Automating Reuse Support in a Small Company

Peter J. Biggs

PhD Thesis

University of Durham

Department of Computer Science

Centre for Software Maintenance

Supervisors: Cornelia Boldyreff and Keith Bennett

August 1998

Abstract

Software engineering has been facing a crisis for several years now - there is more demand for new software than there is ability to supply. Software reuse is a potential way to tackle the problems caused by the software crisis with its promises of increased productivity and cheaper development costs. Several software reuse successes have been reported, but these have been predominantly in large, well structured companies. However, there are numerous smaller companies that could also benefit from reuse if it were made available to them.

This thesis addresses these issues by implementing a reuse programme in a small company. An incremental approach to reuse introduction is adopted, following the *Seven Steps to Success*, and 'lightweight' processes are recommended to support the reuse programme. A prototype tool set, ReThree-C++, was developed to automate support for the reuse programme.

The results of the case study are presented. The reuse programme was successful, with benefits to the company including both increased speed of production and financial gains from selling reusable components. The challenges faced are also identified. Details of the tool set giving automated support for reuse are also presented. The tool set is an approach to reuse repository control which also integrates information abstraction from C++ source code to generate class hierarchy charts and software documentation automatically. It helps developers store, retrieve, understand and use reusable components. The usefulness of the tool set is shown with an experiment and as part of the case study.

The purpose of the thesis is to show that small companies can implement reuse, and that the method presented supports the introduction of a reuse programme. It concludes that although challenges were faced, great benefits can be gained by using the method with automated support for reuse in a small company.

Acknowledgements

The author would like to thank Dr. Cornelia Boldyreff and Prof. Keith Bennett for their advice, assistance and encouragement as supervisors for this research.

This work has been funded by the Engineering and Physical Sciences Research Council.

The author would also like to thank Nigel Hope, Andrew Wilson, Steve Anderson, Sary Andiyapan and all the staff at Public Access Terminals for their help, support and contribution towards this research.

Thanks also to Richard Mortimer and Elizabeth Burd for their useful advice and support throughout the time of this research.

This Ph.D. thesis is dedicated to Kenneth and Marian Biggs. My life is their success.

Declaration

This thesis is solely the work of the author, and no part of the thesis has been submitted for a degree at this or any other university. The copyright of this thesis rests with the author. No quotation from it should be published without his prior written consent and information derived from it should be acknowledged.

Table of Contents

List of Figures

Chapter 1: Introduction

This chapter gives an overview of the research conducted in this thesis, including a statement of the problem to be addressed, and the context in which the research has been conducted. The title of the thesis is "Automating Reuse Support in a Small Company". The research has been funded by the Engineering and Physical Sciences Research Council, and has been conducted at Durham University in conjunction with Public Access Terminals Ltd., a small software systems development company.

Over the years since the computer was first invented, there have been many different pieces of software written for various types of machine. Originally, all software was written from first principles, with programmers deciding what was needed, then designing and coding the required system. With computers flooding the business world, the demand for high quality software has increased dramatically. However, the time taken to write software systems has not decreased significantly. This creates a problem: there is more demand for software than there is ability to supply. This problem is generally known as the *software crisis*.

Software reuse (the use of previously written software in the development of new systems) is a potential way to tackle the problems caused by the software crisis, and has been a subject of research for several years now. The reuse of software is a popular concept in the software development industry, with its promises of increased productivity and cheaper development costs. Some successes have been reported, but these have been predominantly in large, well structured companies with the resources available to invest in reuse. This research is based on the thesis that smaller companies, which tend to rate low on the *process maturity scale* (a measure of the quality of the processes used within a company) and do not have the resources available to invest in long-term payback schemes, could also benefit from reuse if it were made available to them.

The research proposes a method for introducing reuse into a small company which recognises that small companies do not have the processes in place or the resources available to carry out a full scale reuse programme. Using a combination of 'lightweight' processes and automated support for the reuse programme, the thesis recommends an incremental approach to the introduction of reuse which cuts the initial investment required and reduces the amount of time which passes before the benefits of reuse can be realised.

A case study using the method is conducted in association with a small company. Using the working relationship with Public Access Terminals Ltd., this research considers the challenges which are unique to a small company and investigates the validity of the method in an industrial environment. This has enabled the research to address a real problem, which has not been very well considered in software reuse research, namely the combination of technical, organisational and logistical challenges which face a small company wishing to implement a software reuse programme.

The research method has been based on Colin Potts' recommendations for using 'industry as a laboratory' [Pott93]. Potts suggests that research should address real problems faced by industry in order to facilitate improvements, rather than the common 'research-then-transfer' approach which is usually attempted with varying degrees of success. He recognises the importance of revolutionary research, but emphasises also the importance of evolutionary research.

This chapter gives an overview of the structure of this thesis, followed by an overview of the research which has been performed, a statement of the problem addressed, the context in which the research has been conducted and criteria for success.

Chapter 2 discusses the concepts of software reuse. It considers what software reuse is, why it is advantageous to do it, what technologies are currently available to support reuse, why reuse is not practised and the difficulties involved in the introduction of reuse into the software development process.

Chapter 3 looks at technologies which support the introduction of a reuse programme in a small company. First, there is consideration of how a small company is defined. This is followed by a section which looks at the techniques which can be used when changing the way in which a company works, covering the fields of organisational development and process improvement. The applicability of object orientation for reuse is considered and reverse engineering and software documentation for reusable components are also investigated.

Chapter 4 looks at some of the successful reuse programmes which have been reported and then considers some of the alternative methods which were available for conducting this research. There are various approaches to the introduction of a software reuse programme in a small company. These alternatives are considered, along with the course of action which was chosen for this research.

Chapter 5 describes the method which was chosen for the introduction of a reuse programme in a small company, given as the *Seven Steps to Success*. It has been seen in other studies that the support of top level management is vital to the success of a software reuse programme. The initial work, therefore, involves presenting the case for reuse to the top level management. Following this, a study of the company's current working practices is conducted, which leads to recommendations being made to the company for techniques which would help the introduction of a reuse programme. These techniques are consolidated into a plan, including a pilot project to test the implementation of the techniques. The results of the pilot project are collected and studied to identify successes and shortcomings of the approach adopted. Based on the results of the pilot, a plan for reuse is formulated and implemented within the company with associated automated support.

A case study implementing the *Seven Steps to Success* is described in Chapter 6. This study introduces 'lightweight' processes to the company, integrated with a tool set which aids the automation of some of the 'lightweight' processes suggested for the software reuse programme.

The method recommended for introducing reuse into the software development process makes reuse available without the large initial investment which is usually required for a successful reuse programme. The tool set provides support for object-oriented design, reverse engineering, software documentation generation and support for a reuse repository in an integrated environment. The combination of tool support for these areas and the 'lightweight' processes help to reduce the initial effort required when introducing reuse into a small company.

In Chapter 7, the results obtained from the case study are discussed. Success is identified in terms of benefits to the company, and the problems encountered are also identified. The tool set developed is evaluated as a part of this research and also by experimentation. The results of these evaluations are collected and considered.

The final chapter of the thesis gives conclusions about the research carried out. The results of the research are discussed, along with criticism and further work.

The purpose of this thesis is to investigate the practicality of implementing software reuse as part of the development of software within a small company, and to identify those features of the working practices studied which are unique to the small company environment. During this investigation, ways to improve the development strategies used in small companies are identified, and considered for their applicability in this case study. 'Lightweight' processes with automated support for reuse are suggested and evaluated as ways to assist reuse in the small company environment.

Software reuse is considered as a key topic for investigation due to the improvements in productivity and profitability which can be derived from the implementation of a software reuse programme. Of considerable interest is the actual process of introducing a reuse programme into an environment where there are no standard processes currently defined. The difficulties of this task are considered and ways to improve the introduction of reuse into such an environment are suggested.

This thesis approaches the challenges in software reuse by adopting a practical approach to the implementation of, and automation of support for, a reuse programme in a small company. It makes two important contributions to the field of software reuse research:

1. A method for introducing reuse in a small company with a real case study of the implementation of a software reuse programme in such a company. The programme is described in terms of the recommendations made, the work done, problems encountered and success achieved.
2. A practical, fast and simple to use tool for automating reuse support in a small company. This tool aids in storing and retrieving reusable components, as well as reverse engineering and re-documenting source code to provide information about the reusable components.

1.1 Overview

The benefits which the reuse of code can bring have interested many software development companies, and studies have been conducted considering the challenges facing companies setting up a reuse programme. Both successes and failures have been reported, but the successes tend to be in large, well structured companies with the resources available to invest in reuse [Bigg89b]. Smaller companies could also benefit from the reuse of their software if the principles and techniques which support software reuse were made more available to them.

This research attempts to make reuse available on a smaller scale by encouraging small company developers to design with reuse in mind. This will help to make reuse available 'in the small', where companies do not have the processes in place, or the time and resources, to support highly structured reuse frameworks. There is consideration of reverse engineering and re-documentation to allow developers to see how object-oriented (OO) design and software documentation can aid them in understanding their previous developments. This will encourage more structured development processes and help in the maintenance and reuse of current code.

A study of several OO design methods has been conducted, and their applicability to software reuse was considered. The study suggested a notation for representing the structure of a software system. This allows the ideas to be applied to C++ source code, giving a diagrammatic representation of the classes within the code, as well as the structure of the inheritance hierarchy. The representation is associated with a system of using information taken from the static analysis of the source code to generate documentation for the code automatically, based on the comments within the code. This automatically generated documentation is used to index and classify code components for a reuse library. The integration of the tools for reverse engineering, re-documentation and reuse support form an integrated environment automating support for reuse in a small company.

1.2 Statement of Problem

It has been shown that software reuse can offer great benefits to companies when used effectively. Many success stories have been quoted, from Raytheon's 50% increase in productivity due to a reuse rate of 60% [Lane84], to GTE's saving of $1.5 Million from a reuse factor of 14% [Prie90], to the Japanese software factories' claim of annual productivity increases of 20% by implementing a software reuse programme [Mats84].

It would be foolish to claim that software reuse is the solution to all the problems that have caused the current software crisis. Achieving software reuse on a level at which substantial benefits will be gained is a difficult task, and requires a great deal of commitment and effort. Tracz [Trac88b] emphasises that "there is no free lunch when it comes to software reuse". There are, however, techniques which can help a company to maximise its resources and improve its productivity. It has been shown that reuse offers great benefits if used effectively in the right environment; but this raises the questions: how are software reuse techniques best employed; and what is the right environment for software reuse to prosper? All the published examples quoted above have been large, well structured companies, with top level management support

for the reuse programme. This suggests that software reuse tends to prosper in such an environment; but what about the small, less structured companies, whose livelihood depends on the ability to produce their product as quickly as possible, while trying to keep standards high and their maintenance costs low? To them, the benefits of software reuse could be invaluable.

There are two major factors that inhibit reuse which will be considered in this research. The first factor is classified as the technological issues in reuse. These include the lack of reusable components available to a developer (either because they do not exist, or they are not easily available), that the parts needed cannot be found, or cannot be understood or integrated into the current system even when they are found. The second factor includes managerial and sociological inhibitions, otherwise known as organisational issues. These are evidenced by the lack of processes to support reuse, the lack of commitment to a reuse programme and the *NIH (not invented here) syndrome*, where developers are wary of using code that they have not written themselves.

The research will address the first factor by making reuse readily available through automated support for the reuse process. This will aid the identification, classification and retrieval of reusable components. Guidelines will also be made on introducing reuse in the small company environment using 'lightweight' processes, which are tested in practice in an industrial collaboration. This addresses the second factor.

1.3 Context of Work

The basis of this research has been a method called "industry-as-laboratory", as recommended by Colin Potts [Pott93]. This involves bringing researchers into close contact with industry, so that real problems can be identified first-hand. He suggests one of the reasons that the current state of practice in industry is so far behind the leading research being done in software engineering is that the "research-then-transfer" approach has been so predominant in the research community.

In line with this suggested research technique, a project has been undertaken in collaboration with a small software development company, Public Access Terminals Ltd. (P.A.T.). This allows the method for the introduction of reuse proposed within the thesis to be evaluated in practice. It has already been shown that there is a lack of reported research on software reuse within small companies. The examples quoted in section 1.2 are large companies whose structured processes have been updated to incorporate reuse. Small companies tend to have ad-hoc and unstructured processes for software engineering, yet still have considerable success in the market place.

There is also a lack of research being done into how reuse can be capitalised upon in an environment where there are no structured processes available with which to integrate a reuse programme. Interesting results have been gained from the work done with P.A.T. in their small industrial environment. It has been found that ad-hoc reuse is a standard practice. The problems often associated with reuse, such as the storage and retrieval of reusable components are much less significant because of the small scale on which their work is done, and the narrow domain of the software being developed. However, in order to achieve further benefits from reuse, it has been seen that more formalised 'lightweight' processes can be introduced.

1.4 Criteria for success

In evaluating the results of the method for implementing a reuse programme discussed in this thesis, the following criteria for success will be used. Three main issues will be considered:

1. Is the method for introducing a reuse programme successful? Success for a reuse programme can be measured in many ways. However, the most clearly identifiable measure of success is identifying whether reusable components are built, and to what extent they are reused.

2. Does the method bring benefits to a small company? As identified later in section 2.4, benefits will be considered in terms of:
    - Increased speed of production
    - Financial benefits
    - Increased quality of software
    - Ease of maintenance
3. Does automated support aid a reuse programme? The automated support will be considered in terms of the benefits brought to a reuse programme and its usefulness within a small company.

In order to measure these considerations, a reuse programme using the method was implemented in a small company. An experiment was also conducted to investigate the automated support for a reuse programme. The research will be considered successful if all three questions posed above can be answered affirmatively.

Chapter 2: The Field of Software Reuse

2.1 Introduction

This chapter gives an overview of the field of software reuse. It starts by defining what reuse and reusability are, then goes on to discuss why reuse is important and some of the motivations for doing it. The benefits which come from successful reuse are outlined, followed by the two major sets of issues which must be addressed in order for reuse to be successful. These are technological and organisational. Technological issues are discussed in detail, focusing first on the reuse of software components, then looking at the reuse of higher level components. The section on organisational issues considers some of the psychological, sociological and economic factors which can affect a reuse programme. The chapter finishes with a review of the points from the survey of the field of software reuse which are important in this thesis.

2.2 Definitions of Software Reuse

The concept of software reuse has slowly developed over the past 30 years as time, research and experience have modified people's perception of the idea of reuse. The first recognised publication on reuse in software engineering is McIlroy's [McIl68] view foreseeing software development becoming the process of constructing software from standard interchangeable building blocks. He suggested that the software components industry should be comparable to the hardware components industry. He says: "I would like to see the study of software components become a dignified branch of software engineering. I would like to see standard catalogs of routines classified by precision, robustness, time-space requirements and binding time of parameters." This view mainly considers the reuse of source code.

Freeman [Free83] expanded this view of reusability to cover a greater area of the software development process, when he said: "This leads us to define the object of reusability to be any information which a developer may need in the process of creating software." He goes on to describe five main levels at which reusability should be considered, namely: code fragments, logical structures, functional architectures, external knowledge and environment-level information.

In the Software Engineer's Reference Book, Hall and Boldyreff [Hall91] describe software reuse as "the use of a given piece of software in the solution of more than one problem". They go on to further clarify their view by explaining what they do not consider to be classified as software reuse: portability, maintenance and reconfigurability.

Perhaps the simplest definition of reuse was offered by Prieto-Diaz and Freeman [Prie87], when they stated that "Ideally, reuse is a matching process between new and old situations, and, when matching succeeds, duplication of the same actions". They suggest that two levels of reuse should be considered: "(1) the reuse of ideas and knowledge and (2) the reuse of particular artefacts and components".

Hooper and Chester [Hoop91] state that: "Two possible definitions of software reusability are:

1) the extent to which a software component can be used (with or without adaption) in multiple problem solutions;
2) the extent to which a software component can be used (with or without adaption) in a problem solution other than the one for which it was originally developed.

Definition 2 tends to suggest that reuse is incidental to the development process, whereas definition 1 tends to suggest that reuse is a worthy goal in and of itself, and therefore requires planning and effort to achieve it."

Tracz [Trac90] would argue that definition 1 above is the only definition of software reusability, saying that, "software reuse ... is the process of reusing software that was designed to be reused". He goes on to describe "software salvaging, that is reusing software that was not designed to be reused", which would be his description of definition 2 above.

Yu [Yu91] defines reuse as "software engineering activities which focus on the identification of reusable software for straight import, reconfiguration, and adaption for new computing system applications". He goes on to describe the connection between reuse, re-engineering and reverse-engineering: "Once successfully reverse-engineered usable parts from existing software systems and re-engineered these parts for a project specific adaptation, this success will then qualify for a case of apparent software reuse. Software reuse may depend on the reverse-engineering and re-engineering technologies, although software can be written such that it can be easily reused without the need of these two technologies."

Bollinger and Pfleeger [Boll90] expand on this view of reuse in their definition. "Reuse is the process by which software work products (which may include not only source code, but also products such as documentation, designs, test data, tools, and specifications) are carried over and used in a new development effort, preferably with minimal modification."

For the purposes of this research, reuse is defined as the use of any previously written software work product in the development of a new software system, whether the work product has been specifically designed for reuse, or salvaged from some previous development.

There are many examples of software reuse already available in the software industry. Some of the more common are: the libraries associated with windowing systems, mathematical subroutine libraries and clip art. These are all software components that can be used with or without modification in numerous different applications.

2.3 Motivations for Reuse

The motivations for reuse are well summarised by Geary [Gear88]: "There is a finite limit to the amount of software that can be developed annually from first principles, but customers continue to make ever increasing demands for new software. Often, new software is similar to other software developed elsewhere, but with sufficient difference in design and function to make the existing software unsuitable for reuse in the new design without modification of either the new design or the module. Invariably, in this situation, the decision is taken to design new software. If suppliers are to meet ever increasing demands, vast gains in productivity must be accomplished."

Software reuse is an area of software development that is becoming increasingly significant. Arthur [Arth83] quotes Johann W. von Goethe, who said, "Everything has been thought of before, but the problem is to think of it again". This has been confirmed by software development practices. In a study done at the Raytheon Missile Systems Division of the U.S. Department of Defence, it was found that between 40-60% of their applications were doing essentially similar functions which could be standardised into a fairly small set of "standard reusable modules" [Lane79]. Kapur, quoted by Jones [Jone84], when studying commercial banking and insurance applications, noted that about 75% of the functions were common to more than one program. Jones goes on to make a tentative conclusion that "of all the code written in 1983, probably less than 15 percent is unique, novel, and specific to individual applications."

There is one main aim in the implementation of a reuse programme, and that is increased productivity. The reason that software reuse is becoming such a popular concept is that it promises faster software development processes and decreased development costs. In a small company environment, the speed of production and time to market are critical factors for success.

However, in spite of the interest shown in the reuse of software, there has not been a great deal of work on the actual implementation of systems to support reuse. Maarek [Maar90] says "Although software reuse presents clear advantages for programmer productivity and code reliability, it is not practiced enough. One of the reasons for the moderate success of reuse is the lack of software libraries that facilitate the actual finding and understanding of reusable components." DeMarco [DeMa84] estimated that in the average software development environment, only about 5% of code is reused.

2.4 Benefits of Reuse

There are four main areas where the introduction of a reuse programme can benefit a company willing to commit their time and resources to the success of the programme. These four areas are:

- increased productivity
- reduction of development costs
- increased quality of software
- ease of maintenance

Each of these benefits will be considered in more detail in the following sections, with reference to reported reuse successes where available.

2.4.1 Increased Productivity

Reuse offers significant increases in productivity because software is simply being composed from what is currently available, rather than being produced from scratch. Therefore, every software component that is reused is one that does not require effort to produce. Jones [Jone86] reports productivity increases in the range of 50% for projects with high levels of reuse. Lanergan and Grasso [Lane84] tell how the Information Processing Systems Organisation of

Raytheon's Missile Systems Division experienced a 50% increase in productivity through the standardisation of functions and logic structures into reusable modules. Matsumoto [Mats84] reports a 20% increase in productivity per year (measured in terms of lines of code per month) in the field of telephony and process control software, an area in which reuse sceptics doubted the possibility of reuse. Prieto-Diaz [Prie91] quotes Fujitsu's experience, in which they found that after introducing reuse techniques, the number of projects which were completed on schedule rose from 20% to 70%.

2.4.2 Reduction of Development Costs

Less effort and time in software production leads to a reduction of development costs. This has been shown in practice. In 1987, GTE [Prie90] achieved a reuse factor of 14% in their software development, which saved the company $1.5 Million in software development costs. They were predicting that by 1995, they would be experiencing a reuse factor of 50%, which would save them a total of $10 Million.

2.4.3 Increased Quality of Software

If high quality components are being reused, then the resulting software should meet high standards. IBM achieved a reuse factor of 50% in their software development. They also found that, along with this, they obtained an order of magnitude improvement in errors detected.

2.4.4 Ease of Maintenance

Tracz [Trac87a] claims that the greatest payoff from reuse is realised in ease of maintenance, and a corresponding reduction in maintenance costs. This is because, as mentioned in section 2.4.3, if high quality components are reused, the resulting system will be of a high standard. Also, reusable components should be well abstracted and documented, allowing them to be

easily understood. He reports maintenance cost reductions of up to 90% when reusable code, code templates and application generators have been used to develop new systems.

2.5 Issues in Reuse

There are many reasons for the lack of reuse. Standish [Stan84] recognises that there are two main divisions to the issues associated with reuse, namely technological and organisational. Tracz [Trac90] focuses on three areas in reuse when he introduces "the three P's of software reuse: product, or what do we reuse, process, or when do we apply reuse, and finally personnel, or who makes reuse happen." Basili et al. [Basi87] also consider the same three areas, further characterising them as "the reuse of knowledge that exists only within the minds of people (informal knowledge), reuse of specified plans on how to perform certain activities or structure and document certain products (schematized knowledge), and reuse of tools and products (productized knowledge)."

It has been recognised that there are several pre-conditions which must be met in order for a developer to be able to incorporate a reusable component into their software system [Frak92]. These are:

1. The component must exist.
2. The component must be available to the developer.
3. The developer must be able to find the component.
4. Once found, the developer must be able to understand the component.
5. Based on an understanding of the component, the developer must identify the component as being valid for the current system.
6. The developer must be able to successfully integrate the component into the current system.

It can be seen that the reuse of a component is no easy task. Many different techniques must be employed in order for these pre-conditions to be met. These include structured software

engineering [Somm96] to provide reusable components (pre-condition 1), component repositories [Wolf92] to store software components (pre-condition 2), indexing and library searching techniques [Frak88],[Prie87] to facilitate repository searching (pre-condition 3), program comprehension to help in understanding the components (pre-condition 4), systems analysis to identify the component as one which will fit within the current system (pre-condition 5) and finally structured interfaces and systems integration techniques to allow the developer to incorporate the component into their system (pre-condition 6).

The next two sections will consider the technological and organisational issues associated with reuse in more depth.

2.6 Technological Issues

The technological issues in reuse cover many different areas, ranging from domain analysis to the creation of reusable components to the storage and reuse of those components. Wirfs-Brock et al. [Wirf90] suggest that there are three types of software entities that can be reused:

- components - these are atomic entities that can be used in a number of different programs. Examples cited are lists, arrays, strings, radio buttons and check boxes.
- frameworks - these are skeletal structures of programs that must be fleshed out to build a complete application. A cited example is a windows system, which is a framework on which windows applications can be built reusing the windows principles on which the system is based.
- applications - these are complete programs. Cited examples include word processors and spreadsheets. Modern spreadsheets are good examples of reusable applications, because they are generic enough to be used in many different application domains.

In this section on the technological issues in reuse, it is mainly the reuse of components and frameworks which will be considered. Hooper and Chester [Hoop91] state that reuse can be

considered on two levels: *horizontal reuse* and *vertical reuse*. Horizontal reuse is reuse across a broad range of application areas or domain, whereas vertical reuse is the reuse of components within a given domain. It is suggested that horizontal reuse, such as mathematical subroutines and input/output function libraries, has been the most successful form of reuse thus far; however, the greatest potential benefits are seen in vertical reuse.

In order to capitalise on vertical reuse, domain analysis should first be performed. *Domain analysis* is a technique in which an application domain is studied and the information gathered is analysed in order to understand the problem domain, and investigate the potential for reuse. Kang [Kang89] describes it thus: "Domain analysis is a phase of the software lifecycle where a domain model, which describes the common functions, data and relationships of a family of systems in the domain, a dictionary, which defines the terminologies used in the domain, and a software architecture, which describes the packaging, control, and interfaces, are produced. The information necessary to produce a domain model, a dictionary, and an architecture is gathered, organized, and represented during the domain analysis."

It is vitally important to include domain experts when conducting a domain analysis. These are people who work in the domain being studied. Their knowledge of the domain is therefore unrivalled. Without such domain expertise, it will not be possible to do the domain analysis in sufficient depth. Hutchinson and Hindley [Hutc88] report that the goals of their domain analysis were:

- to discover the functions that underwrite reusability,
- to focus the domain specialist's attention on reuse,
- to help the domain specialist ascertain reuse parameters,
- to discover how to redesign existing components for reuse,
- to organise a domain for reuse.

Once the domain analysis has been conducted, commonalities in the software development process can be discovered, and potential areas for reuse identified. Domain analysis is a pre-requisite for vertical reuse, or reuse in a single domain. Once the commonalities in development and, thus the potential for reuse, have been discovered, these can be capitalised on by the production of reusable components and their storage in an appropriate repository. Domain analysis is a costly process, but the benefits that can be derived from it are very worthwhile.

There tend to be fewer domains to be analysed in a small company, and the domains are often narrow, making domain analysis an easier task. However, with the small amount of information available about the domains, it is much more difficult to cross reference the work being done to identify commonalities and capitalise on useful abstractions. Vertical reuse is the most obvious type of reuse in the small company environment. Horizontal reuse would be limited by the small number of domains under consideration.

2.6.1 Reuse Technologies

Biggerstaff and Richter [Bigg87] state that the approaches to reusability can be classified into two basic groups: composition technologies and generation technologies. The former are characterised by the fact that the components are atomic and, ideally, unchanged in their reuse. The latter are not so easily identifiable as entities, but their reuse is more a matter of execution than composition, as is the case of reusing design principles encoded in an application generator.

2.6.2 Composition technologies

Moineau et al. [Moin90] note two main problems in the area of composition technologies: "the first is the specification or description of the component so as to allow easy retrieval and so that the user can understand it properly for future adaptation. The second problem is the definition of the composition principles by which components are combined into target systems."

Burd and McDermid [Burd93a] conducted a study of the factors which limit the appeal of reuse to project managers and software developers. They found that these inhibiting factors include, at a technical level:

- **development** - Knowing what kind of software is reusable, and equally difficult is knowing how to develop a software component which is potentially reusable.
- **storage** - Once we have developed an item of reusable software, how, and where, should it be retained for future retrieval and reuse?
- **retrieval** - If we are to reuse software then we must be able to easily find what we require, matching what is available with our needs.
- **verification** - How can we be sure that the component which we are proposing to reuse actually performs the functions that it claims it does in the environment in which we use it?
- **evaluation** - How can we judge that the functions we require from our reusable unit and the functions that it provides are the same?
- **modification** - If our evaluations have shown that differences exist between the reuse units and our requirements then how do we perform the necessary modifications, and what effect will this have on the reusable unit including the results of previous verifications?
- **integration** - What will the effect be on the reusable unit of attempting to integrate it in our development?

These factors are very similar to steps leading to successful reuse given by Frakes [Frak92]. He considers that "every software lifecycle object that is created from scratch or is modified is, in a sense, a reuse failure". Each of the above factors will be considered in greater depth.

Development

There are two ways of producing reusable components: either extracting them from code already written, or designing them from scratch. As noted earlier, Tracz [Trac90] considers the former to be 'software salvaging' and emphasises the importance of planning for reuse. Biggerstaff and

Perlis [Bigg89a] consider the size of reusable components. They note that small components will be less specific, and therefore more reusable. It would, however, take many of these small components to create a software system. It would also mean that a lot of work would be needed to integrate the large number of components required within the software system. On the other hand, if the components are large, they will be more specific, and therefore less reusable. However, the benefits gained from the reuse of a single large component will be greater.

Weber [Webe91] suggests that all reusable code should look alike, and recommends the 'Canonic Software Component', which could be a standard for all software components. He goes on to suggest the idea of a Concurrently Executable Module (CEM), which has four constituent parts: the export interface, the body, the import interface and the common parameters. These should be standardised throughout all modules in a reusable library, in order to allow different parts to 'plug together'. This concept is often known as 'black-box' reuse [Prie93].

Storage

The issues associated with storage are: what should be stored in a component library and how they should be stored. Wolff [Wolf92] claims that the 80/20 rule applies to a software components library. He says "the rule applied to reuse says that 20% of the components will bring 80% of the reuse savings. Most of the other 80% of the reusable parts make the library and the tools acceptable to the developers." Therefore, within reason, it is wise to add as many components to the library as possible, provided they meet acceptable quality standards.

In terms of how the components are to be stored, one of the most popular suggestions to have emerged is that of faceted classification. This is where the library space is dynamic, and components are assigned 'facets' dependent on their main features. It is the combination of these facets that is used to classify the component. If no facet combination currently exists in the library space to support a new component, then a new section is added for it. Prieto-Diaz and

Freeman [Prie87] extol the advantages of using this approach: "Faceted schemes are more flexible, more precise, and better suited for large, continuously expanding collections."

Retrieval

Geary [Gear88] suggests that effective methods of searching software libraries would be essential to software reuse: "A large library of software components would be too vast to commit to human memory. To be a success, a software component library must be supported by comprehensive search, retrieval and design tools that are able to assist the designer in creating a design that takes advantage of available components." Frakes [Frak88] says: "A fundamental problem in software reuse is the lack of tools to locate potential code for reuse." He goes on to argue that information retrieval systems have the power and flexibility to ameliorate this problem. Maarek [Maar90] discusses the differences between an Artificial Intelligence, or knowledge-based approach to reuse library support tools (such as [Prie87], [Alle89]) and an Information Retrieval approach. She notes that the AI approaches are often 'smarter' than the IR systems; however, they rely far more heavily on domain analysis, which can rapidly get out of hand as the library grows. She opts for the IR approach, considering that it "presents clear advantages over the AI approach in terms of human cost, portability and scalability."

