 004010)
1999/04/15[09:00:27]@%Z%oc1a %I%@rec009@%C%@Product type (616) not currently in use
1999/04/15[09:00:27]@%Z%oc1a %I%@rec009@%C%@Product type (10) not defined in MDBP
1999/04/15[09:00:27]@%Z%oc1a %I%@rec009@%C%@Error updating carrier record   4
1999/04/15[09:00:27]@%Z%oc1a %I%@main@%C%@REC009 function error (node = 71 msg = 004010)
```

Document Name	:	Test Procedure – OC2A/rec041
Author	:	Sanjay N. Mistry
Date	:	22 April, 1999
Project	:	Auto-YAMA-2 – Year 2000 Compliant.

Function : REC041.c
Module : OC2A
Purpose : Request cell configuration.
Description :

The rec041 message is sent by a cell to OC2A to request configuration data. The OC2A delivers the cell configuration data from the LINE_COMMS table to the calling cell in a 042 message. There is only one entry in the LINE_COMMS table.

Focus of Test :

The rec041 function has passed acceptance test for Auto-YAMA-1. The functions that it must perform are identical between the Auto-YAMA-1and Auto-YAMA-2. Hence, for this test there must be evidence that the function can perform the following:

1. The 041 must send a rec042 to the calling cell otherwise an error must be produce.

Test Data.

Test	Calling cell	Expected Result	Pass/Fail
1	68 (Yoke Shift)	A rec42 will be sent with the call configuration data.	PASS

See evidence attached:
Test data, carrier tables, and error messages.

Test data is in file *041testdata.*

Conclusion.

The rec041 has passed the tests. There were no bugs found. The YOKE SHIFT cell was used because it is has been implemented in the AY1 code which does not have an YOKE SHIFT cell.

```
# ********************************************************************************
*
# Filename : 041testdata.
# Author   : Sanjay Mistry
# Date     : 19/4/98
#
# Purpose  :
#
# This test data simulates a rec041 message.
# The message sent in this script tests that a cell configuration data
# is sent to the calling cell by the OC2A module. If there is an error
# retreiving the information from the LINE_COMMS table then an error is logged.
#
# ********************************************************************************
*

# TEST 1
msgsnd 68 "0410" 203;
msgrcv 68 205;
~
~
~
"041testdata" 18 lines, 575 characters
```

```
SQL> select * from line_comms;

LC_TIMEOUT   LC_RETRY CELL_TIMEOUT CELL_RETRY
---------- ---------- ------------ ----------
         8          3          100          3

SQL>
```

```
$ /home/sanjay/testscripts/041testdata
msgsnd: send done
msgrcv: message type:   68
             message:   04218010001000300
msgrcv: receive done
$
```

Conclusion.

- SNIFF+ supports the concepts of source code understanding that has been addressed by research in to software maintenance. For a new engineer the production line software can take up to two months. This tool can reduce that to two to three weeks.
- The tool will reduce the maintenance time which because of multiple source view, hypertext links for navigation and documentation of the code.
- The tools can run on different platforms and supports multiple users.

Recommendations

- Continue evaluation of other tools but strongly consider the SNIFF+ Cross tool.

Summary

SNIFF+ is a source code reengineering tool that can reduce maintenance times considerably. It features tools that support the concepts of maintenance developed by University of Durham during the AMES project. An evaluation of the tool for two weeks concludes that the tool can be effectively used in the company to improve it maintenance practices.

Referenences

[1] "Manipulating and Documenting Software Structures Using ShriMP Views" M. Storey and H. Miller, IEEE, 1996.
[2] "Greater Understanding Through Maintainer Driven Traceability" C. Boldyreff, et al, IEEE, 1996.
[3] "The AMES Approach to Application Understanding: a case study" section 3.5, C. Boldyreff, et al, IEEE, 1995.