Although a significant amount of work has gone into researching this area of software reuse support, it is perhaps one of the less critical areas. The Japanese software factories claim that they have been achieving reuse factors of up to 85% using only simple keyword searching techniques on the components in their repositories [Stan84]. This would suggest that other areas hindering reuse need to be addressed as well as the retrieval issues associated with software libraries.

In a small company, the size of the reuse repository is likely to be small, making overly complicated storage and retrieval procedures too cumbersome and time consuming. It would be better to have a repository that is simple to use and requires little effort to maintain.

Verification

This is a difficult issue. It is very hard to prove that a particular software component does what the associated documentation claims it does. However, without faith in the documentation, the time consuming process of inspecting the component in detail must be carried out. This is obviously not desirable. Frakes and Nejmeh [Frak88] recommend that with each component, reuse statistics and reuse reviews be kept. These would record how many times a component has been reused, and how the reusers felt about the component. If the component has been reused successfully in a similar way to the current developer's intended use, the reviews would either instil or reduce confidence in the component, based on the experiences of others.

Evaluation

To evaluate the quality of a reusable component, Tracz [Trac87b] recommends keeping a maintenance record with each component, which would record such things as the type, date and severity of any problems discovered with the module, and whether those problems have been resolved. By considering such a record, a potential reuser will be able to gain a better appreciation for the quality of the component, or the lack thereof. This technique, and the one described by Frakes and Nejmeh (referenced above) both rely heavily on the reusers and maintainers of a product to be conscientious in filling in the associated documentation when the component is reused/modified. Also, they would not help the first developer who wishes to reuse a particular component.

Modification

This is a very important issue in a reuse oriented environment, and can be considered to be an organisational issue as much as a technical one. If it is discovered that a component within the repository requires a change, how should the change be done, and what should be done to

inform users of this change? The first of these questions is a technical issue, and would be considered maintenance of the component. The second presents far more of a problem. Utilising Tracz's [Trac87b] idea of the maintenance record, future users will be informed of the change. However, what about those developers who have already reused the unmodified component. Should they be informed of the change, and if so, what mechanisms should be used to inform them? Babisch [Babi86], in his book on Software Configuration Management, calls this scenario 'the double maintenance problem'. He notes that changes must be made identically in all copies of the software to prevent a proliferation of multiple versions. He recommends the first principle of configuration management, which is "to avoid multiple copies of the same information". This, however, would defeat the purpose of reusable software. Three possible solutions are:

1) to give no guarantee on any software taken from the repository. The software is 'sold-as-seen', and once it has left the repository, the responsibility of the repository administrator and component creator end. This avoids the problem rather than solving it.
2) make information concerning the change publicly available. It is then up to developers who have reused the component to find out about the change, and take necessary action if they so desire. This would work only if the information on changes to software in the library managed to reach all the users of the repository.
3) keep track of all developers who have used a particular component, and inform them directly of any modification information. This is a far more complete approach, and the improved communication between users and administrators of the repository should mean that the repository will be more responsive to change. It would, however, be a huge configuration management problem for the repository administrators, and would create considerably more work for them.

Integration

Work has been done in the area of integration of components, such as the development of module interconnection languages. The Library Interconnection Language (LIL) proposed by

Gougen [Goug86] is a good example. LIL is a language for defining the way in which software components should be 'plugged together'. Module interconnection languages will work only with components that are highly encapsulated and have well defined interfaces.

2.6.3 Generation Technologies

Biggerstaff and Perlis [Bigg89a] distinguish three subclasses of generation based reuse systems:

- Language based systems
- Application generators
- Transformational based systems

Language based systems are those in which the specification language is "well defined, truly represents a problem domain...and hides the details of implementation from its user." A prime example of a language based system is SETL [Schw86], a language which represents computations as operations on mathematical sets.

Application generators are systems which capture a commonality within architectural patterns, and reuse the pattern to produce instances of a particular application type. Prime examples of this kind of reuse are lex and yacc in the UNIX[1] system. These are tools which have captured the commonalities in lexical analysis and compilation to provide a means by which applications of these types may be generated.

Transformation based systems work on the principle of generating a product by successive application of transformation rules. Cheatham [Chea84] describes transformation based systems as having two main mechanisms for refining an abstract program into a concrete, executable program. The mechanisms are: definition and transformation. The abstract program must be defined in a machine processable form. Cheatham describes this as "providing a binding (or

value) for a procedure, type, data object, or what have you." The transformation is based on a set of transformation rules for "replacing some high-level construct by a (more) concrete construct that realises the intended function." Cheatham has done experiments in two settings, rapid prototyping and custom tailoring. He found that the techniques are a valuable alternative to conventional programming techniques.

The REFORM project generated a transformation tool [Bull94] for translating legacy code into an abstract wide spectrum language, then uses transformations to re-structure that code. The structured software can then be translated back into the original software language. Mortimer [Mort96] describes further work on the tool to include transformations for data structures. This allows legacy code to be reused by transforming it into structured software.

2.6.4 Reuse of higher level components

Source code is not the only object of reusability. Reuse can, and should, be done at higher levels of abstraction in software development if real benefits are to be gained. Jacobson et al. [Jaco92] suggest that "what can give even higher productivity enhancement is reuse in other development phases. Other parts of the construction phase may benefit when reusing entire designs in several systems. Additionally, reuse should also be viewed as natural during analysis and testing."

Atkinson [Atki91b] suggests two distinct activities that need to be considered in object-oriented design: "how to produce software components with maximum potential for reuse - design *for* reuse - and how to design new systems making the most effective use of such components - design *with* reuse." Meyer [Meye94] agrees with this basic classification, calling the two categories "reuse consumers and reuse producers". He feels that the two are not disjoint groups.

Tracz [Trac90] notes that software reuse generally ends by using code, but may start at higher levels of abstraction, depending on:

[1]UNIX is a trademark of Bell Labs.

1) how much effort an organisation is willing to invest in preparing products for reuse
2) how effectively higher-abstraction products can be linked to available implementations
3) how effectively implementations are generalised
4) how effectively the software process supports software reuse

Chao [Chao93] questions the maturity of software reuse technology. He suggests that "the methodologies to implement reuse have not been fully developed, tools to support a reuse process are lacking, and standards to guide critical software reuse activities have not been established."

It is much more difficult to reuse components at higher levels of abstraction. However, the benefits that can come from reuse of a high level component can make the extra effort worthwhile. For example, if a design component is reused, then the code associated with that component can also be reused without any further work. However, for this type of reuse to be successful, there must be traceability between the different levels of abstraction [Mats84]. When traceability is maintained, code components meet their requirements, and are implemented as specified in their design, developers can be confident that they can incorporate the component into their system based on the specification of the component's functionality. These greater benefits are only available in a structured software development environment with well defined processes for each stage of the software lifecycle.

2.7 Organisational Issues

The organisational issues of reuse are perhaps the more difficult to tackle. Tracz [Trac88a] notes that "if one looks at the most-often-stated reasons why software is not reused, the overwhelming majority of them may be classified as psychological, sociological, or economic." He goes on to suggest that "the development of software reuse has been stunted by intra-company and inter-company legal, contractual as well as political conflicts."

Chao [Chao93] feels that organisations "face numerous challenges to effectively implement and practice reuse. An organization must make a significant commitment to reuse because fundamental changes in the organization's software development approach will be needed and significant up-front costs for training and tools will be required. Further, uncertainties in legal policies, such as liability and intellectual property rights that currently hinder software reuse, need to be addressed, and acquisition policies need to be modified to better promote reuse."

The introduction of a software reuse process into a company will require changes to be made in the attitudes and working practices currently in place. One of the main steps to achieving successful reuse is gaining full support of management and staff for the reuse process. Biggerstaff and Perlis [Bigg89b] noted that one of the key similarities in all of the companies with a successful reuse programme covered in their book was that all had the backing and active support of top-level management. Fairley et al. [Fair89] noticed a similar trend in their study of six successful software reuse projects. Hooper and Chester [Hoop91] stress that "Top-level management must take positive action to make software reuse a reality. This means much more than just issuing an edict that software reuse will occur. It means committing the resources necessary to bring about a different way of approaching software development and maintenance - including a different process, tools, a well-trained staff, and an adequate initial library of reusable components."

Software reuse does not come for free. Considerable resources must be made available to a reuse programme in order for it to succeed. This includes not only real capital resources, but also people, time, effort and commitment. Biggerstaff [Trac88b] says: "Software Reuse is like a savings account, before you can collect any interest, you have to make a deposit, and the more you put in, the greater the dividend."

Wasserman [Wass91] recognises some other factors that inhibit the full scale introduction of a reuse programme are the "not invented here" syndrome, an absence of incentives for reuse, and

limited investment in reusability. The not invented here syndrome describes software developer's wariness of using code that they have not written themselves, often caused by a lack of trust in the software. Bott and Ratcliffe [Bott92] describe it in this way: "Technical staff are reluctant to believe that software from another source will be as efficient, effective or reliable as the software they could write themselves; this feeling is often reinforced by bad experiences with imported software. It is easy, however, to overestimate the magnitude of this problem."

Baker and Deeds [Bake89] stress that governments should not get too involved in trying to ensure that the reuse of software is practised, such as providing approved reusable libraries. They further state: "Government should not tell corporations how to reuse software or make them use governmental libraries. If reuse makes sense, they will do it."

Cavaliere [Cava83], based on the experiences of the Hartford Insurance Group's reuse programme, makes the following recommendations:

- Utilise tendencies among staff members to develop code-generation tools oriented to the organisation's needs.
- Develop and maintain an automated index of all programs released into production.
- Be prepared to make full-time staff resources available for the start-up phase and for ongoing support of a reusability programme.
- Provide resources to measure productivity effects of reuse compared against a baseline; this is important to assess the value of reuse and to justify the necessary resource commitment.
- Seek mechanisms for sharing reuse experiences and ideas.

Prieto-Diaz [Prie91] suggests a model for implementing a software reuse plan, which is divided into four stages:

Stage 1: Initiation - reusable components are identified, stored, indexed and made available.

Stage 2: Expansion - As more of the existing software is identified as being reusable, and as more reusable software is developed, the component repository is expanded. A more comprehensive classification scheme is introduced for the repository.

Stage 3: Contraction - Redundant and ineffective components are identified and retired from the repository. The collection of components is streamlined, so that only the most useful remain. This prevents the repository from becoming unmanageably large.

Stage 4: Steady State - As domain knowledge increases, existing components are gradually replaced by those more suited to the specific domain, if required. Components that are designed for reuse should begin to emerge.

Meyer [Meye87] believes that overemphasis on management issues is premature. "It's like expecting better hospital management to solve the public hygiene problem 10 years before Pasteur came along! Give your poor, your huddled projects a decent technical environment in the first place. Then worry about whether you are managing them properly."

2.8 Conclusions

This chapter has given an overview of the field of software reuse, starting with definitions of reuse and reusability. The benefits which successful reuse can bring were identified, including results which have actually been seen in practice. This was followed by a description of the major issues in reuse - technological and organisational. The technological issues in reuse were described along with some suggested solutions to the problems raised. These included technical factors which much be addressed to make reuse possible and different technologies which are available for reuse. The organisational issues were also considered, including some of the psychological, sociological and economic issues that affect the success of a reuse programme.

It can be seen that there are many challenges facing a company wishing to benefit from reuse. Solving either the technological or the organisational problems discussed earlier will bring benefits to the company. However, it is only when both issues are addressed sensibly that the advantages which can be gained from the successful introduction of a reuse programme can be capitalised on.

Chapter 3: Introducing the reuse process and other techniques to support software reuse in a small company

3.1 Introduction

This chapter looks at some of the techniques which can support the introduction of reuse into a small company. The following section defines what is considered as a small company, and looks at some of the characteristics of small companies. The third section considers the effect that the introduction of any type of new technology can have on a company, and looks at ways in which this can be improved. It discusses how to change the way in which an organisation works, then looks specifically at software process improvement. Software process improvement considers the methods and techniques which should be used when changing the method of software development in a company. The fourth section considers risk analysis. This is closely tied to the previous section, as there will obviously be risks involved when introducing any new working practices into a company.

The fifth section considers techniques which will support the introduction of reuse into a small company. These include object-oriented methods, software documentation and reverse engineering. The first part of this section looks at object-oriented design. Object orientation has become increasingly popular over the past 10 years, with the development and inclusion of reusable components being frequently quoted as one of the benefits of using an object-oriented design method. This part of the section starts with definitions of object-oriented principles, then gives an analysis of the advantages and disadvantages of using an object-oriented design method, particularly in regard to how it would support a reuse programme. This is followed by a discussion of the relationship between object orientation and reuse. A more detailed survey of numerous object-oriented methods [Bigg95] was made available to Public Access Terminals Ltd. as part of the case study conducted in this research.

It has been seen in section 2.5 that in order for developers to be able to reuse a component, they must be able to understand the component and recognise it as being appropriate for their current system. Software documentation and reverse engineering can help in the understanding process. The section on software documentation will concentrate on ways to support the creation of documentation, as it has been seen (particularly in the case study associated with this research) that small companies who are low on the process maturity scale tend not to keep good software documentation.

Another technology discussed is reverse engineering. It was noted in the definition of software reuse in section 2.2 that both components which have been designed for reuse and components which have been salvaged from previous developments are appropriate candidates for reuse. Reverse engineering, the abstraction of higher level information from program source code, can support the process of salvaging reusable components from previous developments. The information abstracted from reusable components can also help developers to understand how to reuse the components, an important part of the reuse process (see section 2.5).

3.2 Small companies

It is not easy to define what is considered as a small company. Burns and Dewhurst [Burn86] state that "just what constitutes a 'small business' is open to debate and, even within the UK, differences in the quantitative definitions used by different government statistic-gathering agencies make comparison and conclusions difficult." Indeed, different countries have different formal definitions for the term 'small company'. Andersson [Ande87] notes that number of employees is the most commonly used statistic to define a small company "although, certain countries such as the U.K. have a more elaborate definition, taking into consideration aspects like branch of activity and turnover". He states that: "In the U.K., the range is 1-200 employees".

In trying to define what constitutes a small company, Burns and Dewhurst go on to quote the 1981 Companies Act, which adopted three separate criteria to define small firms:

"*Small companies*: One which for the financial year and the one immediately preceding it, two (at least) of these criteria apply:

a) Turnover does not exceed £1.4m.

b) Balance sheet total assets does not exceed £0.7m.

c) Average weekly number of employees does not exceed 50."

In another publication, Burns and Dewhurst [Burn96a] quote the EC commission's "conditions to be met by a small firm wishing to qualify for state aid:

- a turnover not exceeding ECU 20 million (say £16 million),
- a net capital not exceeding ECU 10 million (say £8 million),
- a number of employees not exceeding 250."

However, it is not just these statistics that define a small company. There is also the organisation structure and culture of the company which sets it apart. Burns and Dewhurst [Burn96a] quote the Bolton Report from 1971 which "described a small business as follows:

- In economic terms, a small firm is one that has relatively *small share of its market.*
- It is managed by its owners or part owners in a *personalised* way, and not through the medium of a formalised management structure.
- It is independent in the sense that it does not form part of larger enterprise and that the owner/managers should be *free from outside control* in taking their principal decisions."

They later suggest that "Personalised management is, perhaps, the most characteristic factor of all. It implies that the owner actively participates in all aspects of the management of the business, and in all major decision-making processes."

Chisnall [Chis87] also lists some of what he feels are the typical characteristics which set a small company aside from other types of business organisation. His list includes:

Inadequate funding – Many entrepreneurs try to run their businesses on shoestring budgets.

Flexibility – Small businesses have a particular characteristic which gives them a strong competitive edge: they are owner-managed, and decisions can be taken quickly.

Specialization – Much of the success of small businesses lies in the fact that they develop products and services with high value-added content: in other words, they offer their customers quality goods which are directly related to their needs.

Technical experience – Small technical firms are often founded by enthusiastic experts with many years of technological experience behind them. However, they often lack good marketing know-how.

Pratten [Prat91] studied numerous small firms, and notes that "Throughout the interviews with managers the flexibility and responsiveness of small firms compared to large firms was emphasised".

This research rates the size of the company based on both its size and company culture, the four key criteria being:

1. Owner managed [Burn96a]
2. Up to 200 employees [Ande87]
3. Specialisation in products and services [Chis87]
4. Flexibility [Prat91]

Public Access Terminals Ltd., the company associated with this research, fulfils all the appropriate criteria for a small company.

3.3 Introducing new technology and software process improvement

The principles of introducing a new technology into a company, or even new working practices, are very similar no matter what the technology or working practices are. The field of 'introducing new working practices into a company' is referred to as organisational development. The goals of organisational development (OD) are usually consistent, no matter what the company does, or how it goes about achieving it. The aim of OD is to improve the working conditions and practices of the company on the assumption that these improvements will also bring an improvement in productivity and profitability, and a happier workforce. OD often calls for process improvement, where a process is a defined set of working practices. Recommendations are made for improvements to working practices in order to meet the overall aims of the organisation.

This section will first look at some of the general principles involved in organisational development and will then consider some of the best methods which can be employed to ensure that the changes are successful and the goals of those changes are met. The section will also consider software process improvement – namely how the principles of organisational development and process improvement are applied to the development of software systems.

3.3.1 Organisational development

Organisational development, already defined as the process of introducing new working practices into a company, is closely linked to process improvement, which is changing the processes which a company uses in order to improve their working practices. Both can be approached in many ways, using many different techniques. However, they have the same overall goals and share techniques in order to reach their goals.

Albrecht [Albr83] suggests that there are four main phases to successful organisational development. "The four steps are really nothing more than the simplest logical progression in

problem-solving: figure out what the problem is, decide what you have to do to change things, put the "fix" into effect, and then compare what happens with what you wanted to happen." More formally, he calls these the Assessment Phase, the Problem-Solving Phase, the Implementation Phase and the Evaluation Phase.

Albrecht recommends that these phases can be carried out by either a staff specialist, an external consultant or an OD task force within the company. However, he feels that OD should never follow a rigid structure in the same way as other company processes. Indeed, he points out: "experience seems to show that the ad-hoc quality of OD is one of its key benefits."

Assessing the current situation and working practices of an organisation must always be the first step before even considering any suggestions on how to improve the situation. There are many ways in which this assessment can be carried out. Perhaps the best way is to watch the company at work and document the results. However, this is a very time consuming exercise which must be done for every function within the whole company. It is easier to use information that is already available within the company. Although company processes may be standardised and documented, with a strict set of company guidelines, talking with the staff involved in the company is the best way to get a true feeling of the company's current working practices.

Burns and Stalker [Burn61] outline their method of assessment. "Our usual procedure, after the first interview with the head of a firm, was to conduct a series of interviews with as large a number of persons as possible in managerial and supervisory positions."

Albrecht agrees that questionnaires and interviewing are useful ways to gather data about the organisation and its processes. He suggests that "the interviewer does best when he or she asks open-ended questions, listens for key themes and concerns, and continues to develop the flow of information without "shopping" for certain kinds of answers, and without steering the people being interviewed too forcefully."

Once the information has been gathered, the problem solving phase begins. Albrecht says that "The result of an effective OD problem solving phase is a realistic, workable, and promising plan of action for the implementation phase." This plan is built by considering the alternative approaches for development of the organisation. These ideas are then assessed and the more promising approaches are formulated into a plan of action. Albrecht suggests that developing a "realistic, stepwise plan for implementing the changes" is one of the key steps to success in this process.

Implementation of the plan can be done in many ways. However, Babcock et al. [Babc90] suggest that "a product or technology that has evolved through a process of incremental improvement has an increased chance of enjoying successful transfer and widespread diffusion." This idea of incremental improvement will be considered in more detail later in the thesis.

Albrecht warns of the "valley of despair", a term he uses to suggest that when implementing the plan, the situation always tends to get worse before it gets better. This is caused by the disruption to the company incident to the changes being put into place. Although people seem to dislike change, Albrecht claims that "people don't like change when they don't think the change will be good for them." Ensuring that employees understand the improvements that will come from their new working practices will encourage them during the difficult transitional period.

In the evaluation phase, the results of the OD programme are measured and evaluated to get valuable feedback about how well the programme is performing and what improvements have been made. Albrecht says that "the primary purpose of the evaluation phase is to discover what course corrections we need to make."

This is also a good time to encourage staff using positive feedback to inform them of the progress being made. Albrecht says that this has several advantages. "First, it focuses the attention on what is working, not on what isn't working. This tends to have a positive influence on overall morale and sense of optimism. Second, it tends to create a sense of expectation and

confidence that things are going to get better. This almost invariably contributes in subtle ways to the commitment people feel toward the organization, and things do tend to get better as a result. And third, the fact that management is giving positive feedback to the people in the organization tends to enhance the sense of "connectedness" people feel towards the executives."

Another method for organisational development is described by Burnes [Burn96b]. He gives an overview of a method by Bullock and Batten [Bull85], who also developed a four-phase model of planned change. "The four change phases, and their attendant change processes, identified by Bullock and Batten are as follows:

1 *Exploration phase.* In this state an organisation has to explore and decide whether it wants to make specific changes in its operations and, if so, commit resources to planning the changes. The change processes involved in this phase are: becoming aware of the need for change; searching for outside assistance (a consultant/facilitator) to assist with planning and implementing the changes; and establishing a contract with the consultant which defines each party's responsibilities.

2 *Planning phase.* Once the consultant and the organisation have established a contract, then the next state, which involves understanding the organisation's problem or concern, begins. The change processes involved in this are: collecting information in order to establish a correct diagnosis of the problem; establishing change goals and designing the appropriate actions to achieve these goals; and getting key decision-makers to approve and support the proposed changes.

3 *Action phase.* In this state, an organisation implements the changes derived from the planning. The change processes involved are designed to move an organisation from its current state to a desired future state, and include: establishing appropriate arrangements to manage the change process and gaining support for the actions to be taken; and evaluating the implementation activities and feeding back the results so that any necessary adjustments or refinements can be made.

4 *Integration phase.* This state commences once the changes have been successfully implemented. It is concerned with consolidating and stabilising the changes so that they become part of an organisation's normal, everyday operation and do not require special arrangements or encouragement to maintain them. The change processes involved are: reinforcing new behaviours through feedback and reward systems and gradually decreasing reliance on the consultant; diffusing the successful aspects of the change process throughout the organisation; and training managers and employees to monitor the changes constantly and seek to improve on them."

3.3.2 Process maturity and process improvement

Software process improvement follows the same principles as organisational development, but is more specific to the processes involved in software development.

Sommerville [Somm96] describes process improvement as "understanding existing processes and changing these processes to improve product quality and/or reduce costs and development time." He goes on to suggest that there are a number of key stages in the process improvement process, namely:

1. Process analysis
2. Improvement identification
3. Process change introduction
4. Process change training
5. Change tuning

The key work in the field of software process improvement is that performed by the Software Engineering Institute (SEI) at Carnegie-Mellon University [Hump89]. The result of their work was the Capability Maturity Model (CMM), which attempts to assess the level of a company's capability based on the processes that they use. The SEI model defines 5 levels of capability:

1) Initial level – No effective management or project plans. Although the company many successfully develop software, this is due to the 'heroic efforts' of the employees, and there is no guarantee that software quality can be produced consistently.
2) Repeatable level – The company has formal management, quality assurance and configuration control methods in place. Therefore, they can repeat projects at the same level of quality. However, there is no formal definition of the processes used.
3) Defined level – The company has defined their processes, and so has a basis for process improvement. The processes have formal procedures to support their use throughout the company's development lifecycle.
4) Managed level – Again, the company has formal processes, but they also have a programme for measuring the quality of the processes being used and the products being developed.
5) Optimising level – Metrics taken from process management are fed back into the company's process improvement programme to ensure that managed processes are improved to increase the company's overall performance.

The CMM has been the basis for considerable further work in software process improvement, for example, the ESPRIT BOOTSTRAP project [Koch93]. Other work in the same field, including SPICE, TickIT and STARTS, is summarised by Thompson and Mayhew [Thom97]. Similar work has also been done in the field of reuse and several different reuse maturity models have been suggested [Trac95], which have been incorporated into McClure's Reuse Readiness Assessment [McCl97].

Although a great deal of work has gone into the CMM and it has been hailed as a step forward in the field of software process improvement, there are still some doubts about its validity. In their evaluation of the CMM, Bollinger and McGowan [Boll91] go as far as to say that "the current grading system is so seriously and fundamentally flawed that it should be abandoned rather than modified."

Considerable time and resources are consumed when simply measuring a company's capability maturity, as a great deal of work must be done to investigate and measure the standard of the company's processes. There is also considerable paperwork involved in investigating, defining, documenting and implementing process improvement on the scale suggested by the SEI using the CMM and other maturity models. These factors contribute to the fact that small companies find the concept of quality assessment and process improvement prohibitive. There is also a fear that process definition and improvement will cause them to lose the flexibility that keeps them competitive within the marketplace.

Humphrey [Hump93] suggests that "people need to be convinced of the effectiveness of new methods before they will change." It has already been stressed that this research will consider those companies which rate at the bottom level of any maturity model. This suggests that methods for successful process introduction and technology transfer methods are more interesting than measuring the company's current capability. Companies at these low levels will only be interested in improving their capability when the see the advantages of doing so, and it is hoped that the benefits of a reuse programme presented in the right way will encourage them to improve their working practices.

3.3.3 Process assessment

The term process assessment describes a variety of different ideas and techniques which can be used to investigate and analyse the way in which work is done. The recent research performed in the field of software process improvement has been based on earlier tried and tested methods in the business areas of organisational development and work study. This section describes this earlier work in an attempt to understand how companies can be studied in order to assess their current processes and identify areas for improvement.

Process assessment can be performed using what is known in the business world as a work study. Buckley [Buck85] describes work study as "a term which covers a number of techniques designed to improve the efficiency of the organisation and help in the control of costs."

There are many different techniques which can be used as part of a work study. One of them is method study. Radford [Radf84] describes method study as "that part of work study that provides a systematic approach to improving the way in which work is done".

Radford suggests that "the procedure of method study has been formalised into six steps as listed below."

(1) Select work to be studied.

(2) Record existing method of working

(3) Examine critically the existing method.

(4) Develop an improved method.

(5) Install the improved method.

(6) Maintain the improved method."

Buckley [Buck85] also confirms the usefulness of using method study as part of work study. He says: "Method study is concerned with how the work is carried out. It looks at existing procedures with a view to improving them. In essence it asks the question, 'Is there a better way of doing this job?'"

He goes on to describe method study in more detail, confirming the steps suggested by Radford. "The procedure has six stages namely: select; record; examine; develop; install; maintain." Buckley goes on to describe these steps in more detail:

1. Select the job to be studied.

This selection should normally come from management. "Once a job has been selected and authority has been obtained for its investigation the most important task before moving on to the next stage is to inform all those who will be affected by the study. Explaining the reasons for the study prior to its commencement will prevent misunderstanding and increase the likelihood of worker co-operation."

2. *Record the present method.*

   "A detailed analysis of present methods is necessary before we can move on to seeing what improvements are possible or desirable."

3. *Examine the existing methods.*

4. *Develop the new improved method.*

   "The existing method which we have now investigated forms the basis for our search for new improved methods...during this stage we carefully question all the we have recorded... Eventually, out of the critical examination will come the ideas for the improved method. These will be discussed with the management in the department concerned...At this stage it is also necessary to draw up a formal report which will outline:

   - the changes recommended;
   - the cost of those changes;
   - the savings which will result;
   - the time needed to institute the changes."

5. *Installation of the improved method.*

   "Work study personnel must pay particular attention to two aspects of installation. First they must persuade everyone concerned of the need for change. A successful installation needs the co-operation of all staff. Secondly the installation will involve considerable planning."

6. *Maintain the new method.*

   "The introduction of the new method will not be without its difficulties, but it would be wrong for work study personnel to consider changes immediately. It will take some time before all employers are fully conversant with the new method and reach the expected level of productivity."

The work in the field of method study presented here forms the basis of the development of the method used in this research, as defined in chapter 5.

3.4 Risk Analysis

Every new endeavour contains an element of risk. There will always be uncertainty as to whether the endeavour will be successful. Risk is a measure of this uncertainty, and analysis of the risks involved should be considered before and during any project. This section will consider what risk is, how it can be analysed, and how the analysis can help risk managers to decide whether to continue with the project or discard it.

Raftery [Raft94] defines risk along the following lines: "Risk and uncertainty characterize situations where the **actual** outcome for a particular event or activity is likely to deviate from the estimate or forecast value. Risk can travel in two directions: the outcome may be better or worse than originally expected. These are known as **upside** and **downside** risks."

He goes on to state that "some people like to distinguish between **risk** and **uncertainty**. The distinction is usually that risk is taken to have quantifiable attributes, whereas uncertainty does not."

Sommerville [Somm96] suggests that "risks are a consequence of inadequate information. They are resolved by initiating some actions which discover information that reduces uncertainty." However, this simplistic view is not always practicable, as gathering the information required to reduce the risk may be more costly than the consequences of failure in the proposed undertaking.

This is particularly important in the case of a reuse in a small company. This research considers the introduction of reuse into the software development methods of a small company. The risks associated with this undertaking are associated with the time, effort and resources which must be

committed to the reuse programme. Indeed, in the worse case, the downside risk is that the resources could be wasted and the time and effort expended on reuse simply end up delaying production of the company's software. However, on the other hand, the upside risks are that the considerable benefits of reuse described in section 2.4 could be made available to the company. This would increase their productivity and reduce their development and maintenance costs. The third option is to keep their current development methods. There are also risks associated with this. Public Access Terminals Ltd., the company associated with this research, have already discovered that in today's fast moving market, a company which stagnates and does not improve soon falls behind its competitors and fails anyway.

The importance of risk analysis in this research is that the method presented for introduction of a reuse programme into a small company attempts at all stages to minimise the risk associated with the changes that are required for reuse. They also attempt to ensure that some benefits from reuse are reaped on a smaller time scale than is the case with large corporate reuse programmes. This enables the company to try reuse techniques, and, if they don't work, to scrap them and try other new techniques or revert to their previous development methods.

Another problem with attempting to reduce uncertainty by gaining further information is that there are no studies considering the introduction of reuse in a small company to gain further information from. This means that the amount of study involved in analysing and reducing the risks associated with this endeavour would be more expensive for the company than simply trying the techniques in practice.

3.5 Techniques to support the introduction of reuse in a small company

This section covers some of the techniques which will help in the introduction of reuse to a small company. One of the key points that has been seen in a small company such as Public Access Terminals is that they often do not employ a formalised design method. This means that each developer has a different way of designing and building software. This can create problems

when trying to integrate software written by different developers, because the designs may not be compatible. The first of the techniques considered below is the use of an object-oriented design method. Object-orientation has been chosen for two reasons. The first is that its proponents claim that object-orientation supports reuse. Secondly, it has been seen within Public Access Terminals (and throughout other software development companies) that there is a move to develop using object-oriented languages such as C++. It makes sense to have a design method which supports the technology being used. The section will look at what object-oriented principles are, and how they support reuse.

Another technique which it is felt will aid small companies is automatic generation of software documentation. In their efforts to produce software for their customers, documentation is always the last priority for developers. In small companies, this is especially true, as it is often felt that writing documentation is a waste of valuable development time which could be used more productively. Techniques and tools to support the generation of documentation would be of great value to the company, and also to their reuse programme.

In the same way, reverse engineering, which is the abstraction of higher level information from source code, is another technique to support reuse. Reverse engineering could be used to gain more information about software which has been produced within the company. This information could aid developers when attempting to reuse that software in a new development.

3.5.1 Object-Oriented Methods

Definitions of Object-Oriented Principles

The principles of object-oriented design have been derived from earlier work on information hiding [Parn72], abstract data types [Lisk74] and, most significantly, work on object-oriented programming languages such as Smalltalk [Gold83] and Simula-67.

Ghezzi et al. [Ghez91] summarise the current state of affairs in object-oriented design admirably. They say: "Unfortunately, the terminology of object-oriented methods is not well standardised, and there is not even agreement as to what object-oriented design really is."

However, many of these views are rather too general to have any empirical evidence to support them. It seems that too often the benefits of the use of object orientation are assumed simply because they sound right, rather than because there is evidence to support the claims made.

Sommerville [Somm89] gives a valuable word of warning: "It is unwise to be dogmatic about the design process and always to adopt an object-oriented approach irrespective of the system being developed. An object-oriented view of system design is not always the most natural." In the next edition of his book [Somm96], he clarifies this further: "No one method is demonstrably better or worse than other methods; the success or otherwise of methods often depends on their suitability for an application domain."

Object orientation contains concepts that allow the real world to be modelled very effectively. The principles of encapsulation and inheritance also make it far more supportive of reuse than many other software design methods. However, object orientation is not the 'be all and end all' of software development techniques. It certainly has its limitations, and is not as effective in modelling some application areas. It is important to recognise this, and only to use object-oriented techniques where they will achieve the best results. Although object-oriented programming languages exist, object-oriented design techniques can be applied to most modern programming languages.

Object-Oriented Design and Reuse

Object-oriented methods have been promoted as inherently supporting reuse. Halladay and Wiebel [Hall93] state that "The most commonly touted benefit of OOP is reuse." Many authors have extolled the advantages of reusability in object orientation. Atkins and Brown [Atki91a] emphasise that reuse is one of the advantages that arises from an object-oriented approach,

specifically from direct support for abstraction. They suggest that the reuse of classes in a hierarchy and object libraries are specific examples of reuse that stem from object-oriented practices.

Ince [Ince91] says that "Polymorphism allows a developer to build up a library of reusable objects, and contributes greatly towards the ability to develop reusable software." Wiener and Pinson [Wien88] consider that one of the main goals of object-oriented software development is "to shorten the time and lower the cost of development by using reusable software components in the form of baseline classes and by employing incremental problem solving using subclasses." Tsichritzis and Nierstrasz [Tsic89] seem to believe that, due to the heavy emphasis on reuse in object-oriented programming, "we can expect extremely large collections of reusable objects to be available to us." They feel that the problems of the future will be associated with managing such large collections of objects. They follow this with a suggestion that expert systems will be the appropriate tools for helping programmers to find their way through databases of reusable object classes.

It has been seen, however, that among the object-oriented design methods available, there is a lack of explicit provision for reuse [Goss90]. Udell [Udel94] also expresses this opinion: "The traditional OOP vision was, at best, vague on the subject of reuse: Objects would appear as by-products of software development, a market would emerge, and programmers would become producers and consumers of objects." This unstructured, and rather naive, view of reuse can be seen in many object-oriented texts.

Meyer [Meye88] offers considerably more advice on the construction of reusable classes in his text. He suggests that: "A good object-oriented environment will offer a number of predefined classes implementing important abstractions. Designers will naturally look into these to see if there is anything they can use...New applications, if properly done, will also produce more specialized reusable classes. As object-oriented techniques spread, the number and abstraction level of available components grow."

Tello [Tell91] also questions the provision for reuse in object-oriented methods. He states: "some say that the key advantage of OOP is the ability to reuse code for many different programs, but, by itself, this is not significantly different from library functions." Mullin [Mull89] would agree: "As most books available today on OOP say, one of the major benefits of objects is that they are reusable. So are C functions. The difference is that objects, representing both data structures and operations that can be performed on these structures, represent functional packages, requiring no additional work on the part of the programmer to use them. The packages are always uniform and they interact identically with other objects, regardless of the purpose of a given object."

Raj and Levy [Raj89] note that one of the problems with inheritance in object-oriented systems is that "classes are not automatically reusable". They suggest that for successful reuse, inheritance requires the use of a set of coding rules and a set of design rules. Johnson and Foote [John88] would agree with this second point, presenting a set of 13 rules for designing reusable classes.

Udell [Udel94] suggests that "object technology failed to deliver on the promise of reuse", but that *componentware,* in which components are *encapsulated,* or combined into a single, separate unit with a well defined interface, in order to make them reusable, is the way forward for reusability.

Cox [Cox86] says: "Object-oriented programming can help to put reusability at the fore-front of a programmer's work. But it can't do it alone unless an information network is provided to help consumers discover useful code quickly and to understand how it applies to their needs."

Winblad et al. [Winb90] note that: "Software reuse does not occur by accident, however - even with object-oriented programming languages. System designers must keep the advantages of reusability in mind, planning ahead to reuse what already exists and designing reusability into

the new components they create. This requires that programmers adopt new programming behavior, values, and ethics. Borrowing classes created by others must be considered more desirable than implementing a new class. Reviewing existing code to identify opportunities for reuse must have priority over writing new code. Finally, programmers must create simple, reusable classes rather than complex, inscrutable classes. Simplicity is a major tenet of the general philosophy of object orientation."

Jones [Jone92] considers that "object orientation may make a marginal difference in implementing reuse, but any major reuse program is largely a matter of will, not of technicalities."

It is important to note that no one method, technology or technique will solve all the problems associated with reuse. There will always be complications, and these must be expected and planned for. Burd and McDermid [Burd92] note that: "Risks are involved in all software developments, however, often those projects which employ reuse are susceptible to greater risks than those which do not." However, with the potential support for reuse provided by the use of object-oriented techniques, these risks, and the difficulties involved in successful reuse, can be reduced. This view is confirmed by Burd [Burd93b]: "Object-oriented design displays the most promise as a re-use methodology...Object orientation on its own isn't sufficient to solve all the problems associated with re-use. This can be achieved only by providing well-defined support that enables re-use to be integrated into a suitable lifecycle model."

Object orientation is far from being a panacea. Even when associated with reuse, it does not solve the problems typically associated with the software crisis. Hatton [Hatt95], in his study of defect rates using various programming languages and strategies, found that the defect densities recorded in object-oriented C++ systems were slightly worse than a comparable system written in conventional C code (2.4 defects per KLOC in C compared with 2.9 defects per KLOC in OO C++). He notes that the defects were also more difficult to find in the OO system. He goes on to say that "unless object-orientated techniques lead to very considerable re-use, they are unlikely

to improve system reliability significantly. They also seem to require much more specialist maintenance attention and are harder to debug in current implementations."

There must be a significant amount of reuse achieved in an object-oriented system for the benefits of the adoption of object-oriented principles to be seen. Melo et al. [Melo95], in a study conducted in the University of Maryland, "provided significant results showing the strong impact of reuse on product productivity and, more particularly, on product quality in the context of object-oriented management information systems." It is interesting to note from their results that it was only when reuse rates of at least 40% were achieved that significant improvements were made in development productivity and the amount of rework required to debug the systems after testing.

A study of several different object-oriented design methods [Bigg95] was written for Public Access Terminals to help them to determine which method would be of the most use to them. The study included details of each of the chosen methods and an worked example using the method, along with an analysis of each method.

3.5.2 Software documentation

It has been readily accepted throughout the software engineering community that documentation is a valuable aid to understanding software. However, useful documentation is not always kept. This is particularly true in small companies without structured processes, where the effort required to produce useful software documentation is often seen as far less productive than other work that could be done by the developers. The development teams are often small, and feel that there is sufficient experience and communication within the team to gain all the relevant information about the software without the need for documentation. This section looks at support which can be provided for software documentation in these situations.

Literate Programming

Literate programming is a phrase first used by Donald Knuth [Knut84]. He uses it to describe his system of software documentation called WEB. The essence of literate programming is that the source code and documentation of a program are tied together in one file. This is done by structuring the source code and comments using TEX commands as instructed by Knuth. The result will be a file that can be machine processed by the WEB system in two ways:

1) TANGLE - this separates the source code from the WEB file in order to produce a file that can be compiled.
2) WEAVE - this produces the 'pretty printed' version of the source code for the program. Features of the pretty printing are that:

   - keywords for the language are emboldened.
   - comments are interspersed through the code to annotate the source code listing.
   - an cross-referenced index of all the sections and variables used in the program is produced.

The WEB system, to date, works with the languages Pascal, C and C++.

The basis of literate programming is to provide an aid to program comprehension. The pretty-printed version of the source code is far more readable, and, with the correct use of comments, far more understandable than wading through standard source code. However, it requires a lot more effort and skill to create a piece of WEB code than to produce a standard piece of commented source code.

For its time, the literate programming principle was valid, but it is now beginning to become dated. This is especially marked in light of the new style of programming environments that are becoming available. Some of the keys to literate programming are the highlighting of keywords and comments and the indexing of variable names. Many new programming environments now

do this automatically. An example is the Microsoft® Visual Workbench for Visual C++ [Micr93a]: "Visual Workbench highlights language keywords, identifiers, comments, and strings in different colors. This feature is useful when learning a language or viewing lengthy and complex source files."

Childs and Sametinger [Chil96] describe a system for software documentation using object-oriented principles on literate programs. This eases the process of reusing documentation. However, as it uses the principles of literate programming described above, it also suffers from the same drawbacks.

Documentation Tools

It has been estimated that software engineering organisations can spend as much as 20-30% of all their software development effort on documentation [Pres92]. The documentation process itself can also be quite inefficient. These factors often lead to poor standards of initial documentation, or poor maintenance on initially good documentation. Both lead to the same problem, which is that software documentation is useless to both maintainers and developers attempting to maintain or reuse software components (because either the documentation does not exist or is out of date).

Documentation tools can help to alleviate these problems by automating support for documentation generation. Some CASE tools can automatically generate software documentation based on the information contained in internal repositories that have been generated during the lifetime of the project. Others support developers and maintainers in writing their own documentation by providing templates in which to place the appropriate information.

Capers-Jones [Cape94] feels that things will change for the better with new technology. "The percentage of human beings who can write clearly is not very high. Therefore software user documentation is likely to remain marginal, except for software produced by large companies

with full technical writing, editing, and illustration departments. The emergence of multi-media technologies and graphical user interfaces are likely to change the nature and appearance of user documentation in fundamental ways."

3.5.3 Reverse Engineering

Reverse engineering is the process of abstracting information from software source or object code. Sommerville [Somm96] describes it as a "process of analysing software with the objective of recovering its design and specification. The software source code will usually be available as the input to the reverse engineering process. Sometimes, however, even this has been lost and the reverse engineering must start with the executable code."

Bennett [Benn93] stresses that "Reverse engineering is seen as an activity which does not change the subject system, nor does it create a new system based on the reverse engineered subject system. It is seen as a process of examination and understanding (and of recording the results of that examination and understanding), not a process of change or replication."

Chikofsky and Cross [Chik90] define reverse engineering as "the process of analyzing a subject system to identify the system's components and their inter-relationships, and to create representations of the system in another form or at higher levels of abstraction."

For the purposes of this research, reverse engineering is defined as any technique which abstracts useful higher level information from a software system without modifying that system.

Over the past 10 years, there have been so many different methods, techniques and tools developed for reverse engineering that they cannot all be considered in this chapter. The next section concentrates instead on the relationship between reverse engineering and software reuse.

Reverse Engineering for Reuse

Frazer [Fraz92] states that the primary purposes of reverse engineering are "to provide an aid for comprehension and a basis for maintenance or future redevelopment". He goes on to suggest that one of the objectives of reverse engineering is to facilitate reuse. He states that "a major inhibiting factor in the rate of growth of the number of users embracing reverse engineering is the lack of integration of current tools and techniques."

Several authors have recognised the importance of reverse engineering as a technology to support reuse (for example: work done in Logica [Walt92] and at the Centre for Software Maintenance, Durham University [Munr92]) and some have suggested methods for extracting reusable components from software systems [Ning93], [Cimi95], [Neig96]. The latter tend to concentrate on *program slicing*, the extraction of functionally related code fragments from a software system.

In this research, reverse engineering is used to provide information for both developers and maintainers about reusable components. It is, therefore, also related to the fields of program comprehension and software documentation. The information generated by reverse engineering reusable components can be used to help software engineers to understand the purpose of a software component. This understanding helps the developer to reuse the component. Reverse engineering is integrated with the other areas of software engineering considered to provide an integrated reuse support environment.

3.6 Conclusions

There are a lot of techniques which can be applied within the field of organisational development and process improvement. Those described in this chapter will be considered in greater detail in the next two chapters, where a method for the introduction of reuse into a small company will be presented, which is based on the work outlined in section 3.3.

There is also a lot of literature available on the subjects of reuse and object orientation. One of the reasons for this is that their influence stretches to every part of the software lifecycle, from requirements analysis through implementation and testing to maintenance. If reuse is taken in its broadest sense, then anything from any part of the software lifecycle can, and should, be reused. In practice, this is very difficult and, in some situations, uneconomical.

However, much of the literature on reuse tends to look at the subject either on a very large scale (covering every aspect of software production), or on an atomic scale (the reuse of components). It has been seen through the literature that object-oriented design principles are best suited to the principles of reuse, although explicit method support is sparse. It is, therefore, wise to encourage the use of object orientation as a design method to accompany a reuse programme. However, it is not wise to rely on the use of an object-oriented method to bring the benefits of reuse without any extra effort being required. Many different object-oriented design methods are currently available, each with a different emphasis. C++ is currently the most popular of the 'object-oriented' languages. There is a problem with the fact that OO design methods don't explicitly support reuse, although the principles of object-orientation do.

In the automatic generation of documentation, it has been seen that literate programming is a useful concept, but one that has been subsumed by modern programming environments. Further work on generating documentation from comments in the software's source code would be valuable, especially if integrated in a reuse environment with information abstracted from the source code using reverse engineering.

In conclusion, from the survey of literature in Chapters 2 and 3, the gap in the field of research that has been seen is that there is little provision for the setting up of a reuse programme in a small, unstructured company. It is felt that a method for introducing a reuse programme, integrated with an object-oriented design strategy, coupled with automatically generated information about the source code, will help to ameliorate this problem. It will make the

principles of reuse more accessible to such companies, because the investment of time and effort needed to benefit from reuse will be reduced.

As previously discussed in section 3.4, it is important to minimise the risks that a small company will be taking when implementing a software reuse programme. As there is currently no further information available on software reuse in small companies, conducting further investigations into this area will not help to reduce uncertainty. Therefore, in the following chapters, a method will be developed which helps to minimise the risks taken by a small company when implementing a reuse programme.

Chapter 4: Solutions

4.1 Introduction

The goal of this research is the effective realisation of software reuse within a small company. In the case study associated with this research, Public Access Terminals Ltd. were motivated by a desire improve their software practices. There were several reasons for this desire to improve.

The first is that the company recognised the impact of changes in the software market, and realised that they could no longer continue with their current software system. Customer demands meant major changes in both the product and its environment. Advances in technology meant that their product, which had previously been a market leader, was falling behind its competition. Realising that change was inevitable, the company wanted to start again, using better methods to develop better structured software.

Secondly, the company realised the importance that the software market was placing on standards and were interested in International Standards Organisation and British Standards accreditation. This, again, would mean an improvement in their software development methods.

Thirdly, the company had heard some of the benefits which could be gained from the success of software reuse, and were excited to gain these benefits for themselves. These, and other factors, led the company to become a part of a Teaching Company Scheme with Durham University, hoping to utilise the expertise of the university to help with these improvements.

In order to identify an appropriate strategy for reuse introduction in the company, several other successful reuse programmes were studied.

4.2 Study of successful reuse programmes

As discussed in Chapter 2, the realisation of software reuse depends on many factors. However, there have been several successful reuse programmes implemented in software companies, results of which have been made available through reports and papers. Section 1.1 identified some of the key reuse programmes which have been reported, along with the benefits that have been gained from the introduction of reuse into these companies.

It has been shown that software reuse can offer great benefits to companies when used effectively. Some success stories have been quoted, and a few of these will be considered in more detail in this chapter. This will be in an attempt to discover commonalities shown across the companies, and identify whether the successes gained in these companies could be transferred to a small company.

Raytheon

The first of these companies is the Raytheon Missile Systems Division of the Department of Defence. Lanergan and Grasso [Lane84] studied over 5000 production COBOL source programs, and identified common categories for tasks performed in the programs. Three main types of function were identified, and were abstracted into standardised reusable logic structures. Developers could then use these structures when building new programs. When reusing the structures, the developers estimated that they achieved a 50% increase in productivity by averaging 60% reusable code.

Although this is a great way to identify candidate reusable components, such a study would be very difficult to do in a small company. Small software development houses often only have a few different programs which they develop and maintain. In the case study associated with this research, the company has a single product. Studying such a small system for common

components would be difficult, as there is not enough material to notice any general trends across programs.

GTE

The second case study considered is that of GTE. In his paper on the implementation of faceted classification for software reuse within GTE, Prieto-Diaz [Prie90] describes the system used for software component classification. He also describes the searching and retrieval support system, the librarian and organisational support and problems with the technology transfer. In discussing the usage experience for GTE's Asset Management Program (AMP), he calculates the reuse factor gained by dividing lines of code reused by the total lines of code produced by the organisation. He estimates that $1.5 Million was saved with a reuse factor of 14%. Prieto-Diaz stresses that "there must be a strong organisational commitment to reusability and an effective management structure to operate a reusability program...an organizational infrastructure is needed for a reuse system to succeed." He goes on to identify 6 groups which should be set up to support the reuse programme: the management support group, the library system, an identification and qualification group, a maintenance group, a development group, and a reuser support group. He then stresses that the role of the librarian is "critical for a successful reuse program."

These are very valid suggestions when taken in context, but far outside the resources of a small company. It is very possible that a single software developer could constitute five of the six groups suggested, acting as librarian, component identifier and qualifier, developer, maintainer, and support group for the reuser, namely him(or her)self. Such a situation would be absurd, and the extra workload added to the developer would probably cause them to scrap the idea of reuse as 'far too much work', and go back to their preferred development method.

Fuchu Software Factory

The third case study describes the software reusability measure in place at the Fuchu Software Factory, a part of the Toshiba Corporation in Japan. Matsumoto [Mats84] describes how the software processes used at Toshiba have been modified to support the reuse of software components. Components are described at three levels - the requirements level, the design level and the program level. Traceability is maintained through the levels, so that the component is designed and programmed to match the specification. Although there is no discussion of how reusable modules are located, Matsumoto indicates that, if considered as assembler code, approximately 50% of lines of code are reused, which has led to an increase in factory productivity of more than 20% per year.

This style of introducing reuse is very valuable, but relies on the fact that there are processes already in place in the software development environment. When development processes are successful, they can be modified to introduce new practices and improve the software process [Carn95]. However, small companies often have no software processes in place.

Other Examples

Karlsson [Karl95] also quotes AT&T, Hewlett Packard, IBM, NEC, CAP and Ericsson as examples of companies with significant corporate reuse programmes. All of these are large companies with structured processes in place. Another key point made is that any reuse programme will only be successful when it is supported by top-level management. This tends to suggest that this is the type of environment in which reuse can be made successful. However, although these large software development companies are a significant part of the computer industry, there are many smaller software development companies which do not fit into the same mould.

Chapter 3 looked at many different techniques which can be applied when introducing a new way of working to a company. In the rest of this chapter, several alternatives to introducing a software reuse programme into a small company are suggested and considered.

4.3 Introduction of Structured Processes

Based on the success of the reuse programmes considered in the previous section, the most logical approach would be to introduce reuse in the same way. The implementation of a reuse programme would follow the guidelines which have been made in many software publications. A good example of these is the book edited by Karlsson [Karl95], perhaps one of the most complete practical texts on the successful implementation of a software reuse programme. This also follows the process assessment and improvement techniques based around the Capability Maturity Model [Carn95] and the Reuse Maturity Models mentioned in section 3.3.2.

This type of reuse programme implementation would be based on the full introduction of structured processes to the company. In essence, it would mean starting by introducing software development processes within the company, then bringing reuse in as a part of those processes. This would move the company towards the International Standards Organisation's 9000/9001 and British Standards 5750 standardised process recommendations, introducing reuse as a part of those standards. The company's software process would be studied, analysed, documented, implemented and improved by this widespread introduction of standardised processes throughout the company. Reuse would be an integral part of those processes, with the excellent recommendations which have been brought forward in many reuse texts being successfully implemented.

Obviously, this would be the ideal solution. However, it is unlikely to work in practice. Introducing this 'large-company' ideal would take a great deal of time and effort for both the management and the staff of any company. Indeed, for a small company which currently has no standardised practices, such an overhaul of working practice and environment would take a vast

amount of time and resources to implement. This is obviously time and money which is not spent developing software - the lifeblood of the company. It is a recognised fact that the introduction of any new working practice takes a large amount of up-front investment, however beneficial it may be in the long term. Often, a small company cannot afford that kind of investment, whether it be of money or time, because their resources are so much more limited than a large company. This is exactly what puts them off the idea of implementing a reuse programme - the fact that there is a large, up-front investment which may not pay for years to come. They cannot afford that kind of risk. As discussed in section 3.4, a method should be considered that reduces the risk faced by a small company when implementing a reuse programme.

4.4 Incremental Introduction of Reuse

The second suggested solution is that of the incremental introduction of software reuse. This is where reuse is the flagship to which the efforts of the company are directed. However, unlike the previous solution, the major changes required to implement a reuse programme are broken down into smaller steps. This is so that the benefits gained from reuse at each level of improvement can help to 'fund' the forthcoming changes that will be required to move to the next level.

The end is the same as the previous solution, but the means to get there are quite different. Staff motivation can be radically improved by this approach. People seldom like change, particularly when they are comfortable with the environment that they are in. However, if they can see the practical benefits which can come from change, they will be much more motivated to do what is required. The idea of using reuse as the flagship for these changes means that when the software developers do something to improve their software practices, they can actually see the benefits because it constitutes real productivity gains in their software development. A reuse repository is built, and developers can use software from it, which is a tangible benefit that they can see in practice. These perceived benefits from the reuse programme also help motivate the staff to actively participate in the programme.

This seems ideal for a smaller company, as the amount of initial investment which would be required would be minimised, at the same time as maximising the benefits which can be obtained from reuse. Of course, there are disadvantages to this approach. The time scales for improvement are lengthened using this approach. This means that it would take the company longer to improve their capability maturity. It would also mean that the software development processes would be in a constant cycle of change. However, the fluidity of this method would allow a small company the flexibility that they require to develop the type of software that their customers require. Real stability in the company's processes would only come when the company had reached the higher levels of the CMM i.e. achieving a repeatable, defined, managed software process.

Perhaps the biggest benefit of this technique is the reduction in risk associated with the incremental changes in working practices. The changes would be implemented on a smaller scale, and those changes which are detrimental could be discarded before they caused serious damage to the company. On the other hand, successful changes would benefit the company almost immediately while minimising the disruption caused by changing the company's development processes.

4.5 Encouraging ad-hoc Reuse

The third solution is perhaps the most practical from the software developer's point of view. The idea is simple - provide the developers with a practical, usable development environment which supports reuse, then let them get on with it. It is expected that reuse will be achieved as the developers learn more about their environment, and the resources that are available to them. As Meyer [Meye87] succinctly put it "Give your poor, your huddled projects a decent technical environment in the first place. Then worry about whether you are managing them properly."

Using this solution, the developers would be given a good technical environment in which to develop their software. The programming language would allow the developers to build their systems using the principles of structured software engineering which encourage the development of systems as reusable components. Valuable component libraries would be sought to support the development environment, allowing the developers to make use of the greater resources available to them. Standard development and project management tools would also be made available. However, no support would be given to the developers in the reuse process, as management do not have the time or resources to worry about the details of what happens in development. There would be no technical or organisational support for reuse, leaving the developers without guidance or instruction on how to benefit from the introduction of a reuse programme. If reuse makes sense, the developers will surely do it, and gain the benefits which it brings.

The limitations of such a solution have already been discussed in Chapter 2. It was seen that there are many factors which inhibit the introduction of software reuse, not all of which are technical. Tracz [Trac88a] stated that "if one looks at the most-often-stated reasons why software is not reused, the overwhelming majority of them may be classified as psychological, sociological, or economic." A good technical environment cannot solve all the problems associated with the introduction of software reuse. Such factors as the not-invented-here syndrome must be addressed, and reuse should be measured and rewarded if the greater gains that it can bring are to be realised. By encouraging ad-hoc reuse, the developers will certainly gain from the measures suggested above, however, the full benefits of reuse will never be realised without top-level management support.

4.6 Introduction of CASE Tools

The final solution suggested is the introduction of Computer Aided Software Engineering (CASE) tools to support the introduction of reuse as part of the software development process. As with the third solution described in the previous section, the developers would be given a

quality technical environment, with access to reusable libraries. However, with this solution, CASE tools which support both structured software development and development with reuse would also be made available to the software developers.

Many different types of CASE tool have been produced over the past 10 years, and each vendor promises improvements to programmer productivity through the use of their tool. By using CASE tools, some of the more mundane tasks carried out by the developers can be eliminated, leaving them free to concentrate on the more difficult, creative development tasks which a tool cannot do. Software tools have been proven to be effective in other engineering environments (such as CAD programs). By making the right tools available to software developers, their job can be simplified and enhanced, supporting them in the reuse process and allowing them greater opportunities to search for and incorporate reusable components.

This solution is a good one, but alone, it is not sufficient to bring real benefits to a small company. CASE tools can be very effective when used correctly. However, they are just tools, and will only be of use when the correct tool is used with the right training in the right environment. A hammer and chisel in the hands of a baker will be of no practical use; but, in the right hands, these simple tools can produce amazing results.

Another problem with the introduction of any tool is that, without any process to support its use, the tool is unlikely to be used effectively. Excellent CASE tools have been installed in software development companies, but have made no practical contribution to the staff because no-one knows how to use them. Such tools, however effective they are, end up as an expensive waste of resources. A process to support the tool, and training in the use of the tool, are required to make it effective.

4.7 Conclusions

Several different options have been presented in this chapter for the introduction of software reuse into a small company. Each of the options has been discussed, particularly with reference to their validity for a small company. The first option was obviously the 'ideal' solution, but it was seen that the widespread introduction of structured processes (as recommended in the CMM and other maturity models) in a small company would probably be too large scale and resource intensive to be successful. There are very real benefits to this approach, which would be achieved in a smaller time scale than using the second suggested option. However, with the amount of resources which would have to be committed to the programme of process improvement, the risk is far, far greater that the company will collapse before the improvements start to pay off. The second option is more practical and far less risky, introducing reuse incrementally in the company, using the benefits at each level of improvement to 'fund' the next level. Encouraging ad-hoc reuse, the third option, was the most likely to be accepted by the company. However, this would not bring the scale of benefits that can be achieved by a properly organised reuse programme. It was felt that the introduction of CASE tools can be valuable, but, on their own, they are not likely to be used effectively. Of course, there is a fifth option, which is to make no changes, but as already discussed in section 3.4, this option has associated risks of its own.

Based on the options available, the decision was made to follow the second option, attempting an incremental introduction of reuse in the case study with Public Access Terminals. As discussed, this option minimises the risk associated with the introduction of reuse into a small company. A method has been developed to facilitate the incremental introduction of reuse into a company, which is described in the next chapter. It was also felt that the benefits which can be gained from the use of CASE tools would be valuable in automating support for the reuse processes within the company. It was seen in Chapter 2 that it is only when both the technological and organisational issues in reuse are successfully addressed that the benefits of reuse can be capitalised on. In this case study, it was decided that the initial stages of the method

would be implemented first, so that some of the organisational issues could be addressed. Then, when the requirements for technical support for the programme could be clearly identified from the work already carried out, the technological issues could be addressed. In this way, the CASE tool developed would address the real needs identified during the first stages of the incremental reuse programme. The following chapter describes the method developed for introducing reuse to the company, with the steps to be followed and an identification of requirements for the CASE tools.