Contact:

TakeFive Software Ltd,
The Surrey Technology Centre
Mr. Kas Subhan
40 Occam Road, The Surrey Research Park GU2 5YG Guildford, Surrey
UK
Tel.: +44 1483 29 50 50 Fax: +44 1483 29 50 51
Email: ksubhan@takefive.co.uk

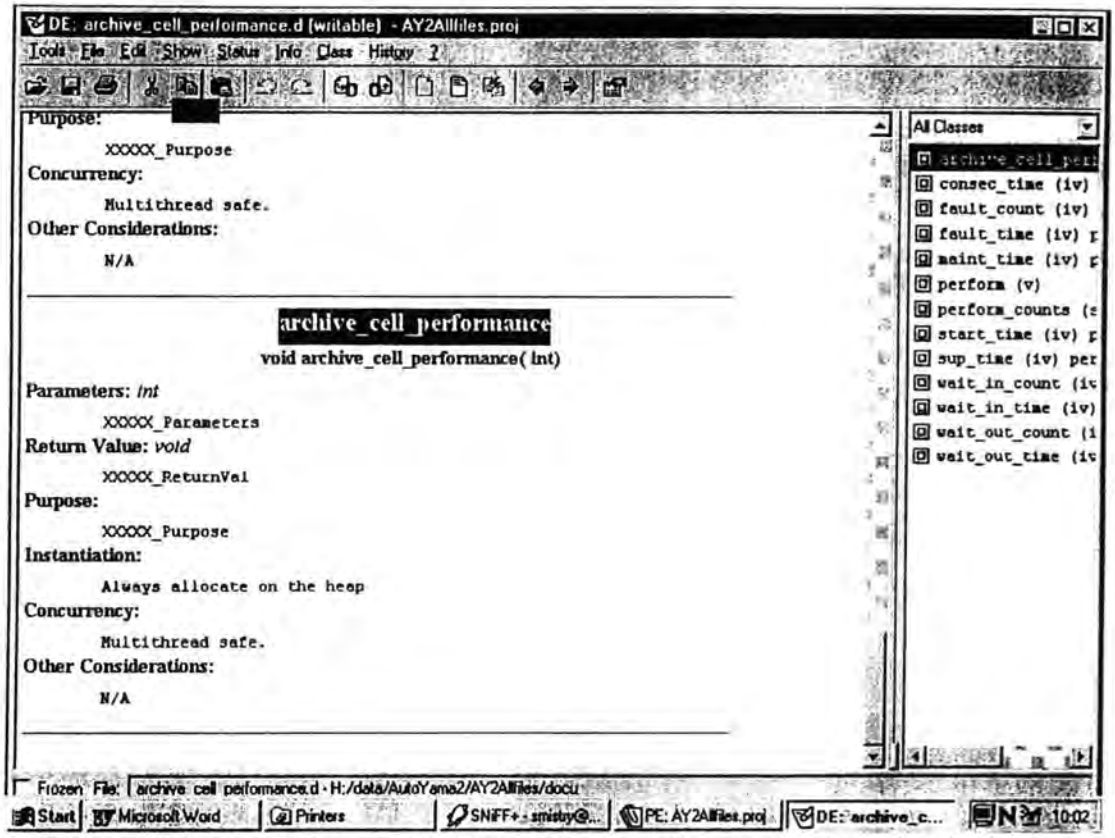


Fig.1 Cross referencer provides multiple views to facilitate code comprehension and navigation.