Chapter 5: Reuse in a Small Company: The method

5.1 Introduction

This chapter looks at the method for introducing software reuse into a small company recommended as part of the thesis. The research has been conducted in conjunction with Public Access Terminals Ltd., a small software development company. The research method adopted is based on Potts' [Pott93] idea of using 'industry-as-laboratory'. Potts suggests that most software engineering research has been following a 'research-then-transfer' methodology, and that this often fails to address significant real-life problems. He introduces the concept of 'industry-as-laboratory', in which he recommends that "researchers identify problems through close involvement with industrial projects, and create and evaluate solutions in an almost indivisible research activity".

In association with a small software company, the thesis explores the possibility of introducing software reuse techniques into a company who are low on the process maturity scale. As such, they rely solely on the 'heroic' efforts of their employees [Curt92] to ensure that their products meet the demands of their customers and are competitive within the marketplace. This chapter describes the method for introducing reuse that has been developed. The next chapter discusses a case study in which the method is implemented within a small company.

5.2 The Issues

It would be foolish to claim that software reuse is the solution to all the problems that have caused the current software crisis. Achieving software reuse on a level at which substantial benefits will be gained is a difficult task, and requires a great deal of commitment and effort.

Introducing reuse in a small company presents a different set of challenges to those faced by a large company. The larger scale of a big corporate reuse programme brings challenges associated with the size of the programme and the difficulties involved with changing the company's processes for structured software engineering. Many of the recommendations for software reuse considered in chapters 2 and 4 relate mainly to reuse programmes of this scale. For a small company, these considerations are significantly reduced.

In comparison, small companies tend to have a small team of software developers (often not more than 10) who are solely responsible for the development and maintenance of the company's software product(s). The size and complexity of the products is significantly less than those built in a large software factory. This has an impact on software reuse. For example, the creation and maintenance of a large component library is one of the key issues discussed in software reuse research. However, for the small number of components which would be available within a small company, problems with storing and finding components are much less significant.

Horizontal reuse is often very difficult, due to the narrow domains in which small companies tend to concentrate their efforts. Vertical reuse, however, is more available because of the narrow domains. This is an area which can be exploited in a small company reuse programme. This research concentrates on those small companies where there are no structured software processes currently in place. For them, the benefits which reuse offers seem unattainable because of the emphasis on considerable up-front investment and formalised processes which are recommended for successful software reuse.

As seen in Chapter 2, there are two main areas which must be considered for effective reuse within a company: technological and organisational [Stan84]. As technology has advanced, with the methods and tools to support reuse becoming available, the technological challenges facing reuse have been surpassed by the economic and organisational issues that face a company intending to implement a reuse programme [Trac88a].

The challenges facing any small company considering a reuse programme can be categorised into five main areas:

Initial investment – small companies do not have the time, money or resources to invest into a programme which does not have immediate returns. The risk is too great. It has already been shown in chapter 2 through experience that in order to gain the benefits of reuse, a considerable investment must be made first.

Lack of defined processes – all the successful reuse programmes discussed in the previous chapter have shown how companies have changed their processes in order to incorporate reuse. Small companies tend not to have processes in place which can be altered for successful reuse implementation.

Minimal resources – the development team in a small company is often only a few people strong. They are busy with developing and maintaining the products which are essential to the company's continued existence. They do not have the time, money, tools and other resources to dedicate to any extra workload.

Short time-scales – small companies tend to work to short, tight deadlines and short term goals. Long term investment which does not directly increase the company's capital is not a viable proposition. A reuse programme falls into this category.

Lack of experience – for a small company wishing to embark on a reuse programme, there are no examples of successful reuse programmes in other small companies for them to base their efforts around. Likewise, there are no examples of unsuccessful reuse programmes from which they could learn.

5.3 The Method

The method which has been developed as part of this research has been built to address the issues described in the previous section. One of the major challenges faced in developing the

method is that it must provide means for introducing a reuse framework into a small company, while reducing both the risk involved and the time taken before benefits are obtained.

The method presented below is based on the principles of organisational development and process improvement described in chapter 3, as well as previous work done in the field of reuse introduction, particularly by Karlsson et al. [Karl95] in the REBOOT project. The steps developed for this research have been adapted from the work done in the field of method study by Radford [Radf84] and Buckley [Buck85], as well as the other background investigation described in chapters 2 and 3. The following section describes the *Seven Steps to Success* when implementing a reuse programme. Each step of the method should be completed before moving on to the next stage and criteria are given in order to check whether the step has been successfully completed.

1) Gain the support of management and staff

The first, and perhaps the most important, step in introducing a reuse programme is to gain the support of the company's top level management [Bigg89b]. This is crucial. The introduction of a reuse programme affects all parts of the software production process in the company. Therefore, the support of the high level management in charge of all aspects of development must be gained so that the programme will be supported and implemented, and to allow changes to company policy to be made as needed [Hoop91]. The method recommends a well prepared and realistic presentation to key members of the management and staff describing both the benefits which reuse can bring and the difficulties involved in creating a successful reuse programme.

If this type of support cannot be obtained, then the reuse introduction project should be abandoned until such time as the commitment level changes. The level of commitment can often be measured by whether management are prepared to be involved personally in the programme, and whether they are willing to commit time and resources to its success. Small companies are characterised by owner management, and it is important that these owner managers are willing not only to be committed to the reuse programme, but to be actively involved in its success. The

risk of failure without full management support is too great at this stage to attempt any further work in the reuse programme.

2) Investigate the domain

The next stage of the method is to gain an in-depth knowledge of the company and its current working practices. This can be done by studying not only the development methods used, but also the company's product and the viewpoint of the staff.

It is recommended that the company's development methods, and the viewpoint of the staff are investigated by conducting informal interviews of certain key members of staff. This should include the manager and members of the development team. A questionnaire should also be used to gain information about both the work of the company and the staff. The investigation should not be an end in itself, but simply a means to reach the next step of the method.

The programme should only be abandoned at this stage if the level of commitment gained during step 1 has decreased during or after the investigation.

3) Identify areas for improvement

Target areas for improvement should be identified which would help the company to be successful in introducing a reuse programme. These areas should be determined using the investigation of the company conducted in the previous stage of the method. The target areas should be based on key areas in the company where changes in working practices could make the development environment more conducive to the growth of a reuse programme.

However, major changes should be avoided initially. As concluded in the previous chapter, an incremental approach to implementation of the reuse strategy should be used. This is because, with an incremental approach to reuse, reuse techniques can be tried and proved on a small scale

before introducing major changes to the company [Prie91]. Also, reuse target areas can be identified where reuse will be most effective.

Some suggested areas for improvement are:

Planning – "The company that fails to plan, plans to fail". Without proper planning for software development, the potential benefits which can be gained from reuse cannot be maximised because opportunities for including reusable components may be missed.

Design – The use of a design method which supports both building reusable components and including components in system development can be a great aid in the reuse programme.

Resource Management – In order to implement a reuse programme, reusable components must be available to software developers. Resource management can help to make this happen.

Documentation – Developers must be able to understand components in order to be able to reuse them. As seen in chapter 3, documentation can aid the understanding process.

Although incredibly unlikely, it is possible that no areas for improvement can be identified. If this is the case, then the programme should be abandoned at this stage. However, unless this is the case, the only other reason to abandon the program at this stage is if the management and staff are not willing to invest their time and resources into making the suggested changes. Their commitment to change can be improved by using them as part of the identification of areas for improvement. Indeed, the greatest commitment is often shown when the staff involved in software development come up with the ideas for areas of improvement. Commitment levels can be gauged by discussing the proposed improvement areas with participating staff, and the programme should only proceed when their full support is given and the appropriate resources are committed to the programme.

4) Define appropriate 'lightweight' processes

Lightweight processes are defined as software processes which are informally defined in terms of recommended working practices for company staff, which are repeatable during the software development lifecycle. 'Lightweight' processes are proposed as part of this research to avoid the large amount of resources which must be committed and documentation which must be produced in a formalised process improvement scheme. They do not require formal definition, training, documentation or management in order to achieve their objectives. These informal processes are the first step to formalising the software process, and as such, are expected to be subject to change and can be discarded if unsuccessful. The informality of these processes makes them ideal for small companies, because improvements to their working practices can be tried and tested before the successful process recommendations are formally accepted.

These 'lightweight' processes are based on the areas for improvement identified in the previous step. The 'lightweight' processes are manifest as a set of recommendations to the company's staff on working practices that will best support the reuse programme. As with the previous step, including the staff in the 'lightweight' process definitions improves their commitment to change, and allows them to capitalise on their current best practice. These recommendations must also be directly linked to the benefits which reuse can bring to provide the motivation for their use. These recommendations should be presented to all those involved in the reuse programme. Again, only when the support of both the management and technical staff for the recommendations is assured should the programme proceed.

5) Select a pilot project

It is not wise to jump straight into a new development strategy that will change the way that the company works without first being assured that the strategy is applicable to the company, its staff, its domain and its working environment. A pilot project allows the 'lightweight' processes to be tried in practice. The pilot project will be a project that is indicative of the type of work

done within the company. It gives members of staff at the company hands-on experience with software reuse techniques.

The pilot project is also a great opportunity to try out recommended reuse techniques to see how effective they are in a real situation. Not all techniques will be equally effective, and the pilot project should highlight those reuse techniques which will be of most benefit to the company. Although software reuse can offer major productivity gains when used in the right way in the right environment, it is not the solution to all problems. There will always be times when it will be more effective to write new code than to try to find and reuse previously written code. The key is to recognise which techniques will be most effective in different development environments, and utilise the most efficient development strategy in each case. The pilot project should also help to identify areas where tool support would assist the developers in achieving the goals of the reuse programme.

An appropriate pilot project is one which is typical of the work done within the company. If an appropriate project cannot be found, it may be wise to wait for a later opportunity rather than using a pilot project which will not allow the suggested techniques for reuse to be properly implemented. However, using an atypical project because it would allow the best results to be seen from the reuse programme is also not ideal, as it will give unrealistic results for the next stage of the method. During the lifetime of the project, the work being done should be monitored with respect to the 'lightweight' processes recommended, so that the results can be evaluated at the next stage. When the pilot project has been completed, move on to the next stage of the method.

6) Based on the results of the pilot, make a plan for integrating reuse into the company

It is important to learn from the experience of the pilot project, so that when reuse is integrated into the company as a whole, tried and proved techniques will be used. Members of staff can have confidence in the changes that will be made, because they have seen the success of the pilot

project. Once the results of the pilot project have been analysed, a new plan for the software reuse strategy of the company should be drawn up based on these results.

It is very important at this stage to identify what did and did not work well in the pilot project, and to consider how the problems highlighted by the pilot can be better addressed. The greater difficulties involved in the wider introduction of reuse throughout the company mean that the reuse programme should only go ahead if it is felt that the successes of the pilot can be transferred to other projects, and that the problems highlighted can be successfully addressed.

The successful parts of the pilot project form the basis of the plan for more widespread reuse introduction. During the course of the project, areas where tool support would have assisted the project should also have been identified. These requirements can be used to procure or develop tools which meet the needs of the reuse programme. If management and staff are still committed to the reuse programme, then further improvements to the programme can be identified and implemented by returning to step 2 of the method.

In cases where the pilot project has completely failed to bring any benefits from reuse, examination of the results of the project should be used to discover why. Failure will usually be caused by one of the following three reasons:

a) the areas for improvement and 'lightweight' processes used did not address the right areas to help the company to capitalise on the benefits of reuse,

b) the staff did not actually implement the 'lightweight' processes in their work or

c) reuse is not an appropriate technique to achieve benefits in the company's current climate.

If either of the first two reasons are identified, the method must be reapplied from step 1 before any progress can be made. If the third is genuinely the reason for failure, and there is no scope

for reuse in the company's software development, then the reuse introduction programme should be abandoned at this point.

However, if the pilot project has been successful in bringing the benefits of reuse to the company, then the next step of the method should be followed only when the full support of the management and staff of the company is received for the reuse plan.

7) Incrementally implement the plan with automated support

Once a plan has been formulated, it should be put into action. This seems obvious, but it is important to consider how the plan will be implemented. As described in the previous chapter, it was decided to use a method of introducing reuse ideas while gradually encouraging the improvement of development methods.

The incremental approach was recommended in this research in order to allow the company to slowly change their working practices at the same time as fulfilling their customer's requirements. This will give the staff a chance to get used to the idea of a reuse framework. It will also allow the new development methods to mature and become a standard practice within the company without an extensive overhaul of current working practices. The progress of the reuse programme against the plan should also be measured, to identify how the programme is progressing, and to update the plan, if necessary.

Automated support is also a recommendation of the method. With tools to support the reuse programme, the impact of the changes that need to be made can be reduced. The tools should be easy to use and provide support for creating, finding and using reusable components. By this stage of the method, areas where tool support would assist the programme should have been identified.

These areas should be used to define the requirements for tools support. Typically, tools which will assist the developers within a small company to understand; store and retrieve; and

incorporate reusable components within their source code will be valuable in automating support for the reuse programme. These three areas are very important in order for developers to be able to achieve effective reuse (see section 2.5).

Investigation was conducted into each of these three areas in order to identify what the tools would need to provide in order to address them. There have already been several tools developed for the retrieval of reusable components from a component library. Much of the work on these tools is in the identification of potential reuse candidates from a large collection of components. However, they always rely on the developers and/or the repository administrator to ensure that good information on the components stored within the library is available. No tools have been seen which support the automatic generation of information about components that are stored in a component library.

The tool set proposed to support the reuse programme in this research will concentrate on automating support for the reuse programme. The tools will integrate retrieval of reusable components with automatic generation of information about those components.

The method developed is summarised below:

Step	**Criteria for continuation**	**Action if criteria has failed**
1. Gain the support of management and staff	Full management support is obtained	Abandon programme or attempt to increase level of support
2. Investigate the domain	Continued support for reuse programme	Abandon programme or return to step 1
3. Identify areas for improvement	Appropriate areas are identified and agreed	Abandon programme, revise selected areas or return to step 1
4. Define appropriate 'lightweight' processes	Recommendations are fully accepted and supported by management and staff	Abandon programme, revise 'lightweight' processes or return to step 1

5. Select a pilot project	Appropriate typical project is found, supported and completed	Wait for appropriate project, abandon programme or return to step 1
6. Based on the results of the pilot, make a plan for integrating reuse into the company	Benefits obtained from pilot which can be transferred to whole company. Plan is fully accepted and supported.	Abandon programme if reuse techniques not appropriate for company or return to step 1
7. Incrementally implement the plan with automated support	Reuse success transferred to all projects	Abandon programme in areas where reuse is not successful

5.4 Conclusions

The method described in this chapter is based on the tried and tested business techniques described in chapter 3 as well as previous software reuse research. It was shown in chapter 4 that there have been several reported reuse successes in large companies. Based on the solutions discussed in the previous chapter, the method described uses a combination of 'lightweight' processes with automated support for the reuse programme to reduce both the effort and the risk involved in introducing reuse in a small company. *Seven Steps to Success* were presented for the introduction of reuse within a small company.

However, without testing the method, the suggestions made in this chapter are simply that. The next chapter describes the implementation of the method at Public Access Terminals Ltd., a small software development company. The case study described tests the method discussed in this chapter. A description is also given of the development of a tool set which supports the reuse programme by automating some of the tasks required to allow reuse to be capitalised on.

Chapter 6: Reuse in a Small Company: The practice

6.1 Introduction

This chapter discusses the implementation of the method described in chapter 5. A case study using the method has been conducted in association with a small software development company. The challenges faced in this environment are discussed, along with the incremental approach used for introducing software reuse into the company.

The development of a set of tools for automating support for the reuse programme is also described. The tool set integrates a reuse repository management tool with automatic processing of source code to generate information about the reusable software.

6.2 The Company

This case study has been conducted in association with Public Access Terminals Ltd., a small computer systems manufacturer who have a single product in the public access and security domain. Their system keeps information on all the people that are currently present at a particular location and can issue and check security badges. The software of the system is connected to specially designed hardware peripherals, as well as being networked across a site using LANs. The system considers many aspects of computing from database manipulation to interfacing with peripheral hardware devices to image handling. The company deals with both software and hardware, and uses technologies such as device drivers and networks.

The company had realised that, with the pressure being applied to its product by customers and the competition, it was time to start using more structured software methods in their software development department. This, along with two teaching company scheme placements in

association with the University of Durham, encouraged the company to look to the expertise of the university in helping to improve its methods.

6.3 The Case Study

The method described in the previous chapter was used in association with P.A.T. Ltd. to attempt the introduction of a software reuse programme. The company followed the *Seven Steps to Success*. Each of these steps as implemented in the case study is described below:

1) Gain the support of management and staff

In this case study, we gained the support of the high level management by giving a presentation on reuse, explaining how it could help their company and how best to utilise it. This presentation was given to the company's technical manager and key members of the hardware and software development teams. It was a good opportunity to present the case for reuse, stressing the benefits that it could bring to the company, and the approach for introducing reuse into a small company that we were recommending. It was also a good point at which to get feedback from the management on what they expected from the reuse programme, and how they wanted the company to change for the future.

It was found that the management were very dubious of the reuse successes reported, as they all related to large corporations. They were not sure how the successes could be related to their company. Their key concerns can be summarised by one of the questions which was asked after the presentation: "We're only a small company and not very structured. Can we still do reuse and is it worth it?"

There was considerable discussion of the challenges that would be faced when implementing a reuse programme within the company. This centred around the changes which would have to be made within the company and the resources that would be required. However, the management

felt that with the incremental approach recommended, the time and resources which would have to be committed to the reuse programme could be minimised. This gave them confidence that the benefits of reuse could be made available to them, and after discussion with the manager of the company's technical development department, it was with enthusiasm that the company agreed to continue with the proposed reuse introduction project.

2) Investigate the domain

The next stage of the method is to gain a working knowledge of the company and its current working practices. This was done by conducting informal interviews of certain members of staff, including the technical manager and members of both the software and hardware development teams. A questionnaire (Appendix A) was distributed to each interviewee before the interview. It was not expected that the questionnaires should be filled in by the staff being interviewed. Rather, it served as a focal point during the interview to give each member of staff an idea of the type of question that would be asked, and the type of information being sought. Notes were taken during the interviews, and the interview with the technical manager was recorded, with permission, in order to study the information gained at a later time.

It was found that the company's methods were very ad-hoc. The developers worked in the way that they found most suitable. Informal communication between the developers helped to clarify the interoperation of the various parts of the system that they were working on. The development team kept only one version of their software product, to which they made all alterations. This ensured that they did not have multiple differing versions of the software in different locations. Although it solved problems with software version management, it created a very difficult to understand, monolithic software system.

They did not have a formalised process for development or maintenance. Their work was based very much on customer requests. When a new customer was obtained, they made additions to the product (if necessary) to cater for the new customer, then installed the new version of the software at the customer's site. Their customers often requested technical support and

modifications to the system, most of which were handled by the development team. There were no specified design methods used, each developer used his or her own preferred method of working. Little or no documentation was kept on the software, apart from the user manual.

It was also found that the staff were keen to improve their development processes. They seemed excited about the opportunity to move their system to a new operating system environment. They wanted to gain the benefits of reuse in the new project. This commitment encouraged both the management and staff to continue with the reuse programme.

3) Identify areas for improvement

The results of the interviews provided very valuable insights into the attitudes and working practices of the company's staff. The staff seemed keen to see the company become a more competent software house in the future. Formalised methods, better planning and the introduction of structured processes were suggested as ways to achieve this. However, most of the company's current plans for the future were based solely on further modification and redevelopment of their software products.

One of the key areas for improvement was a change to an improved operating system. Rather than using MSDOS, the company decided to move to Microsoft® Windows® as the operating system for their software. This would give them access to improved development environments with greater support for reuse. It was also seen that better planning for projects would enable the developers to recognise opportunities for reuse, rather than simply basing their development strategy on requests from customers. Along with this, using a structured design method could also aid the developers in reusing their software.

To support the reuse programme, both management of software resources and software documentation would make reusable components easier to find and understand. As some of these areas for improvement were suggested by the staff at Public Access Terminals during the previous stage of the method, and they were involved in identifying what should be done to help

improve the company, there was little difficulty in receiving their full support for the improvements recommended.

4) Define appropriate 'lightweight' processes

Based on the results of the investigation, a strategy for adopting software reuse techniques was recommended. Suggestions were made on how to set up a reuse programme within the company, along with the amount of the developer's time would be needed to support the reuse programme and what other resources would be required. The resources included a good technical environment and an area of the company's network set aside for reusable components. The company decided to use the Microsoft® Visual C++ development environment.

Recommendations for 'lightweight' processes were made to support the introduction of structured techniques for the following:

Planning and reviews - It was recommended that meetings be held on a regular basis to ensure good communication within the company. It was suggested that during the initial stages of a new project, the meetings were used in order to plan the project in advance. Then, as the project advanced, these meetings could become more focused on technical issues and lower level design and implementation considerations. They would then become a chance to review what has been done so far and plan ahead for the next stages.

Design - It was recommended that an object-oriented method of design be used to support the reuse programme. Object orientation was suggested as a design method because it supports reuse, and would allow for the provision of reusable design techniques and components in software development and maintenance. A survey of object-oriented methods was conducted [Bigg95], and made available to the company. This allowed the company to compare the different methods, and a decision was made to use the Object Modelling Technique (OMT) method described by Rumbaugh et al. [Rumb91]. The main reasons for this decision were that, at the time, it was the most popular of the standardised

OO methods in the software industry [Leac94] and there is considerable tool support for the method [Bigg95]. This move to an OO design method, tied in with a decision to move from a C style of programming to the full use of C++ as their main programming language for development, would give the developers both a sound design method and a good technical environment which both inherently support reuse.

Resource Management - It was recommended that the work done within the company be kept in a reuse repository. This would allow developers to have somewhere to store their reusable code. It was expected that the repository would be a central storage location to which all staff would have access. This repository would be where reusable code which had been written could be kept for inclusion in their software by any of the development staff.

Documentation - It was found in the company's software development process that when the pressure was on, documentation was invariably the first casualty. It was, therefore, recommended that a minimum level of documentation be kept in the company, with extra documentation to be completed as needed.

The areas described above were identified from the study conducted within the company as major target areas for improvement in order to support the reuse programme. These are the major areas of a company's process which will support a software reuse programme, and these areas were especially valid in the case of Public Access Terminals.

After discussion with the manager of the technical development department, it was felt that the areas identified were appropriate for the company, and the reuse programme moved on to the next stage.

5) Select a pilot project

A section of the full system, the *FotoFile* for grabbing images from a video camera, was chosen as the pilot project. Although the developers knew what they wanted the software to do, the objectives for the system being developed sometimes changed quite dramatically. Often, a greater realisation of the work being done by competitors, and the expectations of their customers, induced a change in the direction of the development.

Originally, the plan was to develop the full security access system in Visual C++. The system included a database for storing details of personnel as well as other components which communicated with various peripheral devices. The *FotoFile* was one of these components, and it was originally expected to be built into the full system.

During the course of the pilot project, tight deadlines had to be met. These were caused by a trade show, at which the new version of the software needed to be demonstrated; and requests from new customers for the software to be modified. An estimation of the time it would take to complete the *FotoFile* was given, and it was expected that the project would be completed in time to be demonstrated at the trade show. However, customer requests for modifications to the old system hampered the development of the new system. As the deadline approached, the developers worked with less regard for the reuse recommendations made, in order to get the software working in time. It was when the pressure was off that the recommendations were reviewed, and the code written was reconsidered in order to see if it could be made more reusable. It was originally expected that the recommendations would be followed throughout the lifecycle of the pilot project. However, it was seen that the emphasis on reuse was giving the developers motivation to spend more time planning their code in advance. They were also encouraged to go back to the code once written and restructure it in order to make it more object-oriented and reusable.

Considerable success was gained in the pilot project when the developers gained a greater understanding of the Object Linking and Embedding (OLE) features provided in Visual C++

under Microsoft® Windows®. After some investigation, it was found that the use of OLE would allow the *FotoFile* to be built as a stand alone object, rather than as an integrated part of the full system. OLE is a standardised mechanism for allowing data created by different Windows® applications to be integrated into a single file. These "compound documents" seamlessly access the different applications for creating and editing the various types of data they contain. [Micr93b]

The aims of the pilot project altered, the new goal becoming to make the *FotoFile* an OLE server. This allowed the development of the object to be achieved in complete isolation to the rest of the system. There were many advantages to this style of development. The developers did not need to know the details of the full system being developed in order to successfully complete their project. This was of great benefit to them, because, as has already been emphasised, the proposed system often changed in its objectives. It was, therefore, very useful to have an encapsulated section of the system to work on. Once this strategy was decided upon, the pilot project was successfully completed in 4 months.

The greatest benefits derived from the pilot project were achieved when the company recognised an opportunity to enter the component market. Another company working on the same type of system, in consultation with one of the developers, were impressed with the *FotoFile* and saw it as a perfect addition to their own system. Using OLE, the *FotoFile* server was working successfully with their system in under 2 hours, which also impressed the company. A contract was soon formed, in which the system providers gave a royalty to the component provider for every system sold which included their component. The value of the contract was considerable, and the royalties from the deal provided much needed capital to the company at a critical period for funding their further developments.

The success which had been seen during the pilot project gave the staff and management confidence that the reuse programme would work for them, and they were very willing to continue with the reuse programme. In fact, their main concerns focused on where they could

apply the same techniques to achieve the same results rather than with the challenges that would be faced when implementing reuse on a broader scale. Such benefits from the pilot project cannot always be guaranteed but, in this case, the company's success encouraged the staff to continue with the reuse programme.

6) Based on the results of the pilot, make a plan for integrating reuse into the company

In analysing the results of the pilot project, it was recognised by the development team that the *FotoFile* component developed would provide greater flexibility for the overall system. Due to the reuse strategy considered in the development of the *FotoFile*, the result of the pilot was that a reusable component was built.

This success allowed the company to reconsider their original plans for the development of the system. In the original plan, the full system was to be developed in Visual C++. However, as the main system was a database management tool for keeping information about the people currently at a particular site, it was recognised that using a database application generator for that section of the system would make the development much quicker and easier. As the *FotoFile* component developed would be easily integrated into a full system, the developers looked for a different development environment which would make the full system easier to implement.

It was decided that the database would be developed in Microsoft® Access, rather than C++ as originally planned. This was a considerable success in terms of the development of the entire system, as using an application generator such as Microsoft® Access meant that the overall system would be completed much more quickly than originally anticipated. The flexibility provided by the 'lightweight' processes within the reuse programme allowed the developers to change their plans midway through the programme.

Following the success of this component based development, another OLE server was also proposed to support the full database system. This component would follow a similar style of

development to the *FotoFile*. This was an object for designing and printing identification badges known as the *Badge Server*. This would complement both the *FotoFile* and the main database, allowing customised security badges to be designed and issued.

Some success had been achieved in the case study thus far; however, there were two major problems which were identified during the course of the pilot project. These were the lack of experience in using an object-oriented design method, and the lack of tool support for the developers.

It was felt that with suitable CASE tools, object-oriented development could be better employed, and it would be easier to collect written code into a reuse repository for use by other applications. At least two of the recommendations made to support the reuse programme could be aided by automation. These are resource management and documentation. One of the least successful of the 'lightweight' processes suggested for the pilot project was keeping software documentation. As has been mentioned, it was found that the developers tended to develop code, then go back to the code to try to abstract reusable components from it [Lane84]. Writing documentation after all the interesting work had been done was recognised as least important part of the programme in the eyes of the developers.

Tools which aided this process by giving the developers information about their code would be valuable in the reuse programme. To support these improvements, it was also considered that tool support would aid the developers in implementing these 'lightweight' processes, particularly in the areas of documentation and implementing a reuse repository. If these two processes could be tool assisted, the developers could concentrate more of their time on improving their system development using object-oriented design and communicating with planning and review meetings - where the more challenging work of software development is concentrated. The mundane tasks of software documentation at a minimum level and the storage and retrieval of reusable components would be simplified by the introduction of tool support to automate these areas.

7) Incrementally implement the plan with automated support

The plan for reuse at Public Access Terminals was to follow the same style of development which had been used in the pilot project to complete other components which would plug into the full software system. The same development methods would be used as recommended during step 4 of the reuse programme. Also, a set of tools would be developed to support the reuse programme.

The first part of the plan was to implement the *Badge Server*. The code was again written in Visual C++, with the *Badge Server* being implemented as an OLE server. In this second project, the developers used the lessons they had learned from the pilot project in maximising the potential benefits available in the reuse programme. The developers were not only developing the component for reuse in the main system, but used reuse techniques in the development of the component itself.

In the finished product, there was a total of just over 20,000 lines of code. Of this code, 43% was inherited from the standard libraries available through the Microsoft® Foundation Classes (MFC). Of the remaining 57% of the code, 24% was automatically generated by the Visual C++ wizards. Of the remaining code which was written by the software developers, 31% was abstracted into reusable classes which were used throughout the application. This gives a total reuse factor of 70% for the whole project. These results were calculated by identifying which of the standard library classes were called by the source code and totalling the number of lines of code in those classes; then calculating the number of lines of code automatically generated by the application and class wizards in Visual C++; then measuring the number of lines of code in the classes that were abstracted out into the reuse repository.

The second part of the plan was to implement the database and integrate the two components built during the reuse programme. This proved to be incredibly straightforward because of the

It was decided that the tool set should provide 3 main areas of functionality.

1. Reverse engineering source code to an OO design representation.
2. Re-documenting source code.
3. Managing the storage and retrieval of reusable components.

The first two functions would be aimed at helping developers to understand reusable code, and, through understanding, find it easier to reuse. The third function enables those developing for reuse to store the components for later retrieval, and those developing with reuse to search the component repository for suitable classes to reuse. The next section describes the tool set which automates support for a small company reuse programme developed as part of this research.

6.4 Automated support for the reuse programme

Automated support for the reuse programme at Public Access Terminals was to be provided with a set of tools which would provide support in the three areas identified above. These areas are: understanding; storage and retrieval; and incorporation of reusable components.