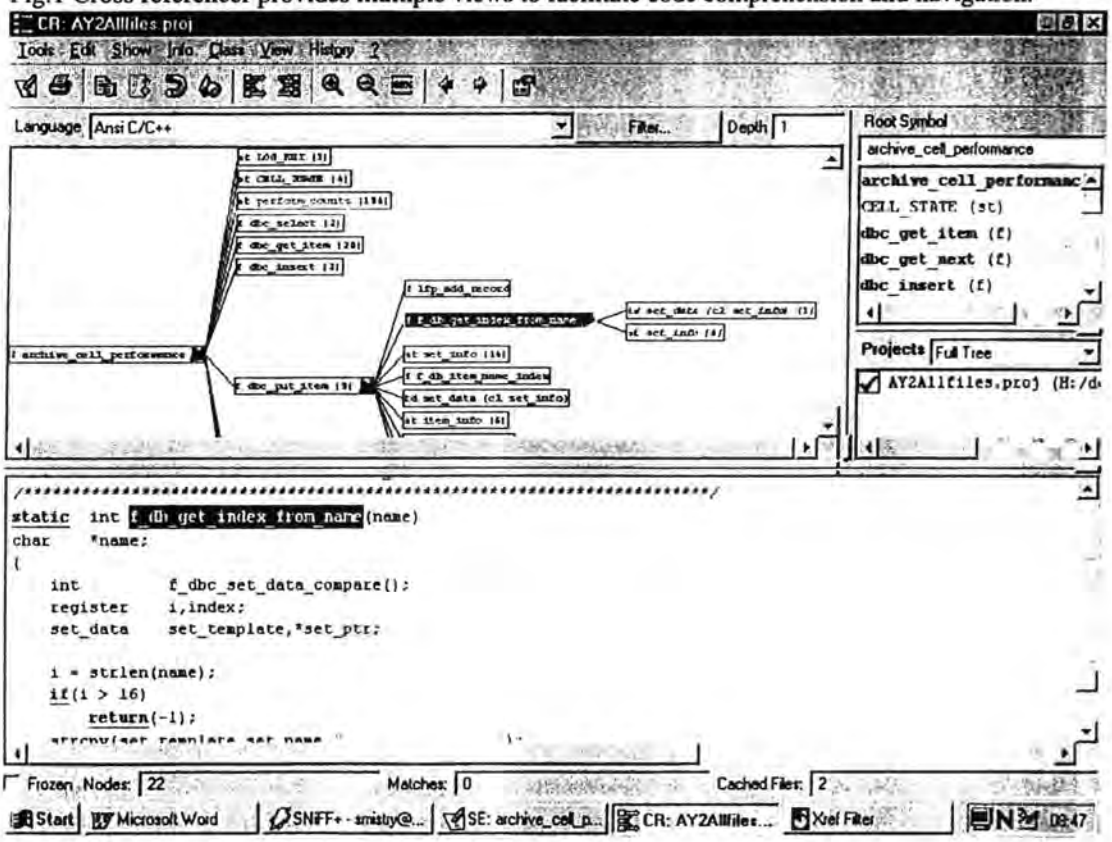


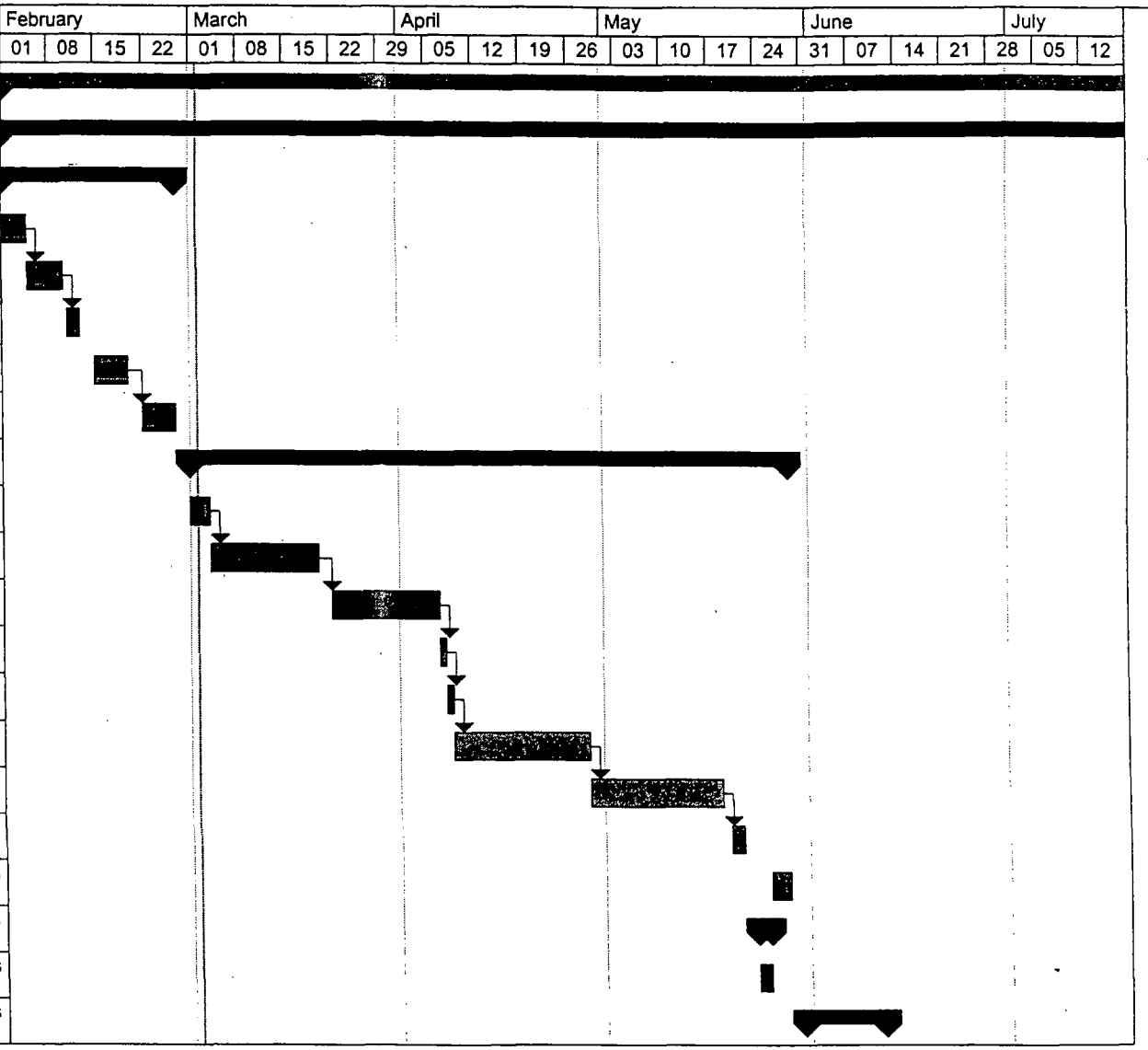
Fig.2 Document Browser prompts for useful information about each function and variable.

Appendix C2

Gantt Chart for re-engineering of Auto-Yama-2 production line controller system.

Plan for Reengineering of Auto-YAMA-2

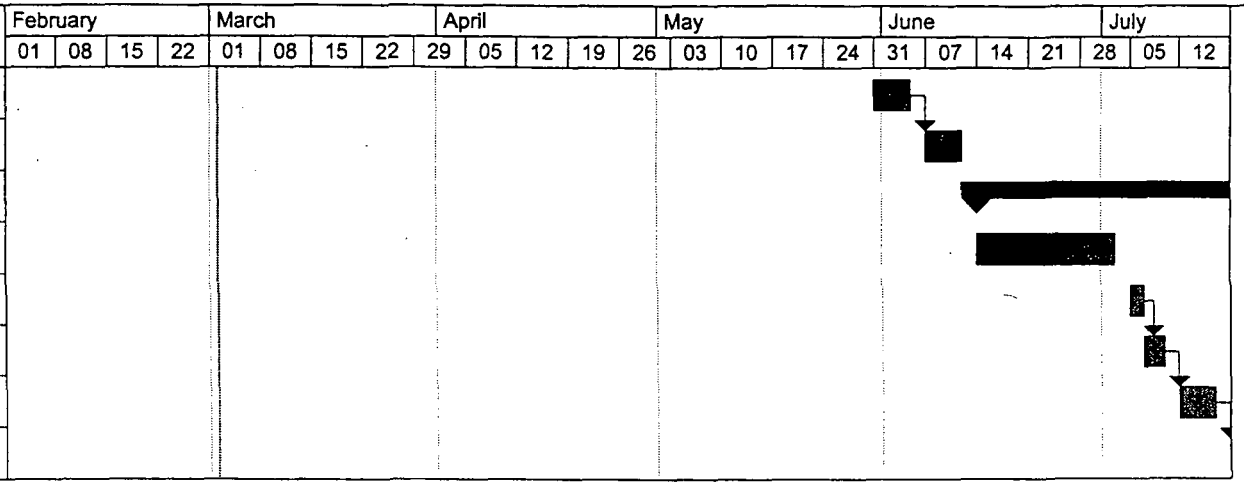
ID	Task Name	Duration
0		########
1	**AutoYama-2 Code Migration**	**125 days**
2	**Set Up Development Environment**	**20 days**
3	set up /lc_dev/ay2	4 days
4	Report Year 2000 Dependencies	3.5 days
5	Document Line Architecture Analysi	2 days
6	ORACLE TRAINING	5 days
7	Reverse Engineer Existing C Code	5 days
8	**Migrate Code**	**65 days**
9	Set Up Database	3 days
10	OC1A	12 days
11	OC2A	12 days
12	Order Server and DTC	1 day
13	Order Hub and Repeater	1 day
14	ADA	14 days
15	MODULES	14 days
16	Install and Configure Server	2 days
17	Eda	3 days
18	**Set Up Hardware**	**2 days**
19	Install cables and PCs	2 days
20	**Installation of AY1**	**10 days**



Project: ay2-plan
Date: Tue 02/03/99

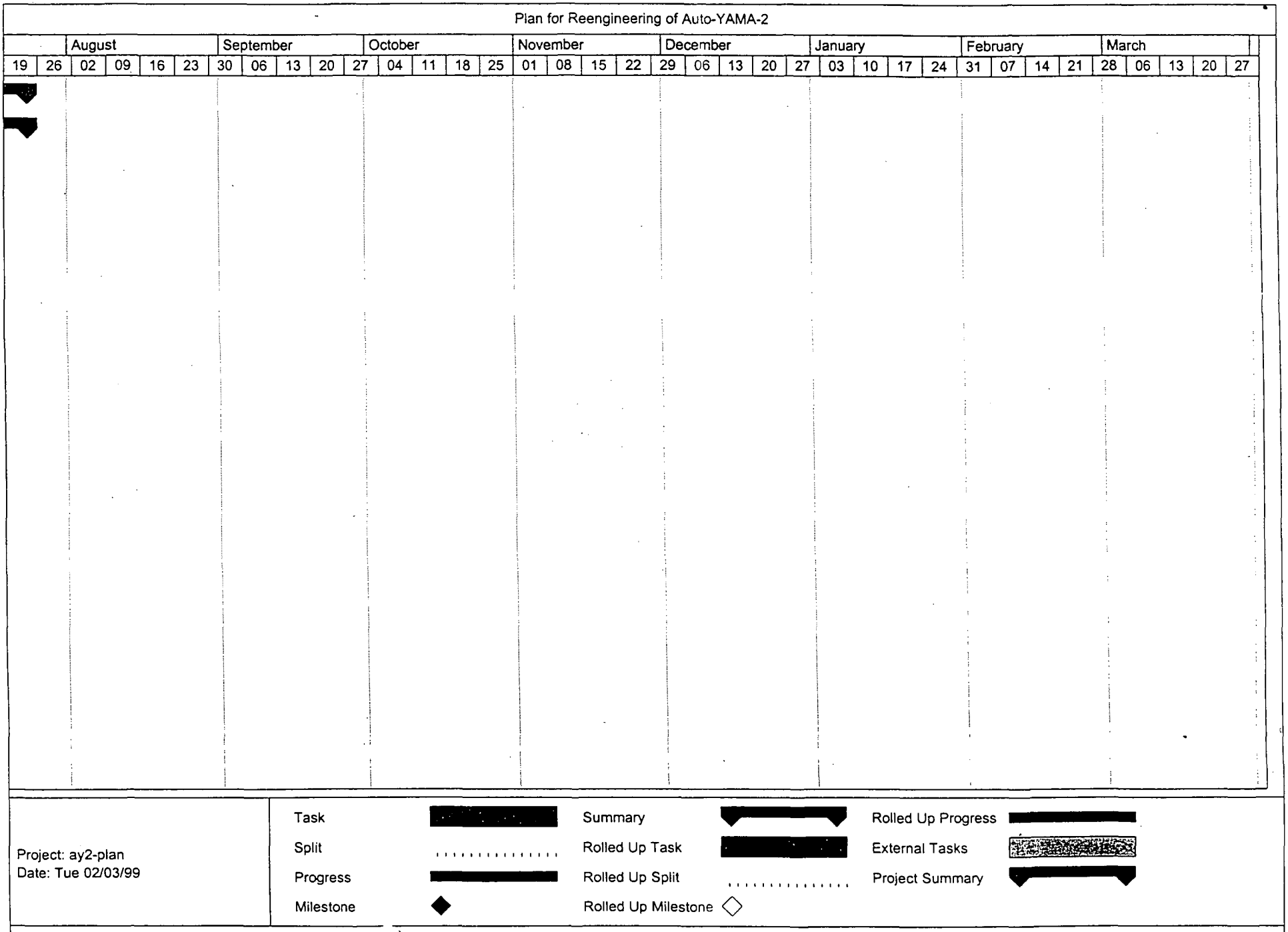
Task
Split
Progress
Milestone
Summary
Rolled Up Task
Rolled Up Split
Rolled Up Milestone
Rolled Up Progress
External Tasks
Project Summary

ID	Task Name	Duration
21	Training	5 days
22	Installation and Trial	5 days
23	**AY2 Code Review**	**30 days**
24	Documentation	15 days
25	Install Software Development Tool	2 days
26	Report on Impact Analysis	3 days
27	Report on Tools for process automa	5 days
28	Auto-YAMA-2 Audit	5 days

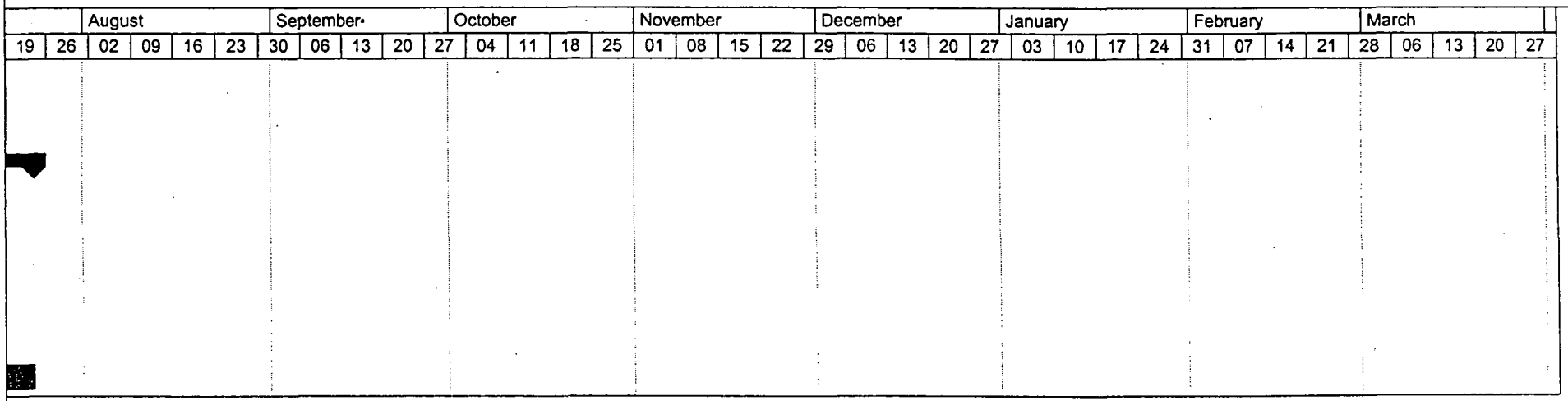