To provide maximum support for the reuse programme, the tool set was designed to function with C++. It was found that the class is the main object of reusability in C++, and that class libraries, when used effectively, can be very useful in building applications. Particular attention was paid to the operation of the Microsoft® Visual C++ development system, as this was the key system used at P.A.T. In designing the system, it was decided that the requirements discussed in chapter 5 would be best fulfilled with a set of tools to generate information on C++ source code and use that information to store and retrieve reusable components.

The information would be given in terms of an object-oriented design notation, and documentation of the code, which it had been seen that developers did not have time to write. A set of tools that could also aid in storing and retrieving reusable components, as well as giving

technology used. The full system was ready for the market ahead of schedule and was well received by the company's customers.

The requirements for the set of tools to support the reuse programme were identified from the results of the pilot project, and in discussion with the technical staff at Public Access Terminals. The tool set was required to work with C++ software, which made use of classes and available component libraries. The results of the requirements analysis for the tool set are given in terms of the three areas identified earlier.

- **Understanding** – From investigating available component libraries, it was found that the main aids to understanding a class library are: a class hierarchy chart describing the inheritance relations and structure of the class library; and documentation describing the purpose of each class and it associated services and attributes. (For example, see the Class Library Reference for the Microsoft® Foundation Class Library [Micr93c].)
- **Storage and Retrieval** - Many different systems are used in the area of information systems for the storage and retrieval of data. However, many of the challenges to effective information retrieval for large data sets do not exist in this case. This system will be dealing with small sets of in-house company software components being stored for later retrieval when constructing new applications. Based on the factors discussed in section 2.6.2, and a previous project on the storage and retrieval of software documentation [Bigg93], it was decided to use a simple class information storage system with IR techniques for retrieval.
- **Incorporation of Reusable Components** - Once a suitable reuse candidate has been found, there are two main factors which determine whether a developer can incorporate that component into their system. These are the quality of the component and the understanding which the developer has of the component. Although quality standards for software engineering can be recommended for component development, a tool cannot ensure that these are being adhered to. The first of these two is therefore outside the scope of this tool set. The second, however, can be assisted using the techniques described above.

information on those components, would be valuable to the developers in the reuse programme. The information about the components would be based on object-oriented design notation and documentation of the source code.

The prototype tool set works on simple but effective principles. One of the key criteria on which the development is based was making the tool set as fast and easy to use as possible. This is to enable the developers to use the tool set to aid reuse without the large overheads, previously discussed, which discourage small companies from incorporating reuse techniques. The development of the prototype tool set was achieved in three main stages, each stage being completed in consultation with the staff at P.A.T.

6.4.1 Development of the Reverse Engineering Tool

The first stage of development was to build a tool which would reverse engineer C++ source code to a diagrammatic depiction of the class inheritance hierarchy. It was decided to perform static analysis on C++ header files, reusing a previously written C++ parser called Docclass[2] in the construction of the Reverse Engineering tool. The tool collects information about the classes contained in the code by parsing the C++ header file and reading in the appropriate information. The information is stored internally as a collection of objects containing class data. These objects are then formatted for output. The output format chosen was based on the Object Modelling Technique notation [Rumb91]. This is because it was found that, at the time, OMT was the most popular standardised OO design method currently being used in software companies [Leac94]. It was decided to interface with a currently available, and popular, object-oriented design tool called OMTool to display the results of the Reverse Engineering tool.

In a very simple prototype form, this tool was given to the developers at P.A.T. They felt that the tool had potential to help them in seeing how their development was structured. They

[2] Docclass © 1993 Trumphurst Ltd. The source code is publicly available and has been used with the author's permission.

interchange. The format chosen for the documentation produced was based on the structure of a maintenance document produced by one of the company's software developers.

More complicated was the task of adding support for project or make files. This was accomplished as follows: each header file contained in the make file was processed in turn, the full collection of classes found being used for output to both formats.

It was seen that both class hierarchies and documentation could be viewed by Web browsers such as Netscape, by using HTML and Java applets embedded within the Web pages. Again, using the information contained in the internally stored collection of class data, new output procedures were written to support output to HTML and Java. The use of Web browsers could support software libraries over both intranet and the Internet, as well as having the advantage of allowing the power of the Web browser's searching facilities to be used on the software documentation.

The new version of the tool was delivered to staff at P.A.T., who were impressed with the new interface, its ease of use, and the availability of automatically generated source code documentation. However, they felt that version information for the documentation would be useful, along with the proposed reuse support.

6.4.3 Development of Reuse Support

Classes could now be parsed and information output in three formats: OMTool class hierarchies, RTF documentation and Web pages. The final stage of the development was to build a reuse repository support tool into the tool set. There are many case tools which reverse engineer and re-document code. There are also tools which provide support for reuse libraries. None have yet been seen which integrate the two, allowing the information gained from reverse engineering and re-documentation to aid developers in reusing their code.

suggested making the tool usable for projects as well as single source files, and making the tool easier to use.

Once the prototype Reverse Engineering tool was working, it was incorporated into a Microsoft® Windows® 3.1 text processing program to allow the user to edit the source files before reverse engineering them. This provided a Graphical User Interface for the Reverse Engineering tool. Microsoft® Windows® was chosen for the application's operating environment, as this was the operating system in use at P.A.T.

6.4.2 Development of the Re-Documentation Tool

It was decided to use the comments from the source code to generate documentation about the code. It has been seen that software documentation can substantially aid a developer's understanding of software systems (as discussed in section 3.5.2). However, it has been reported that there is no significant difference in the effort required for programmers to understand code between commented and uncommented versions of source code when indentation and meaningful identifiers were present [Weis74]. It was, therefore, decided that using the comments to generate structured documentation is a valuable exercise in aiding program comprehension. It would also allow the developers to document their code by commenting it as they wrote it, and then use the tool to generate well structured documentation without any further effort.

Developing support for automatic documentation generation was achieved using the information about the classes contained in the C++ header files extracted by the Reverse Engineering tool. This included the comments associated with each class and its associated members. Based on the information extracted by the parser, it was a relatively simple task to incorporate a new output procedure which gave output to Rich Text Format. This format was chosen because of its text based nature, along with the availability of formatting codes to structure the documentation. It has also been recognised by Sommerville [Somm96] as a defacto standard for documentation

The class information was now stored in a new format which allowed the generation of reuse repositories, containing information about each class and its functionality. These repositories can be built, added to, saved and searched for classes matching a search criteria. Again, based on research in the field of retrieval as well as previous work in this area [Bigg93], it was decided to use a boolean query language (which uses AND, OR and NOT connectives to create a list of the terms which are required) for building search criteria. This is because the system is designed for use by software engineers who will be used to the concept of boolean connectives and it is felt that these users will appreciate the directness and specificity that a boolean search term would offer.

The completed prototype tool set was delivered to the staff at P.A.T. The results of its use are discussed in the next chapter.

6.5 Conclusions

The case study has been a very useful view of the workings of a small company under pressure to meet customers change requests, and demands for new products. This is a considerably different environment to large, well structured software companies, with a different set of challenges. It has been seen that both technological and organisational improvements are required for the implementation of a reuse programme. This was as expected, although it has been seen that the introduction of reuse has encouraged and inspired the staff to improve their development ideas and processes.

The method described in chapter 5 was implemented in full. At each stage, consideration of the results which had been obtained up to that point formed the criteria for moving on to the next stage, and it was only when the support of the management and staff was assured that the reuse programme continued. Although there were difficulties, the method was not abandoned at any one of the seven steps, because the criteria for continuing through the method were met at each stage.

The organisational considerations for introducing reuse into any company have already been considered in depth in other projects. However, prior to this research, the unique challenges of a small company and the technology to support such a process introduction have received little attention. It was seen that, although reuse alone can offer significant benefits to the company, improvements in general development practices and software processes could help to maximise those benefits. It was considered that the 'lightweight' processes recommended were simply the first step in this improvement process. Based on the success of the pilot project using these recommendations, further work should be done in further improving the company's development methods to capitalise on reuse.

This chapter has also described the development of a set of tools to support reuse. Tools for reverse engineering and documentation generation have been integrated with a reuse repository support tool to aid in automating the reuse process. It was seen in Chapter 2 that technological support for a reuse programme can aid developers in capitalising on reuse. It was also seen in Chapter 3 that an integrated tool set could allow information abstracted from source code to be used by software engineers in understanding the code.

ReThree-C++ addresses these issues. The prototype tool set was developed after the initial stages of the incremental introduction of the reuse programme in P.A.T. so that the real needs identified during the programme could be addressed. The company's developers were also consulted throughout the development of the prototype, so that the tools would be well suited to assisting them in the reuse programme. The tool set was used by staff at the company, and an assessment of its use is given in the next chapter.

The prototype tool set integrates the abstraction of useful information from the source code of reusable components with reuse repository facilities. This allows developers to use the tool set in conjunction with their standard PC office tools to view the information generated. They can also easily add components to a reuse repository and search for reusable components.

It was seen that the introduction of new technology and the commitment of management and staff to reuse can make a difference in the development process. Both can work independently to bring improvements, but applying the two together made a significant difference to both productivity and profitability.

The next chapter evaluates the work described in this thesis, including a discussion of the results gained from the incremental introduction of a reuse programme at Public Access Terminals Ltd., and detailed evaluation of the prototype tool set.

Chapter 7: Evaluation of Results

7.1 Introduction

This chapter evaluates the results of the work described in this thesis. The results of the research which has been conducted are evaluated in two main sections.

The first evaluates the results of the implementation of a software reuse programme at Public Access Terminals Ltd. The success of the incremental approach to implementing the reuse programme is considered, as well as the results of the reuse programme in the development of software within the company.

The second section evaluates the integrated tool set, ReThree-C++, which has been built to aid the automation of reuse support within a small company. Its applicability within Public Access Terminals Ltd. is considered. The results of an experiment to test the validity of the CASE tool are also discussed, along with a consideration of the general operation of the tool set.

7.2 Results of the Reuse Programme

In considering the results of the incremental approach to implementing the reuse programme and the results of introducing a reuse programme as part of the software development process within Public Access Terminals Ltd., three main issues will be considered:

1. The success of reuse within the programme. Success is measured simply by identifying whether reusable modules were built, and the extent to which reuse was achieved in the software developed.

2. Benefits brought to the company by the reuse programme. As discussed in section 2.4, benefits will be identified in terms of:
    - Increased speed of production
    - Financial benefits to the company
    - Increased quality of software
    - Ease of maintenance
3. The problems faced by the company in implementing the reuse programme. There will also be some consideration of the techniques and practices which were not adopted within the company.

7.2.1 Success of the Reuse Programme

The success of the reuse programme is to be measured by considering whether the method described in chapter 5 was successfully completed and by identifying whether reusable modules were built, and the extent to which reuse was practised in the company's software development. Using this criteria for success, it can be recognised that the reuse programme was successful. Each of the *Seven Steps To Success* were carried out, and at each stage, the criteria were met for moving on to the next stage of the method.

Also, two significant areas of the company's software system were built as reusable components, and each of these was integrated successfully into the full system.

Implementing the 'lightweight' processes for reuse when building a software component (the *Badge Server*), the developers made a special directory for reusable C++ classes. In that directory, 16 classes were stored in 9 different files, each of which was made available to the whole system for reuse. The classes were abstracted from the software developed and made available as reusable classes. They were used throughout the system under development.

Simply having the reuse directory as a repository for reusable classes has been a success for the company. Whenever considering the reuse programme, the staff can readily see its influence by the existence of that directory and always refer to the reuse directory in discussing the programme. It is particularly beneficial when displaying the success of the programme to top level management, as there is a tangible representation of the programme in the classes contained in that directory.

It must, however, be pointed out that the reuse directory came about after the system had already been built in prototype form. The developers were under a tight deadline to have a prototype of the system ready for display to their customers, and the reuse guidelines recommended were not really considered until after the working prototype had been developed. Then, using the guidelines for reuse, the developers reviewed their prototype, identified commonalities within the software, abstracted reusable classes based on those commonalities and finally built the reuse repository with those classes.

It was seen that a reuse factor of 70% was achieved in this project. It could be pointed out that achieving this level of reuse is a success. However, measuring a reuse factor is simply an estimation of the lines of code that have been reused in relation to the total number of lines of code in the project. There is no consideration of how difficult it was to identify, understand and incorporate those lines of code, or if the reuse was valuable. It is far more interesting to identify the real benefits that have been brought to the company as a result of the reuse programme.

7.2.2 Benefits to the company

It was seen in section 2.4 that there are four major areas in which benefits can be derived from a reuse programme. The benefits derived within P.A.T. from the reuse programme will be considered in these four major areas.

1. Increased speed of production

Both the pilot project and the subsequent development in the reuse programme were built to meet specific deadlines. One of the difficulties of measuring whether the speed of production was increased by the implementation of the reuse programme is that in both cases, the requirements for the software changed as the software was developed. Any initial estimates of the time the software would take to build were based on the original understanding of the software's functionality.

The pilot project (*FotoFile*) was built as a reusable component for the full system being developed. It was completed in time for its deadline, and therefore, it can be concluded the reuse programme did not increase or decrease its speed of production.

A prototype of the subsequent development (*Badge Server*) was built in time for its deadline, however, the reuse work was not conducted until after the deadline had been met. It can therefore be concluded that the reuse programme decreased the speed of production of this component.

The real benefits came in the overall system. Due to the ease of integrating the reusable components into the main system, it could be built using a 4GL database generator (Microsoft® Access). This considerably increased the speed of development of the Windows® version of the software (as compared with the time it would have taken to build in Visual C++), as the main system was basically a database control system. The other, more unique parts of the system (including the *FotoFile* and *Badge Server*), could still be integrated into the full system because of the OLE properties built in as part of these reusable components. The flexibility to achieve this was only available because of the principles on which the reusable components were built.

2. Financial benefits to the company

Two major financial benefits were gained, either directly or indirectly, as a part of the reuse programme.

The first was due to the increased speed of production of the overall system, which was gained thanks to the flexibility given in the choice of development environment for the system as previously discussed. This enabled the company to release the Windows® version of the software earlier than was expected. This pleased current customers who were waiting for the updated version of the software, and also gave the company a better opportunity to compete with other software systems that were currently available.

The second, more direct, benefit came from the opportunity to sell their image processing software (*FotoFile*) as a reusable component to another company. This contract brought a very large, previously unexpected, financial boost to the company, which helped to fund the further developments that were required both for the reuse programme and the system as a whole.

3. Increased quality of software

One of the advantages of the technology employed for building the two main reusable components (*FotoFile* and *Badge Server*) was that once the components had been built and tested successfully, they were easily incorporated into both the system being built by P.A.T. and the other system with which *FotoFile* was included. The quality of the components had been assured through testing, and, therefore, did not need further consideration when building the full system. Testing time was not reduced for the components built, but, when testing the full system, the testing strategies employed needed only to be concerned with the database section of the system. This also helped in identifying where errors were occurring when interfacing the system with the components, as only the component interfaces needed to be tested.

4. Ease of maintenance

As with testing, maintenance has been simplified because the system has been broken down into smaller components. When a change request is received from a customer, it is easy to identify whether the change will affect the overall database system, the *FotoFile* or the *Badge Server*. The appropriate component can then be updated. The interfaces are generally not affected by such changes, therefore, no side effects can be propagated to the other parts of the system. This contrasts a great deal with the earlier version of the system, which was monolithic and maintenance was a full time task for the software developers.

Maintenance has also been assisted by the tools supporting automatic generation of software documentation. It was seen near the end of the reuse programme that, when one of the developers left, he was asked to spend a few days writing a maintenance document for the software that he had written. However, if the code had been properly commented, this document could have been generated automatically in a matter of seconds.

7.2.3 Problems facing Reuse Programme

Some of the major problems which faced the incremental reuse programme are discussed below.

1. Tight deadlines

One of the major difficulties which faced the reuse programme were the tight deadlines which had to be met by the software developers. It has been recognised that small companies are unique in their need to keep up with market trends, and succeed in every project that they undertake. Experimentation and prototyping are key to their success, because they help the developers to understand how systems can be implemented, and what their customers really want.

P.A.T.'s business is dependent upon a single product. If that product failed, then the company would cease to exist. It is, therefore, in the interests of the developers to ensure that their product succeeds. To do this, the product needs to be shown to be competitive in the marketplace. In the project undertaken, the developer's deadlines were demonstrations to potential and existing customers or trade fairs, the dates of which often cannot be changed. In order to meet these deadlines, the reuse guidelines and up-front investment recommended as part of the reuse plan were often ignored in favour of rapid prototyping as the deadline drew closer. However, the developers were prepared to improve their code based on the recommendations of the reuse programme when they had more time, and the pressure had subsided.

2. Changing Requirements

This is not an uncommon problem throughout software development companies. However, because this company are producing a software package rather than a bespoke system, there are many customers, each with different requirements from the product. This was one of the factors that had caused the monolithic growth of P.A.T.'s previous software system - each change request had simply been added to the full system. Better version control would have helped to alleviate this problem. Requests from customers also affected the reuse programme, as each new requirement for the new system would slightly alter the system profile. Sometimes, this would affect the reusable components which were being built, meaning that the original plans for them had to be modified. However, one of the advantages of the component based system was that a change in one component seldom had a radical effect on other parts of the system.

3. Lack of Tool Support

The 'lightweight' processes recommended in section 6.3 suggested that a minimum level of documentation and a reuse repository should be kept as part of the reuse programme. Again, it is a common problem that when the pressure is on, documentation is the first casualty. The staff at

P.A.T. had not been used to writing documentation, and the reuse programme recommendations didn't really change anything.

Also, there was no support for their reuse repository. The development team simply made a directory as a 'dumping ground' for reusable classes without any support for using those classes. There was only some acceptance of the ideas of using a structured OO method for software design. This was partly due to lack of training in this area. However, it was felt that tool support could help automate the first two areas, and assist in the third.

7.3 Tool set Evaluation

This section evaluates ReThree-C++, the prototype tool set developed as part of this research. The evaluation will be given in four sections:

1. Using the tool set to support reuse.
2. An evaluation of the operation of tool set.
3. An independent experiment conducted to test the usefulness of the tool set in reusing classes during software development.
4. The tool set as used in Public Access Terminals Ltd.

7.3.1 Using the tool set to support reuse

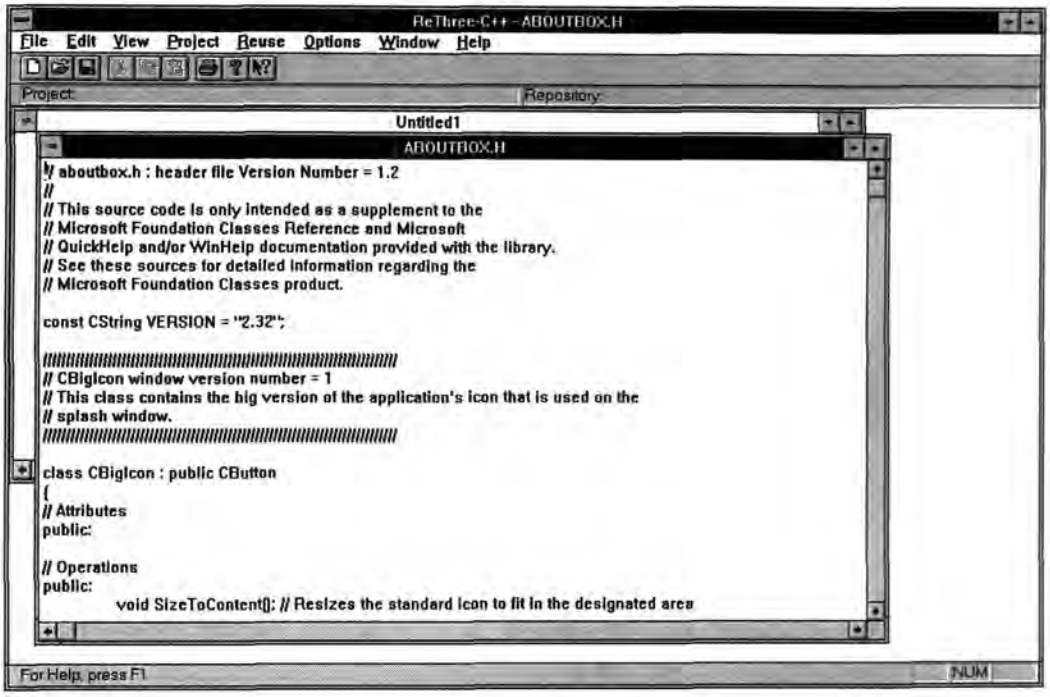


Figure 7.1 - The ReThree-C++ user interface

The integrated Reverse Engineering, Re-Documentation and Reuse environment, ReThree-C++, can be used in several aspects of software development. It has three major modes of operation.

Developing Components

The first mode of operation is when a developer is building reusable components. It is expected that the developer will build software using their chosen development environment. However, when assessing the component's applicability for reuse, the reverse engineering and documentation facilities can give the developer information about the component. The developer can also see how the component will appear to anyone who wishes to reuse it. Based on this information, the developer may decide that the component needs further development and take it back to his or her development environment. The developer may decide that the component is well designed, but requires more comments to explain how the component is to be used. This can be done with the text editing facilities available in the ReThree-C++ environment. Finally, when the developer is satisfied with the component's quality, it can be added to the current reuse repository.

Maintaining Components

The second mode of operation is when a maintainer is trying to understand and maintain a piece of software. It has been estimated that 50%-90% of all maintenance effort is expended in simply understanding the software [Robs91]. The processing facilities of ReThree-C++ will give the maintainer information about the source code, including a class hierarchy chart and documentation for the code. This information should help the maintainer to get an good idea of the purpose of the component. Based on this understanding, the maintainer can now look at the source code itself with a good idea of what to expect. One advantage of basing maintenance information solely on what is contained in the source code is that it helps to alleviate the problems caused by out of date documentation. This depends on the developers keeping current information in the source code about changes that have been made. Although this will not always be the case, developers are far more willing to update comments whilst changing the code than they are to update documentation after the changes have been completed.

Reusing Components

The third mode of operation is when a developer is searching for a reusable component to include in their current system. It has been seen in section 2.5 that there are numerous pre-conditions which must be met in order for a developer to be able to successfully reuse a component. These pre-conditions are listed below, along with the support which ReThree-C++ provides at each level.

1. **The component must exist.**

   ReThree-C++ provides support to developers when preparing reusable components.

2. **The component must be available to the developer.**

   ReThree-C++ enables developers to store components in a reuse repository.

3. **The developer must be able to find the component.**

   ReThree-C++ offers searching facilities for finding components with a reuse repository.

4. **Once found, the developer must be able to understand the component.**

   ReThree-C++ processes source code to give developers information about the components identified in terms of a class hierarchy and structured software documentation.

5. **Based on an understanding of the component, the developer must identify the component as being valid for the current system.**

   The developer can use the information generated by ReThree-C++ to make this decision.

6. **The developer must be able to successfully integrate the component into the current system.**

   This depends a great deal on the developer's current system. However, if the component has been developed properly, the class hierarchy and documentation provided should aid the developer in the integration process.

Specific examples of the use of ReThree-C++ to process a C++ source file are provided in Appendix B.

7.3.2 Evaluation of the operation of ReThree-C++

This section presents results from the ReThree-C++ tool set, applying the tools to various example programs, ranging from simple examples to real world class libraries. The tool set will be evaluated based on the following criteria:

- Does the integrated approach result in a usable system? This will consider the tool set's user interface for ease of use and how much training is required to use the tool set.
- How well does the tool set work on C++ code? This will consider such issues as speed, efficiency, reliability, and quality of results.
- How does the tool set scale up to larger programs? Does the system remain 'fast enough' to be usable with large programs?
- How useful are the searching facilities for reuse repositories?
- What weaknesses does the tool set have?

Usability

The tool set was built using a standard Microsoft® Windows® interface, which gives it a recognisable Graphical User Interface (GUI) for working in the Windows® 3.1 or 95 environment (see figure 7.1). The tool set comes with on-line, context sensitive help to assist the user in understanding how to use the tools. The usability of ReThree-C++ has been measured by applying the tool set in two areas.

The first is delivery of the tool set to staff at Public Access Terminals Ltd. (see section 7.3.4). The staff felt that the tool set was quite easy set up for use, and they learned how to use it very quickly. They felt that the help file was useful in learning how to use the tool set, and referred to it frequently (see Appendix C). Little training was given to the staff, they had only seen a demonstration of the tool set.

The second is the use of the tool set by undergraduates as part of the C++ reuse experiment conducted (see section 7.3.3). Some of the students were using the tool set to search for reusable classes which would assist them in writing the test program given. The students were given an overview of how to use the tool set (Appendix D4), and were left to write the program. Without training, all the students were successfully using the tool set to search for components within the hour allotted for the experiment. The students had few problems in using the tool set to find reusable classes, and did not need to use the help file.

Speed of operation

ReThree-C++ was tested on several different sizes of program to identify the speed of operation of the program. The time taken to execute the different tools which make up the integrated environment was measured and recorded. The results are shown in figure 7.2.

ReThree-C++ Full Source - 19 Files, 31 Classes, 1831 Lines of Code		
Type of Processing Performed	**Average Time (in seconds)**	**Standard Deviation**
Reverse Engineering to OMTool Format	5.61	0.153
Documentation to Rich Text Format	9.08	0.524
Class Hierarchy and Documentation to Web Page	8.76	0.081
Adding to Reuse Repository	3.04	0.349
Searching Reuse Repository	0.48	0.117

MFC Partial Source - 3 Files, 64 Classes, 4241 Lines of Code		
Type of Processing Performed	**Average Time (in seconds)**	**Standard Deviation**
Reverse Engineering to OMTool Format	24.61	1.899
Documentation to Rich Text Format	37.45	1.843
Class Hierarchy and Documentation to Web Page	33.72	1.035
Adding to Reuse Repository	10.72	1.677
Searching Reuse Repository	2.37	0.141

MFC Partial Source - 4 Files, 114 Classes, 7468 Lines of Code		
Type of Processing Performed	**Average Time (in seconds)**	**Standard Deviation**
Reverse Engineering to OMTool Format	46.20	5.098
Documentation to Rich Text Format	74.75	5.456
Class Hierarchy and Documentation to Web Page	68.23	4.175
Adding to Reuse Repository	18.82	2.539
Searching Reuse Repository	3.19	0.321

Single File - 6 Classes, 183 Lines of Code		
Type of Processing Performed	**Average Time (in seconds)**	**Standard Deviation**
Reverse Engineering to OMTool Format	1.06	0.048
Documentation to Rich Text Format	1.59	0.024
Class Hierarchy and Documentation to Web Page	1.77	0.024
Adding to Reuse Repository	0.49	0.052
Searching Reuse Repository	Negligible	Negligible

MFC Full Source - 20 Files, 169 Classes, 12984 Lines of Code		
Type of Processing Performed	**Average Time (in seconds)**	**Standard Deviation**
Reverse Engineering to OMTool Format	64.40	6.350
Documentation to Rich Text Format	96.21	7.263
Class Hierarchy and Documentation to Web Page	88.74	5.790
Adding to Reuse Repository	35.02	10.496
Searching Reuse Repository	3.88	0.303

Figure 7.2 - Results of evaluation of the speed of execution of ReThree-C++

It can be seen that the reuse repository support tool is the fastest of the tools, followed by the reverse engineering tool, then the documentation tool. It was expected that the generation of Web pages, which includes information from both reverse engineering and documentation of the source code, would be the slowest of the tools. This has been demonstrated in practice. During testing, it was noticed when the tools were run on large software systems, the prototype would run progressively slower each time the system was processed. As this was seen with all the tools, it was suspected that this problem was caused by the C++ parser, which is common throughout the tool set. The problem may have been caused by inadequate garbage collection in the parser.

The next section shows how these results were used to test the scalability of the prototype tool set.

Scalability

To test the scalability of ReThree-C++, the relationship between the number of classes being processed and the time taken to process those classes was measured.

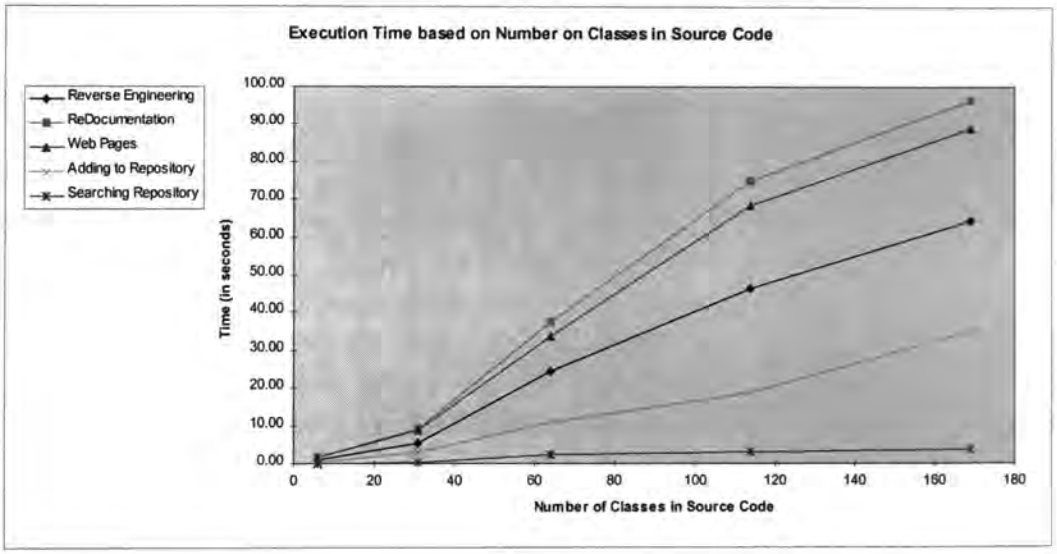


Figure 7.3 - Graph showing the results of evaluation of the speed of execution of ReThree-C++

It can be seen from figure 7.3 that the relationship between the increase of execution time and the number of classes in the source code is approximately linear. With the cases used, the relationship between the number of classes and the size of the source code was also approximately linear, so using either as a measure did not affect the linear nature of results shown.

Searching Repository

ReThree-C++ uses boolean keyword searching when identifying classes relevant to the user's specified search term. It is very difficult to conclusively evaluate the searching facilities which ReThree-C++ provides. This is because the indexing information used to search for relevant classes is taken directly from the comments in the C++ source code. This means that when meaningful class names and descriptive comments are provided in the code, the results of searching the reuse repository are significantly better than when comments are not provided, or, worse still, misleading. It has been seen that this style of information abstraction from the source code is useful in encouraging developers to include meaningful comments within their code. This is because the comments that they include will directly affect the usefulness of the information provided for them by the tool set (see Appendix C).

Weaknesses of the tool set

Several weaknesses have been discovered within the prototype tool set as it has been evaluated.

1. The processing of the tool set gets considerably slower each time that a large project is processed during a single execution of the program. This compounds to the extent that the tool set eventually grinds to a halt. It is suspected that this is due to poor deallocation of memory resources used within the parser of the ReThree-C++ system, as the effect is seen no matter what type of processing is currently being conducted.
2. The reuse repository searching facilities are not always effective. As discussed in the previous section, this could be attributed to the lack of meaningful comments within the source code.
3. The tool set relies on an outdated application. OMTool has now been superseded by other, better OMT CASE tools, and is no longer easily available. This is always a hazard in interfacing with other applications. OMTool has not been upgraded for Windows® 95, and only those who already have it would be able to interface the tool set with this application. The reverse engineering facilities to OMT class hierarchies are, therefore, only available to a small subset of users. However, interfacing to a new display tool would not entail significant effort.

7.3.3 An experiment to test the use of ReThree-C++

This section gives an overview of an experiment conducted to test the usefulness of ReThree-C++ in assisting developers to find and use reusable components. This experiment was conducted independently of the case study associated with this research to test the usefulness of the tool set to C++ developers. There are several steps [Pfle95] which were followed in the implementation of this experiment, which will be discussed.

Conception

The experiment was conceived to test the usefulness of the facilities provided by the prototype tool set in helping a developer to reuse available components. The idea was to get several different C++ developers writing the same program in order to see how the type of information that was presented to them concerning available reusable classes affected the way in which they wrote the program.

Design

C++ programmers would be the subjects of the experiment. The experiment had two hypotheses:

Null hypothesis: There is no difference in the code produced by programmers based on the amount of information provided to the programmers about reusable components.

Alternative hypothesis: The amount of information provided to programmers about reusable components makes a difference to the code they produce.

In order to test these hypotheses, an experiment was devised in which C++ programmers would write a program, each set of programmers having differing amounts of information about reusable classes which were available. A program was prepared to test two major areas of C++ programming - file handling and string manipulation (see Appendix D2). Visual C++ was chosen as the programming environment, as classes to assist in writing this program were available in the Microsoft® Foundation Classes (MFC). It is expected that programmers who have more detailed information available about reusable classes will make use of those classes. It is also expected that the use of the reusable classes will make a difference in the time taken to write the program.

Four groups of programmers would be identified. Group 1 would have no information about the reusable classes available - they would only have the C++ programming environment (including on-line help), a C++ reference manual, and a C library reference manual. Group 2 would have

Execution

Many of the subjects of the experiment had not used Visual C++ before. However, the environment was set up so that they could use both standard C and C++ and the Microsoft® Foundation Classes within the same program. Each group was given an hour to complete a working version of the program. Details of the references used by each of the subjects was recorded as the experiment progressed. The researcher was careful to ensure that none of the students knew what the experiment was about until after their contribution had been completed.

The results gathered during the execution of the experiment can be seen in figure 7.4.

the same information as group 1, and would also have the Class Library Reference manual for the MFC (a 1000 page reference manual containing details of all the classes available in the MFC). Group 3 would have the same information as group 2, as well as the results of re-documenting the appropriate MFC classes with ReThree-C++. Group 4 would have the same information as group 2, as well as ReThree-C++ running on their machines, with the source of the MFC pre-loaded as a reuse repository for searching.

It was expected that group 3 would achieve the best results in writing the program, as they had the information about the required reusable classes on paper as part of their reference materials. This was based on the fact that the subjects in group 3 would have the relevant information on printed paper in front of them, and would not have to spend time searching for it in reference manuals, or using the tool set.

Preparation

The subjects chosen for the experiment were final year Computer Science undergraduates, each of whom had been through the previous year's course on OOD and C++. The students volunteered to take part in the experiment, and were asked to qualify their skill at programming in C++. The students were then divided into four mixed ability groups. Different instructions were prepared for each group (see Appendix D1). The information for each group was also prepared. The reference books used were "Software Engineering with C++ and CASE Tools" by Michael J. Pont [Pont96], Visual C++'s "Run-Time Library Reference" [Micr93d] and Visual C++'s "Class Library Reference for the Microsoft Foundation Class Library" [Micr93c]. The other materials required were also prepared, including the class information generated by ReThree-C++ for group 3 (see Appendix D3), and instructions on the use of ReThree-C++ for group 4 (see Appendix D4).

Group	Pre-test C++ Skills rating	Post-test C++ Skills rating	Used C Reference	Used C++ Reference	Used MFC Reference	Used Tool Results	Used Tool	Used Help	Used standard lib	Used MFC Classes	Used own classes	Does program work	Error checking	Size of source code (LOC)	Preparation time (minutes)	Start time	Finish Time	Estimated extra time needed	Total Estimated Time	Usefulness of reference	Difficulty of writing program
1																					
A	7	7	S	U				N	2	0	Y	2	5	47	5	14:16	15:16	60	120	7	8
B	6	4	S	U				C	7	0	Y	0	8	130	180	15:20	16:20	60-75	120-135	7	5
C	6	5	S	U				C	10	0	Y	2	8	149	0	14:22	15:42	15	95	5	4
D	4	5	R	U				C	10	0	N	0	7	66	0	15:20	16:20	20-30	80-90	5	6
E	3	3	S	U				N	1	0	N	0	3	56	0	14:10	15:11	60	120	5	8
2																					
F	7	7	S	U	N			C	4	0	Y	C	0	66	30	10:55	11:57	60	122	6	5
G	6	6	U	R	S			C	4	0	Y	N	2	89	0	11:03	12:02	120	179	3	3
H																					
I	3	3	U	U	R			C	5	0	N	C	0	31	0	14:22	15:24	120	182	3	9
J																					
3																					
K	7	7	N	N	N	U		MFC	0	10	N	3	9	74	0	11:05	12:05	60	120	5	3
L	6	5	N	N	U	U		N	0	10	N	0	7	100	0	11:05	12:05	30	90	8	8
M	6	4	N	N	N	U		N	4	6	N	0	4	65	10	11:05	12:05	30	90	5	9
N	3	3	N	N	N	U		C/MFC	0	10	N	0	7	57	0	11:05	12:05	60	120	6	9
O	3	4	U	N	N	R		N	6	0	N	0	0	53	10	11:05	12:05	120-180	180-240	6	7
4																					
P																					
Q	6	6	N	N	U	U	U	N	5	5	N	0	8	71	5	12:25	13:25	30	90	7	5
R	5	5	N	U	N	U	U	N	5	5	N	4	0	37	10	12:20	13:20	30	90	9	8
S																					
T	1	1	N	U	U	U	U	N	0	10	N	0	0	29	0	12:25	13:25	Day	Day	8	10

Key

Skills Rating

1-10 Subject's rating of C++ skill (10 = excellent, 1 = poor)

Resources

N Resource Not Used
S Resource Scanned for Information
R Resource Read
U Resource Used
C C/C++ Language Help
MFC Microsoft Foundation Classes Help

Program

Resources Used

0-10 Rating scale measuring extent to resource was used (10 = used exclusively, 0 = not used)

Own Classes

Y/N Y = Yes, subject wrote their own classes, N = No, subject did not write their own classes

Program Working

0-10, C Rating scale measuring extent to which program worked (10 = works as specified, 0 = does not compile, C = compiles, but gives no output)

Error Checking

0-10 Rating scale measuring extent to which error checking is employed (10 = excellent error checking, 0 = no error checking)

Feedback

Usefulness of reference

1-10 Subject's rating of usefulness of reference materials provided (10 = very useful, 1 = no use)

Figure 7.4 - Table of Results from C++ experiment

The first point to note is that none of the subjects actually finished the program specified. This is a reflection that the program was too difficult for the allotted time. In order to gauge the subject's feelings on how they were progressing with the program, each was asked to estimate how much more time would have been needed for them to finish the program.

Unfortunately, not all the subjects who had signed up actually took part in the experiment. This makes it difficult to compare results between the groups, as there was not a uniform number of subjects in each group. Some subjects also expressed difficulty simply with remembering how to write code in C++, rather than difficulty in how to write this particular program. These two factors affect the overall discussion of the results of the experiment. Perhaps the most useful way to evaluate the results is to compare the two most skilled subjects from each group, and the least skilled subject from each group.

In group 1, the two most skilled subjects (A&B) wrote their own classes to assist them in achieving the functionality requested by the program specification. Their mean estimated completion time was 127.5 minutes, with a standard deviation of 7.5 minutes. One of the two achieved a small measure of functionality.

The two most skilled subjects in group 2 (F&G) again both wrote their own classes to assist them in achieving the functionality requested by the program specification. Their mean estimated completion time was 150.5 minutes, with no standard deviation. Neither achieved a reasonable level of functionality.

The two most skilled subjects in group 3 (K&L) did not write their own classes to assist them in achieving the functionality requested by the program specification. Instead, they both used classes available as part of the MFC. Their mean estimated completion time was 105 minutes, with no standard deviation. One of the two achieved a small measure of functionality.

The two most skilled subjects in group 4 (Q&R) did not write their own classes to assist them in achieving the functionality requested by the program specification. Instead, they both used classes available as part of the MFC. Their mean estimated completion time was 90 minutes, with no standard deviation. One of the two achieved a reasonable measure of functionality.

Perhaps the most interesting result is the use of classes made by these subjects. There was no significant difference between the two subjects under consideration from groups 1 & 2, in spite of the fact that the subjects in group 2 had access to the MFC reference book containing details of the reusable classes available. The subjects in both groups wrote their own classes.

By comparison, none of the subjects in groups 3 & 4 wrote their own classes, preferring instead to use the reusable classes provided by the MFC. This is a significant result. Groups 2, 3 and 4 each had the same reference books available to them. Therefore, it must have been the other reference materials which caused the subjects in groups 3 & 4 to choose to reuse classes rather than writing their own. As the other reference materials were directly produced by ReThree-C++, it can be concluded that ReThree-C++ assisted these programmers to reuse classes.

Less significant is the time taken by the subjects in writing the program, because none of the subject actually completed a working version of the program. Based on the mean times calculated, it can be seen that the subjects in groups 3 & 4 were more confident that they could finish writing the program in a shorter time scale. Without further experimentation, it cannot be conclusively shown that reusing classes increased the productivity of the programmers. It was surprising, however, that the group 3 subjects (with the printed results of ReThree-C++) did not seem to do any better than the subjects in group 4 (who actually used the tool set). This may be due to the fact that the subjects in group 3 spent more time reading all the class reference materials provided (not all of which were directly relevant), whereas the group 4 subjects used the tool set to search for classes only when they encountered a need for a class to perform a function. This may have saved them time.

Looking now at the least skilled programmers, it can be seen that there is no significant difference in the reuse of classes between the subjects (E, I, O & T). Both subjects O & T expressed a concern at their difficulty in simply writing any program in C++, not just this one. None of these subjects wrote their own classes in attempting a solution, and only one (subject T)

attempted to reuse the MFC classes. This seems to suggest that with inexperienced programmers, the level of information provided made very little difference.

The small scale of this experiment provides interesting results, but the wide variance of C++ programming knowledge and experience obviously plays a considerable role in the results.

7.3.4 Evaluation of the use of ReThree-C++ at P.A.T.

In line with the 'industry-as-laboratory' approach adopted by this research, the prototype tool set was developed in association with the staff at Public Access Terminals Ltd. This has enabled them to make suggestions about how they would like to see the tools developed to assist them in their work.

The tools were made available to the company throughout the reuse programme as they were developed. This enabled the company to incrementally introduce the tools into the programme as needed. However, the integrated environment was not available until nearing the end of the programme.

One of the major problems which has been experienced in evaluating the use of the tool set within the company is that the two C++ developers who were key members of the reuse project left the company before the full tool set was developed. Following this, the company began to use different programming languages to build their software system, which meant that the prototype tool set, when delivered, was less applicable to the company's current development needs.

The staff at the company, however, have found that the tool set meets some of their needs very well. A questionnaire about the prototype was filled in by the company's technical manager (Appendix C). He said that the tool was easy to set up and use. He felt that the automatic generation of documentation was the most valuable tool for his work, and that the

documentation produced was quite useful in helping him to understand the C++ code processed. He also felt that the fact that documentation was based solely on the source code, and the comments within the code, would encourage programmers to tidy up their code and add comments. He also felt that having this documentation produced automatically would be very useful in keeping a minimum set of documentation about the code for the programmers and also the customers interested in information on, and quality assurance for, the software system.

He also felt that the reuse repository facilities were useful, but that it was not always easy to find appropriate classes using the repository searching facilities. He felt that the processing facilities of the tool set were reasonably helpful in understanding the classes once found, but would have been more helpful if there were better comments within the source code.

Although he was not able to use the prototype on a live system, he used it on previously written C++ code, and hoped to be able to use it on code ported to a 32 bit platform in the near future. He thought that the tool set will prove to be very useful.

7.4 Conclusions

The research described in this thesis has been conducted in two main sections. Each section has been evaluated in this chapter.

The first was the incremental introduction of a reuse programme at Public Access Terminals Ltd. It was noted that the programme had some success, but problems were encountered. The successes were identified in terms of: increased speed of production, financial benefits to the company, increased quality of software, and ease of maintenance. Problems were identified in terms of: tight deadlines, changing requirements and lack of tool support. However, it can be concluded that the benefits of the reuse programme far outweighed the problems and challenges faced. It could be considered that much of this success can be attributed to the new technology which the company adopted. New technology will only provide a platform for making

improvements. It is only when the opportunities provided by this technology can be identified and exploited that benefits will be gained.

The second section was the development of a prototype tool set (ReThree-C++) to assist programmers by automating reuse support in a small company. The tool set was evaluated by staff at Public Access Terminals Ltd. It was also evaluated as part of an experiment testing the difference that varying levels of information about reusable classes made to programmers when writing a program. Results from the operation of the tool were presented. Further discussion of these results will be conducted in the next chapter.

Chapter 8: Conclusions

8.1 Introduction

This chapter summarises the thesis and reviews the work that has been conducted during this research. It also considers the results which have been achieved using the proposed method for incrementally implementing reuse in a small company in association with the prototype tool set which has been developed. These are assessed against the original goals of the thesis, which were:

1. To show a real case study of the implementation of a software reuse programme in a small company. The programme will be considered in terms of the recommendations made, the work done, problems encountered and success achieved.
2. To produce a practical, fast and simple to use tool for automating reuse support in a small company. This tool will aid in storing and retrieving reusable components, as well as reverse engineering and re-documenting source code to provide information about the reusable components.

The research conducted is analysed to identify the lessons which have been learned, and to make recommendations for further work in this area of study.

8.2 Summary of Thesis

This thesis has evaluated the practical considerations involved in automating reuse support in a small company. Chapter 1 gave an overview of the thesis, introducing the research which would be described, along with a statement of the problem to be addressed. The real problem of introducing reuse in a small company, and providing tools to support the reuse process, was

identified. The context in which the research would be conducted was given and an 'industry-as-laboratory' approach was adopted.

Chapter 2 looked at the field of software reuse, identifying some of the key areas in the field. Software reuse was introduced as a principle which could help to alleviate the current software crisis and the techniques with which reuse can be employed were discussed. Some of the benefits that can come from introducing a reuse programme were identified, as well as the challenges which will face a company trying to capitalise on the benefits which reuse can bring.

Chapter 3 went on to look at how a small company is defined and some of the techniques which will assist a small company when implementing a reuse programme. The fields of organisational development and process improvement were studied in order to provide a basis for developing a method for introducing reuse into a small company. Object-oriented methods, which are often associated with software reuse, were also considered. It was concluded that the introduction of object-oriented methods could help to support reuse, but that reuse is not exclusively an OO phenomenon. This was followed by a brief overview of the fields of reverse engineering and software documentation, and how they can be applied to reuse.

In Chapter 4, several successful reuse programmes were considered. It was seen that reported reuse programmes were exclusively in large companies, and that the challenges which they faced in introducing reuse were often different to those that would be faced in a small company. A set of solutions to the problem of introducing reuse into a small company were identified. These solutions were:

1. Introduction of structured processes
2. Incremental introduction of reuse
3. Encouraging ad-hoc reuse
4. Introduction of CASE tools

Each was discussed, and it was argued that a combination of the incremental introduction of reuse with CASE tools to support the reuse programme would be the best approach for a small company.

Chapter 5 summarised the method for introducing reuse in a small company which has been developed as part of this research. An incremental approach was stressed, along with 'lightweight' processes and automated support for the reuse programme. These factors would help to reduce the risk in introducing reuse by reducing both the initial investment required and the time before benefits could be gained from reuse. The *Seven Steps to Success* were presented, including a pilot project to test the recommended techniques so that the company could learn what would be most successful for them and focus on those areas. At each step, criteria for assessing the readiness of a company to move on to the next step of the method were given. Ideas for tools to support the reuse programme were also presented.

Based on the method developed in the previous chapter, Chapter 6 described a case study using the method to implement a reuse programme in a small company. The work done in each of the seven steps was presented, with a discussion of the progress of the reuse plan at each stage. The development of ReThree-C++ was described, with the input of the company's staff aiding the structure of the prototype tool set.

Chapter 7 gave an analysis of the results of the research conducted. The incremental approach to reuse introduction in a small company was evaluated and the more and less successful parts of the programme were identified. Success was described in terms of the development and use of reusable modules. Benefits to the company were described as well as the problems facing the reuse programme. These were the pressure of tight deadlines, changing requirements and lack of tool support. ReThree-C++, the prototype tool set, was first described then evaluated in three stages. First, the prototype was evaluated using the code of the tool set itself and the Microsoft Foundation Classes as test examples. Secondly, an experiment to test the usefulness of the prototype tool set in aiding developers to reuse components was conducted and the results of the

experiment in helping programmers to reuse code were analysed. Finally, an overview of its use within P.A.T. was given.

8.3 Reuse in a Small Company Revisited

The results gained from the case study have been varied. There have been some successes, but the challenges and difficulties encountered during the course of the project have also been interesting. Small companies are unique in their need to compete strongly within their chosen market, and succeed in every project that they undertake. Unlike larger companies, and even single project teams within a large organisation, a small company cannot afford to fail in any project, because the livelihood of the company, and every employee, depends upon keeping and improving upon their market share. In the company with which this project was associated, their business was dependent upon a single product. If that product failed, then the company would cease to exist. This does not compare with even isolated parts of a large company, because although the project may fail and cause difficulty within the company, this would not generally cause the collapse of the business. The stakes are much higher in a small company, and their willingness to take unexplored risks is much smaller.

Based on the evaluation of the reuse programme described in Chapter 7, there are several conclusions which can be drawn from the incremental implementation of a reuse programme conducted in the case study with Public Access Terminals Ltd. In spite of the risks that they faced, the company's staff were willing to attempt a reuse programme in order to gain the benefits of reuse. The first conclusion is that the incremental approach to reuse was very successful in the company. The method for introducing reuse into a small company, based on the work done in the fields of organisation development and process improvement, proved to be very successful. At each stage of the method, the progress of the method was discussed with the management and staff at the company and the criteria for continuation were met. The key areas of using a pilot project to achieve real gains for the company whilst testing the 'lightweight'

processes and incrementally introducing the programme with tool support were invaluable in the success of the reuse programme.

This conclusion is based on the benefits gained by the company described in section 7.2.2. The company developed a better, more flexible system faster than expected. They also benefited financially by entering the component market and selling one of their reusable components to another company. This increase in profitability may never have been realised if not for the company's emphasis on reuse.

However, the benefits did not come without challenges. There were problems in the reuse programme which had to be addressed. The least successful of the 'lightweight' processes recommended was that the company keep a minimum level of documentation about their software. Documentation was always the first casualty when the pressure was on. This problem was recognised and was addressed by allowing developers to utilise the information which they had included as part of the source code (in the form of descriptive comments) as software documentation using the prototype tool set described in Chapters 6 and 7. This provides a feasible, convenient and easy way for the developers to keep a minimum level of documentation. This has been seen by the staff at P.A.T. as one of the major advantages of the prototype tool set, as seen in section 7.3.4.

The planning and review meetings were successful initially. Although they started formally on a weekly basis, later in the programme, they were often on an informal basis as there were only a few people in the software development team who needed to meet at any one stage. The emphasis on object-oriented principles and resource management were the more successful techniques in the company. However, it was the general emphasis on reuse which came with the techniques, supported by management, development staff and the author, which made the projects successful. The developers were highly skilled, and needed little training to understand the principles of reuse. More important were the motivation and opportunity to implement the principles with reuse as a clear goal. These came from the willingness of top level management

to be involved in the reuse programme and the expertise of the author acting as a reuse consultant.

After the successes which have been described, however, the reuse programme has taken a back seat in the company. This is due to many factors, not least of which is that the two members of staff who were the key developers in the reuse programme have left the company. Along with this, the company have since moved to different development languages and environments. However, it has been seen that the principles advocated as part of the reuse programme have given the company the flexibility to move to better environments. The prototype tool set which has been developed to support the reuse programme will still be of use to the company in both their maintenance work, and with proposed new developments (see Appendix C). This will enable the company to produce useful information about their previous software, as well as giving them guidance for future developments.

To summarise, the challenges to introducing reuse in a small company have been met and overcome. The company have climbed the *Seven Steps to Success* and were clearly pleased with the very tangible benefits that they have gained from the reuse programme.

8.4 ReThree-C++ - The Prototype Tool Set

It was shown that ReThree-C++ is a practical and useful prototype of an integrated tool set which can automate reuse support in a small company. It addresses one of the key failings of the reuse programme - lack of software documentation to describe reusable components - by automatically generating useful information from the company's source code. This information is given as a class hierarchy and associated documentation, which can easily interface with standard desktop software packages. It also provides support when indexing and searching for reusable components.

The validity of the prototype has been demonstrated through experimentation and analysis. Unfortunately, the prototype has not, as yet, been used in a live development environment as part of the case study. This is because the company have now moved away from C++ development. However, the staff are keen to gain the benefits which use of the prototype can bring in both the maintenance of their previously written code, and when embarking on new developments (see Appendix C).

The experiment conducted also demonstrated very well that ReThree-C++ supports developers and helps them to locate and use reusable components when building a software system.

The ReThree-C++ system is a step forward in automating reuse support for a small company. There are CASE tools which support reverse engineering to OO formats, and software documentation generators supporting both word processors and Web browsers. There are also tools which support reuse libraries, allowing for the indexing and retrieval of reusable components. Although some work is beginning to be done in this area of tool integration [Zigm95], there are still no tools available to small companies which integrate these concepts, supporting development throughout the reuse programme. ReThree-C++ was shown to do this (see section 7.3).

With their limited resources, both in terms of time and money, a small company could not afford to introduce a large CASE environment into their software development practices. They also could not afford the effort required to integrate a set of smaller tools. The ReThree-C++ tool set integrates the tools identified in chapter 6 as being important in supporting software reuse.

Two of the key failings of the prototype are its problems with the repeated processing of large systems and its reliance on an outdated tool to display some of its results. With further work, these failings could be overcome, and the prototype made into a valuable production system. It is fast enough not to be cumbersome, and easy enough to use that little training is required. This has been shown both in the case study and the experiment conducted.

The goals of the prototype tool set were that it should be a practical, fast and simple to use set of tools for automating reuse support in a small company. This tool set was to aid in storing and retrieving reusable components, as well as reverse engineering and re-documenting source code to provide information about the reusable components. These goals have been met.

8.5 Analysis of the research

This section of the thesis compares the work done with other work in the field, and considers the lessons that can be learned from this research.

As the interest in reuse has grown, more and more companies have attempted to implement reuse programmes with varying results. Successful examples are being quoted to show that implementing reuse is possible, and great benefits can be gained from it. However, it seems that more publications are now concentrating on the organisational difficulties of implementing a successful reuse programme rather than the technical issues considered previously.

Books by McClure [McCl97], Jacobson [Jaco97] and Leach [Leac97] all suggest methods and techniques which can be applied to reuse, quoting examples of successful companies which have applied the principles. Many companies described in these publications have started to recognise the advantages of adopting an incremental or evolutionary approach to reuse introduction. However, these books consider only the difficulties faced by large companies. There are still no reuse programmes in small companies discussed. Techniques are considered in terms of their applicability to the company's software processes and the changes which would be applied to those processes for successful reuse. Some tools are discussed with their applicability for reuse, but these are mostly either reuse repository or OO tools.

The combination of 'lightweight' processes with an integrated tool set for reuse is unique to this research. The 'lightweight' processes are ideal for a company which currently has no software

processes. The tool set automates support for reuse and makes those processes easier to implement.

Specifically, there are seven lessons which can be learned from this research when implementing a reuse programme in a small company.

1) As has been seen in other research, the support of management and staff are vital. This must be reassessed at every step of a reuse programme. In a small company such as Public Access Terminals Ltd., the staff were very concerned with the challenges involved in implementing a reuse programme. More small company success stories would help encourage them to make the changes required for reuse to prosper.
2) Analysing the company's current working practices is an invaluable second step. If you are travelling from one point to another, you must know where you are starting from and where you are going in order to plan your route. Therefore, the company must also have an idea of what they want to achieve from the reuse programme.
3) Reuse will always require an investment before benefits can be gained. However, the use of 'lightweight' processes helps to reduce the impact of the changes to working practices which must be made. It is the flexible nature of these processes which make them so suitable for a company entering an evolutionary stage of development and that enables them to reduce the risk of failure. The recommended processes must relate to the company's current working practices, as well as the reuse programme itself.
4) A key to the success of small companies is the ability to be flexible. This enables them to meet their customers specific requirements. As was seen in the case study, reuse can aid the development of a flexible system. Planning is important in the progress of the reuse programme, but you must plan to be flexible. Reuse can support this type of flexibility.
5) Don't waste the developers' time by making them get involved with the mundane aspects of reuse. Automating support for reuse with a tool set which integrates repository control with automatic generation of software documentation reduces the time developers need to spend on administration. As has been seen at P.A.T., this encourages developers to write tidy, well

structured code, and leaves them free to concentrate on the more challenging and imaginative issues of developing for and with reuse.

6) Small companies do not have the resources to invest in the techniques often recommended for the implementation of corporate reuse programmes. They must be treated separately. The work ethic is different in a small company, and reuse strategies must recognise and incorporate this.

7) Make reuse available to everyone. The *Seven Steps to Success* described as part of the method for reuse introduction in this research are flexible enough to be applied to any company in any situation.

Assessing the method itself which was described in chapter 5, one of the key weaknesses of the method is that it is very generalised and does not go into great detail about any of the steps. This leaves a great deal of work to be done by a company using the method to make it specific to their own needs. However, this is also one of the method's key strengths, because it allows the method to be very generic, meaning that it could be used for any technology introduction or process improvement and is not strictly limited to reuse.

8.6 Further Work

The incremental approach to the introduction of a reuse programme has proved to be successful in the case study associated with this research. To support these results, it would be very valuable to conduct further case studies, with two objectives:

1. To provide further evidence that the incremental approach to reuse introduction allows companies to benefit from reuse both in the short term and in the long term.
2. To investigate whether the successes gained using 'lightweight' processes in an incremental approach to reuse introduction can be transferred to other areas of process introduction (and improvement) within both small and large companies.

One of the main areas which this research has not been able to address is the use of the prototype tool set in a live development environment. Another case study with a different small company, allowing the tool set to automate reuse support within the company from the start of the programme, would be very valuable in confirming the value of this approach.

There is also further work which could be done with the prototype tool set. Some areas of interest would be:

- Further development of the Java output from the tool set to support interactive class diagrams. The Java development language is progressing rapidly, and the possibilities for using this new language are increasing. A complete CASE tool for OO design and reverse engineering C++ code could be built in Java to support the processing of ReThree-C++.
- Further development of the documentation output offered by the prototype. The tool set currently supports RTF and HTML output, but there are other formats which could be considered (e.g. LaTeX).
- Support for other object-oriented languages. Since the prototype has been made publicly available, there has been interest in similar work for Ada 95 and Java. By incorporating different parsers, the same information could be generated for other languages.

8.7 Final Analysis

The criteria for success for this research as identified in Chapter 1 were given in terms of three questions:

1. Is the method for introducing a reuse programme successful? The success of a reuse programme can be measured in many ways. However, the most clearly identifiable measure of success is identifying whether reusable components are built, and to what extent they are reused.

2. Does the method bring benefits to a small company? As identified in section 2.4, benefits will be considered in terms of:
    - Increased speed of production
    - Financial benefits
    - Increased quality of software
    - Ease of maintenance
3. Does automated support aid a reuse programme? The automated support will be considered in terms of the benefits brought to a reuse programme and it's usefulness within a small company.

Answering the first question: Yes, as seen in chapter 6, the method developed was successfully applied at Public Access Terminals Ltd. At each stage of the method, the criteria for continuing were met, and real benefits were brought to the company as a result of implementing a reuse programme using the method. In the previous chapters, it has been seen that two main reusable components were built as part of the reuse programme introduced in the case study. Within the development of those components, a reuse factor of up to 70% was achieved by the developers.