Project: ay2-plan
Date: Tue 02/03/99

Task
Split
Progress
Milestone
Summary
Rolled Up Task
Rolled Up Split
Rolled Up Milestone
Rolled Up Progress
External Tasks
Project Summary


Plan for Reengineering of Auto-YAMA-2
August
September
October
November
December
January
February
March
19 26 02 09 16 23 30 06 13 20 27 04 11 18 25 01 08 15 22 29 06 13 20 27 03 10 17 24 31 07 14 21 28 06 13 20 27
Project: ay2-plan
Date: Tue 02/03/99
Task
Split
Progress
Milestone
Summary
Rolled Up Task
Rolled Up Split
Rolled Up Milestone
Rolled Up Progress
External Tasks
Project Summary

Plan for Reengineering of Auto-YAMA-2

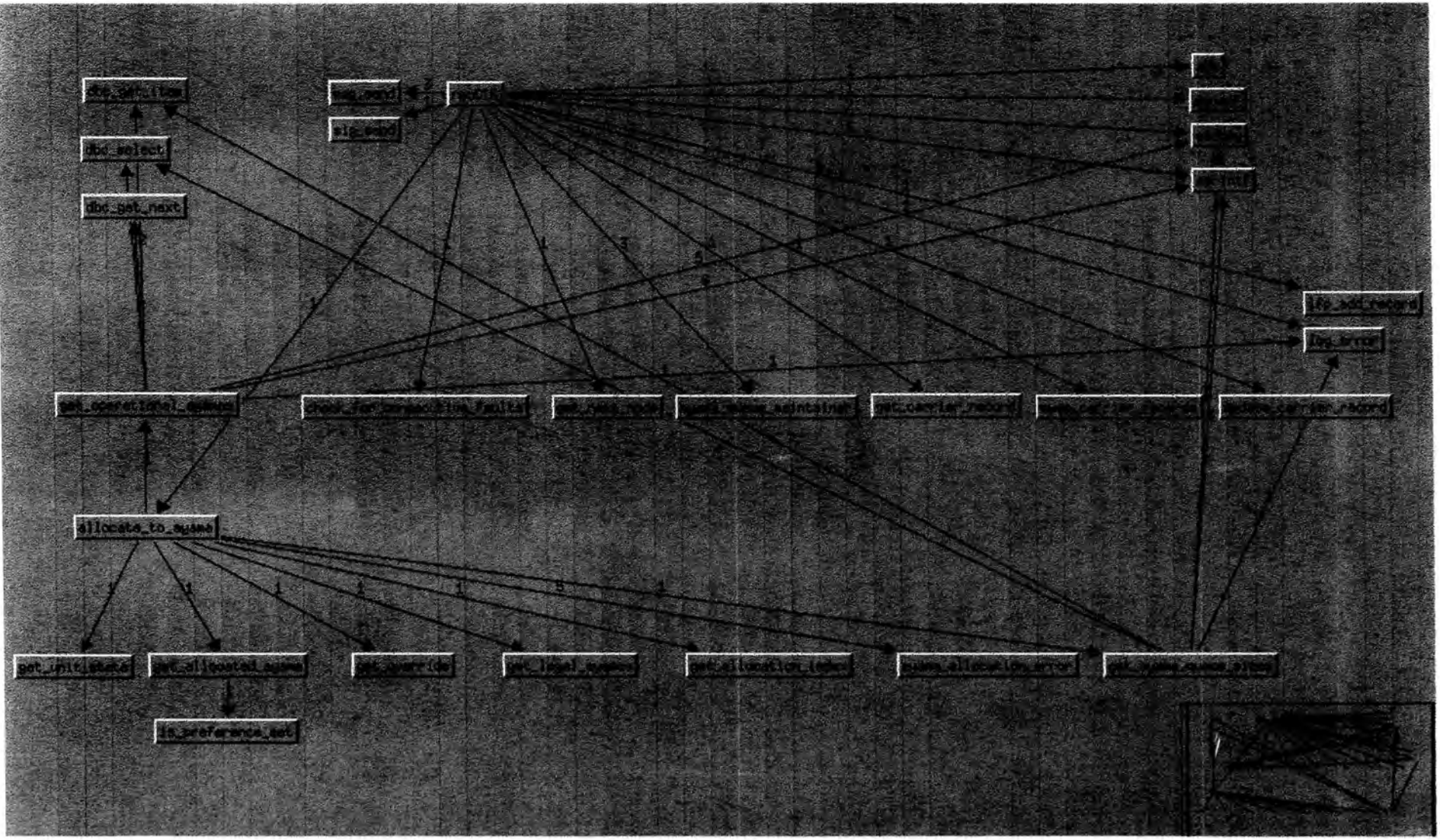


Project: ay2-plan
Date: Tue 02/03/99

Task
Split
Progress
Milestone
Summary
Rolled Up Task
Rolled Up Split
Rolled Up Milestone
Rolled Up Progress
External Tasks
Project Summary

Appendix C3

Sample of Source Code documentation from the A.U. Tool.



A Call-Graph representation produced by the A.U. Tool of the rec016.c function that is used in all of the Production Line Controller Systems.