The answer to question 2 has more interest to a company considering embarking on a reuse programme. The benefits which P.A.T. have gained from reuse are much more important than the amount of code reused. The flexibility provided by the reusable components built allowed the system to be developed using an application generator for databases, which saved the developers the time required to write their own database system. This considerably speeded up production of the company's new system. Selling one of the components brought a very large contract to P.A.T., bringing much needed financial gain midway through the reuse programme. As seen in the previous chapter, there were also benefits to the company in terms of the quality and ease of maintenance of their system.

The previous chapter discussed the success of the reuse tool set, which answers the third question. Its usefulness was shown both by its introduction at P.A.T. and by experimentation.

The experiment demonstrated that the tool set assisted programmers to develop with reuse when writing a program. Although the tool set was not used in a live development environment, when used on the company's previous and current developments, the company's technical manager believed it would be "very useful" (see Appendix C).

All three of these questions have been answered affirmatively. The case study conducted at Public Access Terminals Ltd. has provided interesting results in the field of reuse introduction in a small company which is low on the process maturity scale. It has been shown that a software reuse programme can be implemented in a small company. Although there were challenges, real benefits were achieved from the introduction of reuse within the company. Based on this case study, an incremental approach to software reuse using 'lightweight' processes, supported by useful and practical tools which can be easily integrated into a small company's development systems, is recommended for achieving success in a reuse programme. This addresses the organisational issues facing a reuse programme.

The prototype tool set has also been shown to be an effective method of automating support for the reuse programme. The integrated approach which the tool set adopts allows developers easy management of a software component repository, as well as automatically generating information about those components. These two factors help to solve the technological problems facing the successful introduction of a reuse programme.

These two solutions, when combined, offer a practical, manageable method for introducing reuse and gaining real benefits from reuse without the costly up-front investment often needed in order for reuse to succeed. The incremental approach to reuse introduction allows benefits gained from the earlier stages of reuse to fund the further investment needed to improve the reuse programme, and the prototype tool set aids the process by providing much needed automated support.

The method described in this thesis is based on previous work done in the fields of organisational development, process improvement and software reuse. As has been shown throughout the thesis, there have been several reported reuse successes in large, structured software development companies.

Although the work done with a small company must recognise the differences between the ethos and working practices of small and large companies, the overall structure of the method presented is based on the successful reuse case studies already reported.

The major differences appear in the way in which the reuse programme is introduced and supported. The incremental introduction of reuse is not unique, but the combination of the use of 'lightweight' processes with automated support for the reuse programme is. It has been seen that both of these additions to the method have been successful in implementing reuse in a small company.

Chapter 9: References

[Albr83] Albrecht, K.; 'Organisational Development: A Total Systems Approach to Positive Change in Any Business Organisation'; Prentice Hall; 1983

[Alle89] Allen, B.P., Lee, S.D.; 'A knowledge-based environment for the development of software parts composition systems'; In: Proc. of the 11th ICSE; Pittsburgh, PA; May 1989; P104-112

[Ande87] Andersson, T.D.; 'Profit in Small Firms"; Avebury; 1987

[Arth83] Arthur, L.J.; 'Programmer Productivity - Myths, Methods, and Murphy's Law'; John Wiley and Sons; 1983

[Atki91a] Atkins, M.C., Brown, A.W.; 'Principles of object-oriented systems'; In: Software Engineer's Reference Book; McDermid, J.A. (ed.); Butterworth-Heinemann Ltd.; 1991; P39/3-39/13

[Atki91b] Atkinson, C.; 'Object-Oriented reuse, concurrency and distribution : an Ada based approach'; ACM Press, Addison-Wesley, Reading, Mass.; 1991

[Babc90] Babcock, J.D., Belady, L.A., Gore, N.C.; 'The Evolution of Technology Transfer at MCC's Software Technology Program: From Didactic to Dialectic'; In: Proceedings of the 12th International Conference on Software Engineering; IEEE 1990; P290-299

[Babi86] Babisch, W.A.; 'Software Configuration Management - Coordination for Team Productivity'; Addison-Wesley, Reading, Mass.; 1986

[Bake89] Baker, B., Deeds, A.; 'Industrial Policy and Software Reuse: A Systems Approach'; In: Proc. of the Reuse in Practice Workshop; Baldo, J., Braun, C. (ed.); Software Engineering Institute, Pittsburgh, Penn; Jul 1989

[Basi87] Basili, V.R., Rombach, H.D., Bailey, J., Joo, B.G.; 'Software Reuse: A Framework'; In: Proc. of the Tenth Minnowbrook Workshop (1987, Software Reuse); Blue Mountain Lake, N.Y.; July 1987

[Bell92] Bell, D., Morrey, I., Pugh, J.; 'Software Engineering - A Programmer's Approach (2nd Ed.)'; Prentice Hall, New Jersey; 1992

[Benn93] Bennett, K.H.; 'An Overview of Maintenance and Reverse Engineering'; In: The REDO Compendium: Reverse Engineering for Software Maintenance; van Zuylen, H.J. (ed.); John Wiley and Sons; 1993

[Bigg87] Biggerstaff, T.J., Richter, C.; 'Reusability Framework, Assessment, and Directions'; IEEE Software; Jul 1987; Vol.4 No.4 P41-49

[Bigg89a] Biggerstaff, T.J., Perlis, A.J., (ed.); 'Software Reusability. Concepts and Models, vol. I'; ACM Press, Addison-Wesley, Reading, Mass.; 1989

[Bigg89b] Biggerstaff, T.J., Perlis, A.J., (ed.); 'Software Reusability. Applications and Experience, vol. II'; ACM Press, Addison-Wesley, Reading, Mass.; 1989

[Bigg93] Biggs, P.J.; 'Information Retrieval Applied to Software Documents (Including Source Code)'; Final Year Project Report, Dept. of Computer Science, University of Durham; 1993

[Bigg95] Biggs, P.J.; 'A Survey of Object-Oriented Methods'; Computer Science Technical Report 6/95; Dept. Of Computer Science, University of Durham; 1995

[Boll90] Bollinger, T.B., Pfleeger, S.L.; 'The Economics of Reuse: Issues and Alternatives'; In: Proc. of the Eighth Annual National Conference on Ada Technology, Atlanta; Mar 1990; P436-447

[Boll91] Bollinger, T.B., McGowan, C.; 'A Critical Look at Software Capability Evaluations'; IEEE Software; July 1991; Vol.8 No.4 P25-41

[Bott92] Bott, F., Ratcliffe, M.; 'Reuse and Design'; In: Software Reuse and Reverse Engineering in Practice; Hall, P.A.V. (ed.); Chapman & Hall, London; 1992

[Buck85] Buckley, M.W.; 'The Structure of Business'; Pitman Publishing Ltd, London; 1985

[Budd91] Budd, T., 'An Introduction to Object-Oriented Programming'; Addison-Wesley, Reading, Mass.; 1991

[Bull85] Bullock, R.J., Batten, D.; 'It's just a phase we're going through: a review and synthesis of OD phase analysis'; Group and Organisation Studies; Dec 1985; Vol.10 P383-412

[Bull94] Bull, T.; 'Software Maintenance by Program Transformation in a Wide Spectrum Language'; Ph.D. Thesis; Durham University; 1994

[Burd92] Burd, E.L., McDermid, J.A.; 'Guiding Reuse with Risk Assessments'; University of York Technical Document YCS 183 (1992); York; 1992

[Burd93a] Burd, E.L., McDermid, J.A.; 'Risk Management: the Key to Successful Reuse'; In: Proc. of the Sixth Annual Workshop on Software Reuse; Poulin, J. (ed.); IBM Federal Systems Company, Owego, NY; Nov 1993

[Burd93b] Burd, E.L.; quoted in 'Spiral of Success'; Peltu, M.; Computing; 28 Jan 1993; P24

[Burn86] Burns, P., Dewhurst, J. (ed.); 'Small Business in Europe'; Macmillan; 1986

[Burn96a] Burns, P., Dewhurst, J. (ed.); 'Small Business and Entrepreneurship (2nd Edition)'; Macmillan; 1996

[Burn96b] Burnes, B.; 'Managing Change: A Strategic Approach to Organisation Dynamics (2nd Edition)'; Pitman, London; 1996

[Carn95] Carnegie Mellon University, Software Engineering Institute; 'The Capability Maturity Model: Guidelines for Improving the Software Process'; Addison-Wesley; 1995

[Cava83] Cavaliere, M.J.; 'Reusable Code at the Hartford Insurance Group'; In: Software Reusability. Applications and Experience, vol. II; Biggerstaff, T.J., Perlis, A.J., (ed.); ACM Press, Addison-Wesley, Reading, Mass.; 1989; P131-141

[Chao93] Chao, D.; 'Software Reuse: Major Issues Need to Be Resolved Before Benefits Can Be Achieved'; In: Proc. of the Sixth Annual Workshop on Software Reuse; Poulin, J. (ed.); IBM Federal Systems Company, Owego, NY; Nov 1993

[Chea84] Cheatham, T.E.; 'Reusability through program transformations'; IEEE Transactions on Software Engineering; Sept 1984; Vo.10 No.5

[Chik90] Chikofsky, E.J., Cross, J.H.; 'Reverse Engineering and Design Recovery: A Taxonomy'; IEEE Software; Jan 1990; Vol.7 No.1; P13-18

[Chil96] Childs, B., Sametinger, J.; 'Literate Programming and Documentation Reuse'; In: Proc. of 4th International Conference on Software Reuse; IEEE, Orlando, Florida; IEEE Computer Society Press; Apr 1996; P205-214

[Chis87] Chisnall, P.M.; 'Small Firms in Action'; McGraw-Hill; 1987

[Cimi95] Cimitile, A., De Lucia, A., Munro, M.; 'Identifying Reusable Functions Using Specification Driven Program Slicing: A Case Study'; In: Proc. of International Conference on Software Maintenance; IEEE, Nice, France; IEEE Computer Society Press; 1995; P124-133

[Cox86] Cox, B.J.; 'Object-Oriented Programming - An Evolutionary Approach'; Addison-Wesley, Reading, Mass.; 1986

[Curt92] Curtis, B.; 'Maintaining the Software Process'; IEEE Proc. of the Conference on Software Maintenance 1992; P2-8

[DeMa84] DeMarco, T., Lister, T.; 'Controlling Software Projects: Management, Measurement, and Evaluation'; Seminar Notes; Atlantic Systems Guild Inc.; 1984

[Dijk79] Dijkstra, E.; 'Programming Considered as a Human Activity'; Classics in Software Engineering, Yourdan Press, New York; 1979

[Fair89] Fairley, R., Pfleeger, S.L., Bollinger, T., Davis, A., Incorvaia, A.J., Springsteen, B.; 'Final Report: Incentives for Reuse of Ada Components, vols. 1 - 5'; George Mason University, Fairfax, VA; 1989

[Frak88] Frakes, W.B., Nejmeh, B.A.; 'An Information System for Software Reuse'; In: Software Reuse: Emerging Technology; Tracz, W. (ed.); IEEE Computer Society Press; 1988

[Frak92] Frakes, W.B.; 'Software Reuse: An Empirical Approach'; In: Annual Review of Automatic Programming; Elzer, P., Haase, V. (ed.); Pergammon Press, Oxford; 1992; Vol.16 Part II P41-44

[Fraz92] Frazer, A.; 'Reverse Engineering - hype, hope or here?'; In: Software Reuse and Reverse Engineering in Practice; Hall, P.A.V. (ed.); Chapman & Hall, London; 1992

[Free83] Freeman, P.; 'Reusable Software Engineering: Concepts and Research Directions'; In: Workshop on Reusability in Programming; Perlis, A. (ed.); ITT Programming, Newport, RI; Sept 1983; P2-16

[Gear88] Geary, K.; 'The practicalities of introducing large-scale software reuse'; Software Engineering Journal; Sept 1988; Vol.3 No.5 P175-176

[Ghez91] Ghezzi, C., Jazayeri, M., Mandrioli, D.; 'Fundamentals of Software Engineering'; Prentice Hall, New Jersey; 1991

[Gold83] Goldberg, A., Robson, D.; 'Smalltalk-80: The Language and its Implementation'; Addison-Wesley, Reading, Mass.; 1983

[Goss90] Gossain, S., Anderson, B.; 'An Iterative-Design Model for Reusable Object-Oriented Software'; ECOOP/OOPSLA'90 Proceedings; Oct 1990; P12-27

[Goug86] Gougen, J.A.; 'Reusing and Interconnecting Software Components'; IEEE Computer; Feb 1986; Vol.19 No.2 P16-28

[Hall91] Hall, P.A.V., Boldyreff, C.; 'Software Reuse'; In: Software Engineer's Reference Book; McDermid, J.A. (ed.); Butterworth-Heinemann Ltd., Oxford; 1991; P41/3-41/12

[Hall93] Halladay, S., Wiebel, M.; 'Object-Oriented Software Engineering'; R&D Publications Ltd., Prentice Hall, London; 1993

[Hatt95] Hatton, L.; 'Bugs: Avoiding the avoidable and living with the rest'; In: Proc. of the 9th European Workshop on Software Maintenance; Centre for Software Maintenance, Dept. of Computer Science, Durham University; 1995

[Hoop91] Hooper, J.W., Chester, R.O.; 'Software Reuse: Guidelines and Methods'; Plenum Press, New York; 1991

[Hump89] Humphrey, W.S.; 'Managing the software process'; Addison-Wesley, Reading, Mass.; 1989

[Hump93] Humphrey, W.S.; 'The Personal Software Process – Rationale and Status'; In: Proceedings of the 8th International Software Process Workshop; Schaffer, W. (ed.); IEEE Computer Society Press, 1993; P102-103

[Hutc88] Hutchinson, J.W., Hindley, P.G.; 'A Preliminary study of Large-Scale Software Reuse'; Software Engineering Journal; Sept 1988; Vol.3 No.5 P208-212

[Ince91] Ince, D.; 'Object-Oriented Software Engineering with C++'; McGraw-Hill, London; 1991

[Jack83] Jackson, M.A.; 'System Development'; Prentice Hall, New Jersey; 1983

[Jaco92] Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.; 'Object-Oriented Software Engineering - A Use Case Driven Approach'; ACM Press, Addison-Wesley, Reading, Mass.; 1992

[Jaco97] Jacobson, I., Griss, M., Jonsson, P.; 'Software Reuse: Architecture, Process and Organisation for Business Success'; Addison-Wesley, Reading, Mass.; 1997

[John88] Johnson, R.E., Foote, B.; 'Designing Reusable Classes'; Journal of Object-Oriented Programming; Jun/Jul 1988; Vol.1 No.2 P22-30, 35

[Jone84] Jones, T.C.; 'Reusability in Programming: A Survey of the State of the Art'; IEEE Transactions on Software Engineering; Sept 1984; Vol.10 No.5; P488-494

[Jone86] Jones, T.C.; 'Programming Productivity'; McGraw-Hill, New York; 1986

[Jone92] Jones, R.; 'How applicable is the object-oriented approach to the IS environment?'; In: Software Reuse and Reverse Engineering in Practice; Hall, P.A.V. (ed.); Chapman & Hall, London; 1992

[Kang89] Kang, K.C.; 'Features Analysis: An Approach to Domain Analysis'; In: Proc. of the Reuse in Practice Workshop; Baldo, J, Braun, C. (ed.); Software Engineering Institute, Pittsburgh, Penn.; Jul 1989

[Karl95] Karlsson, E. (ed); 'Software Reuse: A Holistic Approach'; John Wiley & Sons; 1995

[Knut84] Knuth, D.E.; 'Literate Programming'; The Computer Journal; 1984; Vol.27 No.2 P97-111

[Koch93] Koch, G.; 'Process assessment: The BOOTSTRAP approach'; Proceedings of Software Process Modelling in Practice; April 1993; P22-23

[Lane79] Lanergan, R.G., Poynton, B.A.; 'Reusable code - The application development technique of the future'; In: Proc. of Joint SHARE/GUIDE/IBM Applications Development Symposium; Oct 1979; P127-136

[Lane84] Lanergan, R.G., Grasso, C.A.; 'Software Engineering with Reusable Design and Code'; IEEE Transactions on Software Engineering; Sept 1984; Vol.10 No.5 P498-501

[Leac94] Leach, E.; 'The Likely Impact of Object Technology on Software Development & Maintenance'; In: Proc. of 8th European Software Maintenance Workshop; University of Durham; September 1994

[Leac97] Leach, R.J.; 'Software Reuse: Methods, Models and Costs'; McGraw-Hill, New York; 1997

[Lisk74] Liskov, B., Zilles, S.; 'Programming with Abstract Data Types'; ACM Sigplan Notices; Apr 1974; Vol.9 No.4 P50-59

[Maar90] Maarek, Y.; 'Indexing Software Components for Reuse by Using Natural-Language Documentation'; In: Proc. of the Third Annual Workshop: Methods and Tools for Reuse; Frakes, W. (Chair); CASE Center Technical Report Series; June 1990

[Mats84] Matsumoto, Y.; 'Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels'; IEEE Transactions on Software Engineering; Sept 1984; Vol.10 No.5 P502-513

[McCl97] McClure, C.; 'Software Reuse Techniques: Adding Reuse to the Systems Development Process'; Prentice Hall, New Jersey; 1997

[McIl68] McIlroy, M.D.; 'Mass-produced Software Components'; In: Software Engineering Concepts and Techniques, 1968 NATO Conference Software Engineering; Buxton, J.M., Naur, P., Randell, B. (ed.); 1976; P88-98

[Melo95] Melo, W.L., Briand, L.C., Basili, V.R.; 'Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems'; University of Maryland Technical Report CS-TR-3395; Dept. of Computer Science, University of Maryland; Jan. 1995

[Meye87] Meyer, B.; 'Reusability: The Case for Object-Oriented Design'; IEEE Software; Mar 1987; Vol. No. P50-64

[Meye88] Meyer, B.; 'Object-Oriented Software Construction'; Prentice Hall, New Jersey; 1988

[Meye94] Meyer, B.; 'Reusable Software: Base OO Component Libraries'; Prentice Hall, New Jersey; 1994

[Micr93a] Microsoft Corporation; 'Visual Workbench User's Guide'; Microsoft Corp.; 1993

[Micr93b] Microsoft Corporation; 'OLE 2 Classes for the Microsoft Foundation Class Library'; Microsoft Corp.; 1993

[Micr93c] Microsoft Corporation; 'Class Library Reference for the Microsoft Foundation Class Library'; Microsoft Corp.; 1993

[Micr93d] Microsoft Corporation; 'Run-Time Library Reference'; Microsoft Corp.; 1993

[Moin90] Moineau, T., Abadir, J., Rames, E.; 'Towards a Generic and Extensible Reuse Environment'; SE90, Proc. of Software Engineering 1990; Hall, P.A.V. (ed.); Cambridge University Press; 1990; P543-569

[Mort96] Mortimer, R.E., Bennett, K.H.; 'Maintenance and Abstraction of Program Data using Formal Transformations'; Proc. of 1996 International Conference on Software Maintenance; IEEE, Monterey, U.S.A.; IEEE Computer Society Press; Nov 1996; P301-310

[Mull89] Mullin, M.; 'Object-Oriented Program Design with Examples in C++'; Addison-Wesley, Reading, Mass.; 1989

[Munr92] Munro, M.; 'Software maintenance, reuse and reverse engineering'; In: Software Reuse and Reverse Engineering in Practice; Hall, P.A.V. (ed.); Chapman & Hall, London; 1992

[Neig96] Neighbours, J.M.; 'Finding Reusable Software Components in Large Systems'; In: Proc. of Third Working Conference on Reverse Engineering; IEEE, California, U.S.A; IEEE Computer Society Press; Nov 1996; P2-10

[Ning93] Ning, J.Q., Engberts, A., Kozaczynski, W.; 'Recovering Reusable Components from Legacy Systems by Program Segmentation'; In: Proc. of 1993 Working Conference on Reverse Engineering; IEEE, Maryland, U.S.A,; IEEE Computer Society Press; May 1993; P64-72

[Oxfo90] Illingworth, V., Glaser, E.L., Pyle, I.C. (ed.); 'Dictionary of Computing (3rd Ed.)'; Oxford University Press, Oxford; 1990

[Parn72] Parnas, D.; 'On the Criteria to be Used in Decomposing Systems Into Modules'; Communications of the ACM; Dec 1972; Vol.15 No.12 P1053-1058

[Pfle95] Pfleeger, S.L.; 'Experimental Design and Analysis in Software Engineering'; Software Engineering Notes; Jan 1995; Vol.20 No.1 P22-26

[Pont96] Pont, M.J.; 'Software Engineering with C++ and CASE Tools'; Addison-Wesley, Reading, Mass.; 1996

[Pott93] Potts, C.; 'Software-Engineering Research Revisited'; IEEE Software; Sept 1993; Vol.10 No.5 P19-28

[Prat91] Pratten, C.; 'The Competitiveness of Small Firms'; Cambridge University Press; 1991

[Pres92] Pressman, R.S.; 'Software Engineering - A Practitioner's Approach (3rd Edition - European Adaptation)'; McGraw-Hill, London; 1992

[Prie87] Prieto-Diaz, R., Freeman, P.; 'Classifying Software For Reusability'; IEEE Software; Jan 1987; Vol.4 No.1 P6-16

[Prie90] Prieto-Diaz, R.; 'Implementing Faceted Classification for Software Reuse'; In: Proc.of 12th International Conference on Software Engineering; IEEE, Nice, France; Mar 1990; P300-304

[Prie91] Prieto-Diaz, R.; 'Making Software Reuse Work: An Implementation Model'; Software Engineering Notes; Jul 1991; Vol.16 No.3 P61-68

[Prie93] Prieto-Diaz, R.; 'Status Report: Software Reusability'; IEEE Software; May 1993; Vol.10 No.3 P61-66

[Radf94] Radford, J.D.; 'The Engineer and Society'; MacMillan Publishers, London; 1984

[Raft94] Raftery, J.; 'Risk Analysis in Project Management'; Chapman & Hall; 1994

[Raj89] Raj, R.K., Levy, H.M.; 'A Compositional Model for Software Reuse'; In: ECOOP'89 Proc. of the 1989 European Conference on Object-Oriented Programming; Cook, S. (ed.); Cambridge University Press; 1989; P3-24

[Robs91] Robson, D.J., Bennett, K.H., Cornelius, B.J.; 'Approaches to Program Comprehension'; Journal of Systems and Software; 1991; Vol.14 No.2 P79

[Rumb91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.; 'Object-Oriented Modeling and Design'; Prentice Hall, New Jersey; 1991

[Schw86] Schwartz, J.T., Dewar, R., Dubinski, E., Schonberg, E.; 'Programming with Sets: An Introduction to SETL'; Springer-Verlag, New York; 1986

[Somm89] Sommerville, I.; 'Software Engineering (3rd Ed.)'; Addison-Wesley, Reading, Mass.; 1989

[Somm96] Sommerville, I.; 'Software Engineering (5th Ed.)'; Addison-Wesley, Reading, Mass.; 1996

[Stan84] Standish, T.A.; 'An Essay on Software Reuse'; IEEE Transactions on Software Engineering; Sept 1984; Vol.10 No.5 P494-497

[Tell91] Tello, E.R., 'Object-Oriented Programming for Windows'; John Wiley and Sons; 1991

[Thom97] Thompson, H.E., Mayhew, P.; 'Approaches to Software Process Improvement'; Software Process – Improvement and Practice; 1997; Vol.3 P3-17

[Trac87a] Tracz, W.; 'Software Reuse Myths'; In: Proc. of the Workshop on Software Reuse; Booch, G., Williams, L. (ed.); Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.; Oct 1987

[Trac87b] Tracz, W.; 'Reusability Comes of Age'; IEEE Software; Jul 1987; Vol.4 No.4 P6-8

[Trac88a] Tracz, W.; 'Software Reuse Myths'; ACM Software Engineering Notes; Jan 1988; Vol.13 No.1 P17-21

[Trac88b] Tracz, W.; 'Software Reuse Maxims'; ACM Software Engineering Notes; Oct 1988; Vol.13 No.4 P28-31

[Trac90] Tracz, W.; 'Where Does Reuse Start?'; ACM Software Engineering Notes; Apr 1990; Vol.15 No.2 P42-46

[Tsic89] Tsichritzis, D.C., Nierstrasz, O.M.; 'Directions in Object-Oriented Research'; In: Object-Oriented Concepts, Databases, and Applications; Kim, W., Lochovsky, F.H. (ed.); ACM Press, Addison-Wesley, Reading, Mass.; 1989

[Udel94] Udell, J.; 'Componentware'; Byte; May 1994; Vol.19 No.5 P46-56

[Walt92] Walton, P.; 'The management of reuse'; In: Software Reuse and Reverse Engineering in Practice; Hall, P.A.V. (ed.); Chapman & Hall, London; 1992

[Wass91] Wasserman, A.I.; 'Object-Oriented Software Development : Issues in Reuse'; Journal of Object-Oriented Programming; May 1991; Vol.4 No.2 P55-57

[Webe91] Weber, H.; 'The Integration of Reusable Software Components'; Journal of Systems Integration; 1991; Vol.1 P55-79

[Weis74] Weissman, L.; 'Psychological complexity of computer programs: An experimental methodology'; ACM SIGPLAN Notices; 1974; Vol.9 No.6 P25-36

[Wien88] Wiener, R.S., Pinson, L.J.; 'An Introduction to Object-Oriented Programming and C++'; Addison-Wesley, Reading, Mass.; 1988

[Winb90] Winblad, A.L., Edwards, S.D., King, D.R.; 'Object-Oriented Software'; Addison-Wesley, Reading, Mass.; 1990

[Wirf90] Wirfs-Brock, R., Wilkerson, B., Wiener, L.; 'Designing Object-Oriented Software'; Prentice Hall, New Jersey; 1990

[Wolf92] Wolff, F.; 'Long-term Controlling of Software Reuse'; Information and Software Technology; Mar 1992; Vol.34 No.3 P178-184

[Yu91] Yu, D.; 'A View On Three R's (3Rs): Reuse, Re-engineering, and Reverse-engineering'; ACM Software Engineering Notes; Jul 1991; Vol.16 No.3 P69

[Zigm95] Zigman, F.J., Wilson, M.L.; 'Integrating Reengineering, Reuse and Specification Tool Environments to Enable Reverse Engineering'; In: Proc. of Second Working Conference on Reverse Engineering; IEEE, Ontario, Canada; IEEE Computer Society Press; Jul 1995; P78-84

Chapter 10: Bibliography

[Abbo83] Abbott, R.; 'Program Design by Informal English Description'; Communications of the ACM; Nov 1983; Vol.26 No.11 P882-894

[Aran91] Arango, G., Prieto-Diaz, R.; 'Part 1: Introduction and Overview, Domain Analysis Concepts and Research Directions'; In: Domain Analysis and Software Systems Modelling; IEEE Computer Society Press Tutorial, IEEE; 1991

[Bank93] Banker, R.D., Datar, S.M.; Kemerer, C.F. Zweig, D.; 'Software Complexity and Maintenance Costs'; Communications of the ACM; Nov 1993; Vol.36 No.11

[Bank94] Banker, R.D., Kaufmann, R.J., Wright, C., Zweig, D.; 'Automating Output Size and Reuse Metrics in a Repository-Based Computer-Aided Software Engineering (CASE) Environment'; IEEE Transactions on Software Engineering; Mar 1994; Vol.20 No.3 P169-187

[Bilo91] Bilow, S.C.; 'Book Review: Object-Oriented Design'; Journal of Object-Oriented Programming; Oct 1991; Vol.4 No.6 P73-74

[Booc86] Booch, G.; 'Object-Oriented Development'; IEEE Transactions on Software Engineering; Feb 1986; Vol.12 No.2 P211-221

[Booc87] Booch, G.; 'Software Engineering with Ada (2nd Ed.)'; Benjamin/Cummings, California; 1987

[Booc91] Booch, G.; 'Object-Oriented Design with Applications'; Benjamin/Cummings, California; 1991

[Coad90] Coad, P., Yourdan, E.; 'Object-Oriented Analysis'; Yourdan Press, Prentice Hall, New Jersey; 1990

[Coad91a] Coad, P., Yourdan, E.; 'Object-Oriented Design'; Yourdan Press, Prentice Hall, New Jersey; 1991

[Coad91b] Coad, P.; 'Why use object-oriented development? (A management perspective)'; Journal of Object-Oriented Programming;' Oct 1991

[Dunt90] Duntemann, J., Marinacci, C.; 'New Objects for Old Structures'; Byte; Apr 1990; P261-266

[Evan90] Evans, R.A.; 'Criteria for an OOD method'; In: Object-Oriented Software Engineering; Anderson, B. (ed.); British Computer Society, London; 1990

[Goss91] Gossain, S.; 'Book Review: Designing Object-Oriented Software'; Journal of Object-Oriented Programming; Mar/Apr 1991; Vol.4 No.1 P82-84

[Grah93] Graham, I; 'Object-Oriented Methods'; Addison-Wesley; 1993

[Hall92] Hall, P.A.V.; 'Software Reuse, Reverse Engineering and Reengineering'; In: Software Reuse and Reverse Engineering in Practice'; Hall, P.A.V (ed.); Chapman & Hall, London; 1992

[Hend93] Henderson, P.; 'Object-Oriented Specification and Design with C++'; McGraw-Hill, London; 1993

[Hoar72] Hoare, C.A.R., Dahl, O., Dijkstra, E.; 'Structured Programming'; Academic Press, London; 1972

[Hodg92] Hodgson, R.; 'Finding, building and reusing objects'; In: Object-Oriented design; Robinson, P. (ed.); Chapman & Hall, London; 1992; P48-76

[Hood93] Delatte, B., Heitz, M., Muller, J.F. (ed.); 'HOOD Reference Manual 3.1'; Masson, Paris; 1993

[Horo89] Horowitz, E., Munson, J.B.; 'An expansive view of reusable software'; In: Software Reusability. Concpets and Models, vol. I; Biggerstaff, T.J., Perlis, A.J., (ed.); ACM Press, Addison-Wesley, Reading, Mass.; 1989; P19-41

[Ince88] Ince, D.; 'Reusable Software - The False Frontier'; In: Software Development: Fashioning the Baroque'; Ince, D.; Oxford University Press; 1988

[Kang87] Kang, K.C.; 'A Reuse-Based Software Development Methodology'; In: Proc. of the Workshop on Software Reuse; Booch, G., Williams, L. (ed.); Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder, Colo.; Oct 1987

[Knut94] Knuth, D.E., Levy, S.; 'The CWEB System of Structured Documentation'; Addison-Wesley, Reading, Mass.; 1994

[Lewi92] Lewis, J.A., Henry, S.M., Kafura, D.G., Schulman, R.S.; 'On the relationship between the object-oriented paradigm and software reuse: an empirical investigation'; Journal of Object-Oriented Programming; Jul/Aug 1992; P35-41

[Luba88] Lubars, M.D.; 'Code reusability in the large versus code reusability in the small'; In: Software Reuse: Emerging Technology; Tracz, W. (ed.); IEEE Computer Society Press; 1988

[Myer94] Myers, W.; 'Workshop explores large-grained reuse'; IEEE Software; Jan 1994; P108-109

[Orms91] Ormsby, A.; 'Object-Oriented Design Methods'; In: Object-Oriented Languages, Systems and Applications; Blair, G., Gallagher, J., Hutchison, D., Shepherd, D. (ed.); Longman, London; 1991; P203-222

[Reen92] Reenskaug, T., Andersen, E., Berre, A., Hurlen, A., Landmark, A., Lehne, O., Nordhagen, E., Ness-Ulseth, E., Oftedal, G., Skaar, A., Stenslet, P.; 'OORASS: seamless support for the creation and maintenance of object-oriented systems'; Journal of Object-Oriented Programming; Oct 1992; Vol.5 No.6 P27-41

[Rent82] Rentsch, T.; 'Object-Oriented Programming'; SIGPLAN Notices; Sept 1982; Vol.17 No.12; P51

[Robe93] Roberts, S.; 'Productivity Benefits in Major Maintenance projects: Reverse is the Wrong Direction'; In: Proc. of Reuse and Reverse Engineering For Productive Software Development; Unicom Seminars; 1993; P49-91

[Robi92] Robinson, P.J.; 'Hierarchical Object-Oriented Design'; Prentice Hall, London; 1992

[Rumb94] Rumbaugh, J.: 'Getting started: Using use cases to capture requirements"; Journal of Object-Oriented Programming; Sept 1994; Vol.7 No.5 P8-12,23

[Sepp92] Seppanen, V.; 'Acquisition, organisation and reuse of software design knowledge'; Software Engineering Journal; Jul 1992; P238-246

[Shea93] Shearer, D.; 'Working Examples from the BT Corporate Reuse Program'; In: Proc. of Reuse and Reverse Engineering For Productive Software Development; Unicom Seminars; 1993; P93-108

[Shla92] Shlaer, S., Mellor, S.; 'Object Lifecycles: Modeling the World in States'; Prentice Hall; 1992

[Smit90] Smith, J.D.; 'Reusability and Software Construction: C and C++'; John Wiley and Sons; 1990

[Stev91] Stevens, W.; 'Code Reuse'; In: Software Design, Concepts and Models; Stevens, W.; Prentice Hall, London; 1991

[Walk92] Walker, I.J.; 'Requirements of an object-oriented design method'; Software Engineering Journal; Mar 1992; Vol.7 No.2 P102-113

[Wass89] Wasserman, A.I., Pircher, P.A., Muller, R.J.; 'An Object-Oriented Structured Design Method for Code Generation'; ACM Software Engineering Notes; Jan 1989; Vol.14 No.1 P32-55

[Wass90] Wasserman, A.I., Pircher, P.A., Muller, R.J.; 'The Object-Oriented Structured Design Notation for Software Design Representation'; IEEE Computer; Mar 1990; Vol.23 No.3 P50-63

[Webe93] Weber, H.; 'Uniformity and Invariance in Support of Re-Use'; In: Advances in Software Reuse; Prieto-Diaz, R., Frakes, W.B. (ed.); IEEE Computer Society Press; 1993

[Wegn87] Wegner, P.; 'Varieties of reusability'; In: Tutorial: Software Reusability; Freeman, P. (ed.); IEEE Computer Society Press; 1987

[Wegn90] Wegner, P.; 'Concepts and Paradigms of Object-Oriented Programming'; OOPS Messenger; Aug 1990; Vol.1 No.1 P7-87

[Wilk90] Wilkerson, B.; 'How to Design an Object-Based Application'; Develop; Apr 1990; P178-203

Appendix A

A1. Software Reuse Questionnaire

Name?

Position in Company?

How long in Company?

Your Work

What does your work consist of?

If programming, what languages do you use? What compilers?

Who provides the drive behind the work that you do (customers/company/self)?

When you have a new idea, what process do you follow to get from the idea to the realisation of the idea?

How do you write down the requirements/specification of new ideas and modifications?

What design methods have you used?

Do you conduct/participate in design reviews?

What type of design do you think would be most suitable for your work?

To what extent do you use an object-oriented methods?

Do you ever use: Modularisation?

Inheritance?

Overloading?

Class libraries/hierarchies?

How easy do you find it to understand: your own code?

a colleagues code?

standard library code?

To what extent do you document your code?

How could understanding code be made easier?

If you need a function/method, do you: look for it in the standard libraries/try to find someone else who has done it/write it yourself?

If another function doesn't do quite what you want it to do, do you: look for another/modify it/write it yourself from scratch?

When writing a function, do you ever consider that someone else may use it, and take steps to make it more generic/easier to use?

The Company

How would you describe the company at the moment?

Where do you see the company going in the next few months?

Where do you see the company going in the next few years?

How do you think that software reuse could help the company achieve its goals?

Appendix B

B1. ReThree-C++

ReThree-C++ is an integrated reverse engineering and reuse tool set. It can be used to extract information from C++ source code and to create a repository of C++ classes for later retrieval. Using visualisation and re-documentation techniques, software documentation and class structure hierarchies for candidate software components are automatically generated from the software source code. The tool set can be divided into three main functions:

1. Automatically reverse engineering C++ source code to give a visual class hierarchy representation in OMT object model format.
2. Documenting C++ source code, based on the comments contained within the code, to provide automatically generated software documentation.
3. Building, maintaining and searching a reuse repository of C++ classes which can be reused in later applications.

ReThree-C++ is designed with small company developers in mind, who are under pressure to complete their coding to tight deadlines. Its purpose is to help them to achieve the benefits that reuse of code can bring without the large up-front investment that is usually required for reuse to be successful. In order to reuse code, it is necessary to have appropriate code available, as well as being able to find the code, modify it (if necessary) and integrate it into the current system.

The principles of object-oriented design are useful for building reusable code in manageable components. However, there is little tool support for the process of making code reusable, storing it for later use, retrieving it when needed and understanding the structure of reusable components. ReThree-C++ addresses these problems. It is based solely on C++ source code, and provides automatic reverse engineering and documentation of source code to help developers

understand the structure of code to be reused. It also provides reuse repository support, allowing classes to be added to a reuse repository and providing search facilities for repositories. Classes that match the search criteria can be automatically reverse engineered and documented to help the developer understand the structure and purpose of the code.

The source code is used as the base for all information generated so that the software engineers are encouraged to spend more time on developing and maintaining their code effectively. The commented source code can then be automatically converted into class hierarchies and documentation for the code. This automatic generation of information is done by static analysis of the source code in a few seconds. However, as the re-documentation is based on the comments contained within the source code, the information given about the classes, their services and their attributes, will only be as useful as the comments provided by the developers.

Reverse engineering provides an Object Modelling Technique class hierarchy diagram of the classes described in the C++ source code. Documentation is taken from the comments in the source code which describe the functionality of the code. The system interfaces with Windows® tools, namely Word, OMTool and Netscape, to display the results generated in an informative fashion. The latter also has the advantage allowing the full power of the browser's searching facilities to be employed on the documentation.

B2. Examples of Use

ReThree-C++ has three different forms of output, all based on information taken directly from C++ source code. Section B2.1 contains an example C++ header file (which is taken from the source of the ReThree-C++ system). Sections B2.2, B2.3 and B2.4 show the different types of output which ReThree-C++ gives based on that file.

B2.1 Example Source Code

```
// aboutbox.h : header file Version Number = 1.2
//
// This source code is only intended as a supplement to the
// Microsoft Foundation Classes Reference and Microsoft
// QuickHelp and/or WinHelp documentation provided with the library.
// See these sources for detailed information regarding the
// Microsoft Foundation Classes product.

const CString VERSION = "2.32";

/////////////////////////////////////////////////////////////////////////
//
// CBigIcon window version number = 1
// This class contains the big version of the application's icon that
is
// used on the splash window.
/////////////////////////////////////////////////////////////////////////
//

class CBigIcon : public CButton
{
// Attributes
```

```
public:

// Operations
public:
      void SizeToContent();
            // Resizes the standard icon to fit in the designated area
            // on the splash window.

// Implementation
protected:
      virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
      // Draws the big icon to the rectangle specified on
      // the splash window, including a border and shadowing.

      //{{AFX_MSG(CBigIcon)
      afx_msg BOOL OnEraseBkgnd(CDC* pDC);
      // Background does not need to be erased -
      // this function does nothing.
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
//
// CAboutBox dialog. version number = 2
// This dialog contains information about the name and version number
of
// the current application. It also gives information about the
current
// system status, including how much memory is free, whether the
// computer has a maths co-processor and how much disc space is free.
/////////////////////////////////////////////////////////////////////////////
//
```

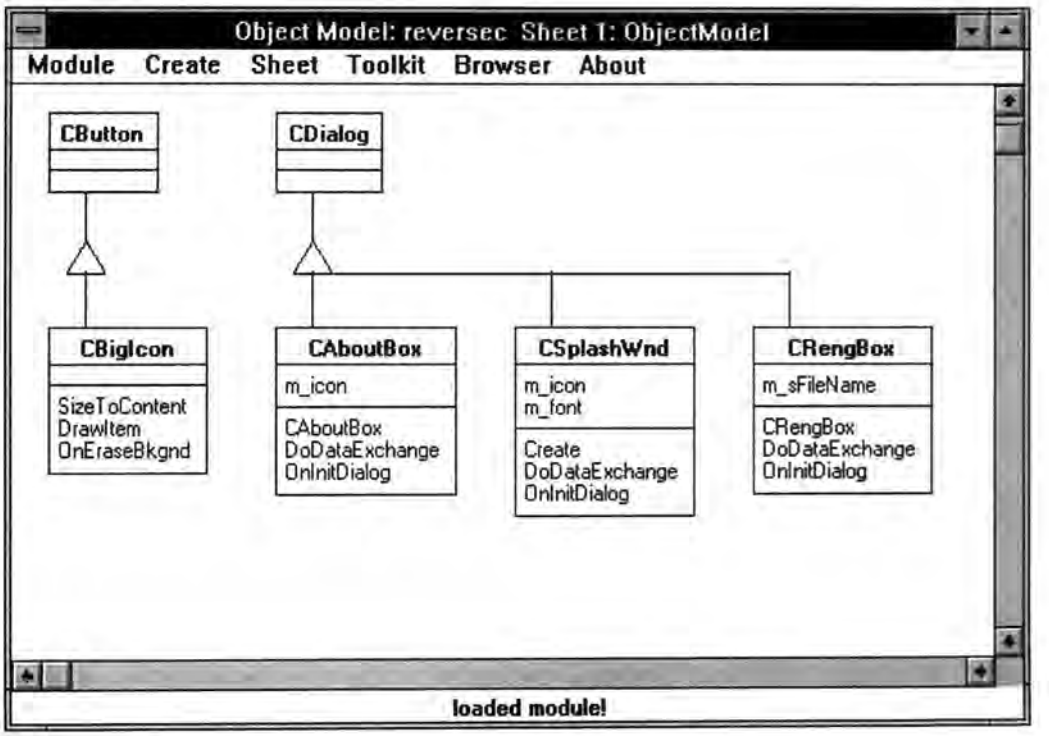
```
class CAboutBox : public CDialog
{
// Construction
public:
      CAboutBox(CWnd* pParent = NULL);
      // standard constructor with no member initialisation

// Dialog Data
      //{{AFX_DATA(CAboutBox)
      enum { IDD = IDD_ABOUTBOX };
      // NOTE: the ClassWizard will add data members here
      //}}AFX_DATA

// Implementation
protected:
      virtual void DoDataExchange(CDataExchange* pDX);
      // DDX/DDV support. This method is controlled by the VC++
      // Class Wizard.

      CBigIcon m_icon;
      // self-draw button. A large version of the application's icon.

      // Generated message map functions
      //{{AFX_MSG(CAboutBox)
      virtual BOOL OnInitDialog();
      // Includes all the initialisation that is done when this dialog
      // is called. This method draws the big icon, gets the current
      // version number of the application, calculates current free
      // memory and disc space and whether a math co-processor is
      // present.
      //}}AFX_MSG
```

```
	DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
//
// CSplashWnd dialog. version number = 3
// This dialog is called when the application is initialised to give
the
// user information about the application, including the version
number
// and copyright information.
/////////////////////////////////////////////////////////////////////////////
/


class CSplashWnd : public CDialog
{
// Construction
public:
	BOOL Create(CWnd* pParent);
	// Returns an error if the splash window could not be created.

// Dialog Data
	//{{AFX_DATA(CSplashWnd)
	enum { IDD = IDD_SPLASH };
		// NOTE: the ClassWizard will add data members here
	//}}AFX_DATA

// Implementation
protected:
	virtual void DoDataExchange(CDataExchange* pDX);
	// DDX/DDV support. This method is controlled by the VC++
	// Class Wizard.
```

```
        CBigIcon m_icon;
        // self-draw button. A large version of the application's icon.
        CFont m_font;
        // light version of dialog font

        // Generated message map functions
        //{{AFX_MSG(CSplashWnd)
        virtual BOOL OnInitDialog();
        // Initialisation code for the dialog. Draws the big version of
        // the icon and gets the current version number for the
        // application.
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
//
// CRengBox dialog. Version Number = 1
// This dialog is used to ask the user if they wish to start the
// appropriate visualisation program for the processing that has just
// been carried out.
/////////////////////////////////////////////////////////////////////////////
//

class CRengBox : public CDialog
{
// Construction
public:
        CRengBox(CWnd* pParent = NULL);
        // standard constructor with no member initialisation.

// Dialog Data
```

```
        //{{AFX_DATA(CRengBox)
        enum { IDD = IDD_RENGBOX };
        CString     m_sFileName;
        //}}AFX_DATA


// Implementation
protected:
        virtual void DoDataExchange(CDataExchange* pDX);
        // DDX/DDV support. This method is controlled by the VC++
        // Class Wizard.


        // Generated message map functions
        //{{AFX_MSG(CRengBox)
        virtual BOOL OnInitDialog();
        // Initialises the dialog with the name of the source/make file
        // which has just been processed.
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

B2.2 OMT Class Hierarchy Representation



This is displayed using the demonstration version of OMTool. The layout shown above is directly from ReThree-C++. OMTool is fully interactive, meaning that the classes can be re-arranged and updated as required.

One of the disadvantages of interfacing with OMTool is that it has now fallen out of fashion, and has been replaced by better OMT CASE tools. This is always a danger when interfacing with 'standard' desktop software. OMTool is not compatible with Windows® 95, and this type of class hierarchy generation is therefore limited only to users who have the software running under Windows® 3.x.

An alternative is available for Windows® 95 users - the class hierarchies generated in Java for Web browsers. The results of this type of processing can be seen in section B2.4.

CButton

Sub Classes: CBigIcon

Location: C:\PETEPROJ\REUSE\TESTREP\ABOUTBOX.H

CBigIcon

Version: 1

Super Classes: CButton

Location: C:\PETEPROJ\REUSE\TESTREP\ABOUTBOX.H

Overview

CBigIcon window version number = 1 This class contains the big version of the application's icon that is used on the splash window.

Services

Public Members

void SizeToContent()

Resizes the standard icon to fit in the designated area on the splash window.

Protected Members

virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)

Draws the big icon to the rectangle specified on the splash window, including a border and shadowing.

afx_msg BOOL OnEraseBkgnd(CDC* pDC)

Background does not need to be erased - this function does nothing. AFX_MSG

CDialog

Sub Classes: CAboutBox CSplashWnd CRengBox

Location: C:\PETEPROJ\REUSE\TESTREP\ABOUTBOX.H

CAboutBox

Version: 2

Super Classes: CDialog

Location: C:\PETEPROJ\REUSE\TESTREP\ABOUTBOX.H

Overview

CAboutBox dialog. version number = 2 This dialog contains information about the name and version number of the current application. It also gives information about the current system status, including how much memory is free, whether the computer has a maths co-processor and how much disc space is free.

Services

Public Members

CAboutBox(CWnd* pParent = NULL)

standard constructor with no member initialisation

Protected Members

virtual void DoDataExchange(CDataExchange* pDX)

DDX/DDV support. This method is controlled by the VC++ Class Wizard.

virtual BOOL OnInitDialog()

Includes all the initialisation that is done when this dialog is called. This method draws the big icon, gets the current version number of the application, calculates current free memory and disc space and whether a math co-processor is present. AFX_MSG

Attributes

Protected Members

CBigIcon m_icon

self-draw button. A large version of the application's icon.

CSplashWnd

Version: 3

Super Classes: CDialog

Location: C:\PETEPROJ\REUSE\TESTREP\ABOUTBOX.H

Overview

CSplashWnd dialog. version number = 3 This dialog is called when the application is initialised to give the user information about the application, including the version number and copyright information.

Services

Public Members

BOOL Create(CWnd* pParent)

Returns an error if the splash window could not be created.

Protected Members

virtual void DoDataExchange(CDataExchange* pDX)

DDX/DDV support. This method is controlled by the VC++ Class Wizard.

virtual BOOL OnInitDialog()

Initialisation code for the dialog. Draws the big version of the icon and gets the current version number for the application. AFX_MSG

Attributes

Protected Members

CBigIcon m_icon

self-draw button. A large version of the application's icon.

CFont m_font

light version of dialog font

CRengBox

Version: 1

Super Classes: CDialog

Location: C:\PETEPROJ\REUSE\TESTREP\ABOUTBOX.H

Overview

CRengBox dialog. Version Number = 1 This dialog is used to ask the user if they wish to start the appropriate visualisation program for the processing that has just been carried out.

Services

Public Members

CRengBox(CWnd* pParent = NULL)

standard constructor with no member initialisation.

Protected Members

virtual void DoDataExchange(CDataExchange* pDX)

DDX/DDV support. This method is controlled by the VC++ Class Wizard.

virtual BOOL OnInitDialog()

Initialises the dialog with the name of the source/make file which has just been processed.

AFX_MSG

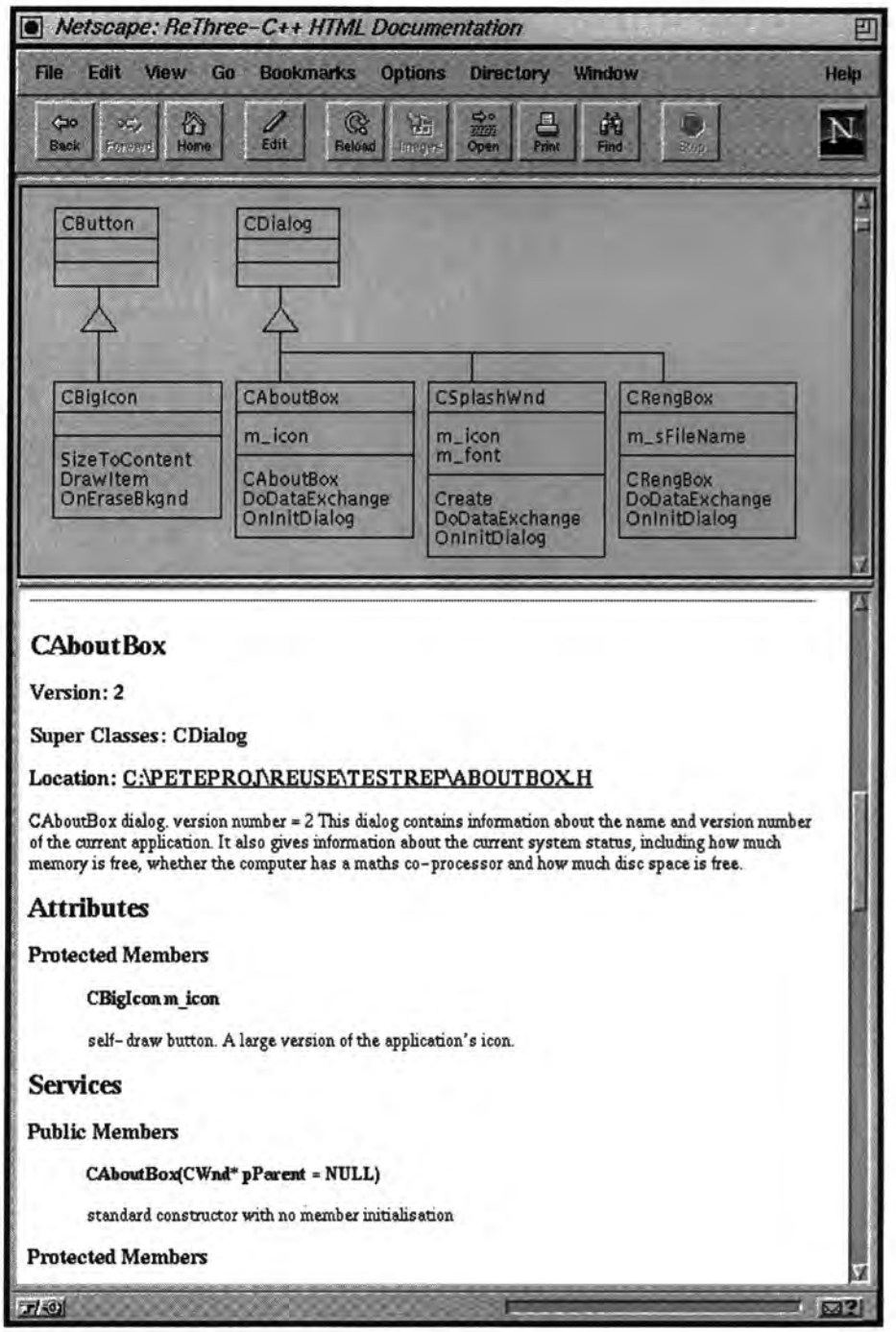
Attributes

Public Members

CString m_sFileName

AFX_DATA

B2.4 HTML and Java Web Page



The Java applet, which draws the class hierarchy diagram shown in the upper frame, is a clickable, client-side image map, which allows the user to click on any of the classes shown. This causes the lower frame to find and display the documentation for that class. The HTML documentation in the lower frame is almost exactly the same as the RTF documentation shown

in B2.3. The only differences are that in the HTML version, the attributes are listed before the services of a class, and that there are links from the documentation to the source code where the location of the source is specified.

Appendix C

C1. ReThree-C++ Evaluation Questionnaire

The information contained in this questionnaire will be used when evaluating the use of ReThree-C++ as a reuse support tool. The results of the questionnaire will be used in academic research, so please be as open and honest as possible.

When finished, please send to: Peter Biggs, Dept. of Computer Science, University of Durham, Durham, DH1 3LE, England.

1. Ease of use

a. How easy was it to set ReThree-C++ up for use?

Easy **1 2 ③ 4 5 6 7 8 9 10** Hard

b. How easy is it to use ReThree-C++ to process files?

Easy **1 2 ③ 4 5 6 7 8 9 10** Hard

c. How quickly did you learn to use ReThree-C++?

Very Quickly **1 ② 3 4 5 6 7 8 9 10** Very Slowly

d. How often did you use the ReThree-C++ Help file?

Never **1 2 3 4 5 6 7 ⑧ 9 10** Very Frequently

e. How helpful is the ReThree-C++ Help file?

No Help **1 2 3 4 5 ⑥ 7 8 9 10** Very Helpful

2. Using ReThree-C++ to generate information about C++ files

a. Which feature of ReThree-C++ did you use most?

Generating Documentation for Code ✓ **Generating Web pages**

b. How helpful were the results of ReThree-C++ in understanding C++ code?

No Help **1 2 3 4 5 6 ⑦ 8 9 10** Very Helpful

c. Which feature of ReThree-C++ did you find most helpful when trying to understand C++ code?

Generating Documentation for Code ✓ **Generating Web pages**

d. Please comment on any experiences in processing C++ code with ReThree-C++:

Quality depends very much on standard of documentation throughout the code - useful in forcing programmer to tidy the code and add comments etc.
Much of the output describes classes that are not necessarily implemented in a particular build of the target application. This is because ReThreeC++ only examines the header files. I think there is a great potential for developing the Web output further, by improving the graphical representation, and allowing for more interaction with the various objects in the class diagram.

3. Reusing with ReThree-C++

a. Did you use the reuse repository facilities of ReThree-C++? (YES) NO

b. If so, approximately how many classes did you add to your repositories?

~50

c. How easy was it to find appropriate classes when searching a reuse repository?

Easy 1 2 3 (4) 5 6 7 8 9 10 Hard

d. How well did you understand how to use the class, once found, using ReThree-C++'s processing options?

Didn't understand 1 2 3 (4) 5 6 7 8 9 10 Understood very quickly

(Due to lack of documentation in source code)

e. How many times have you reused a class found and understood using ReThree-C++'s reuse repositories?

0

4. Further comments

Please add any further comments about ReThree-C++

I was not able to test ReThreeC++ on a live development project, only on old C++ code. However, I expect to use it soon when code is ported to 32 bit platform. I believe it will prove to be very useful.

Appendix D

D1. Group Task Descriptions

C++ Experiment

Group 1

Welcome to the C++ experiment. In the next hour, you will be asked to complete a working version of the attached C++ program. You may use the two reference books provided. Please do not use your own reference books, as this will affect the results of the experiment.

Group 2

Welcome to the C++ experiment. In the next hour, you will be asked to complete a working version of the attached C++ program. You may use the three reference books provided. Please do not use your own reference books, as this will affect the results of the experiment.

Group 3

Welcome to the C++ experiment. In the next hour, you will be asked to complete a working version of the attached C++ program. You may use the three reference books provided, as well as the class reference materials attached. Please do not use your own reference books, as this will affect the results of the experiment.

Group 4

Welcome to the C++ experiment. In the next hour, you will be asked to complete a working version of the attached C++ program. You may use the three reference books provided, as well as the ReThree-C++ tool running on your machine. Details of how to use the tool are attached. Please do not use your own reference books, as this will affect the results of the experiment.

You will be using Visual C++ to create a QuickWin application. This is very similar to writing C++ for the GCC or G++ compiler. Use the project menu to compile, build and execute your program.

Before you leave, please fill in the questions at the bottom of this page. Please be as honest as possible, as the responses will be used when evaluating the experiment.

Questions

1. How long did you spend reviewing C++ in preparation for this experiment?

Time:

2. How difficult did you find writing this program?

Easy **1 2 3 4 5 6 7 8 9 10** Hard

3. How useful did you find the reference materials provided?

No Use **1 2 3 4 5 6 7 8 9 10** Very Useful

4. How would you now rate your C++ skills?

Poor **1 2 3 4 5 6 7 8 9 10** Excellent

D2. Test Program

A program has generated a file which contains a list of file names (each on a separate line) followed by a search term (on the last line of the file). The programmers want to search all of the specified files for occurrences of the search term. They cannot specify how many files are to be searched each time their program is run. They want to automate the searching process with a program which gives the following output:

"Search term" appears in 'file name' <no. of occurrences> times.

For example:

"int count" appears in 'test1.cpp' 12 times.

Write a program to do this using C++.

Programming Tips

It is suggested that you open the file TEST1.DAT and read in the contents a line at a time. Save the list of file names read in (remembering to strip out any unnecessary characters such as spaces and new lines) until the end of the file is reached.

Then take the last item of the list as the search term, open each of the search files in turn, read in from the search file and see if the search term appears. If it does, increment the count. When the end of the search file has been reached, write out how many occurrences of the search term were found.

If any files cannot be opened, give an appropriate error message.

D3. Class Information for Group 3

CStdioFile

Super Classes: CFile

Location: c:\petetest\mfch\afx.h

Overview

raw binary file CStdioFile A CStdioFile object represents a C run-time stream file as opened by the fopen function. Stream files are buffered and can be opened in either text mode (the default) or binary mode. Text mode provides special processing for carriage return-linefeed pairs. When you write a newline character (0x0A) to a text mode CStdioFile object, the byte pair (0x0A,0x0D) is sent to the file. When you read, the byte pair (0x0A,0x0D) is translated to a single 0x0A byte. Several CFile member functions are over-ridden for this derived class. The CFile functions Duplicate, LockRange and UnlockRange are not implemented for CStdioFile. For examples of using this class, see the Class Library Reference for the Microsoft Foundation Class Library. To access CStdioFile, you must: #include <afx.h>

Services

Public Members

CStdioFile()

Constructors Standard constructor

CStdioFile(FILE* pOpenStream)

Constructor given a file pointer returned by a call to the C run-time function fopen.

CStdioFile(const char* pszFileName, UINT nOpenFlags)

Constructor given a string that is the path of the desired file, which may be relative or absolute. nOpenFlags specifies the sharing and access modes. These can be combined using the bitwise-OR (|) operator. See the CFile constructor for a list of mode options.

virtual void WriteString(LPCSTR lpsz)

writes a string to the file, like "C" fputs. The terminating null character ('') is not written to the file. lpsz specifies a pointer to a buffer containing a null terminated text string. Any newline character in lpsz is written to the file as a carriage return-linefeed pair.

virtual LPSTR ReadString(LPSTR lpsz, UINT nMax)

Reads text data into a buffer, up to a limit of nMax-1 characters (like "C" fgets). Reading is stopped by a carriage return-linefeed pair. If, in that case, fewer than nMax-1 characters have been read, a newline character is stored in the buffer. A null character ('') is appended in either case. lpsz is a pointer to a user-supplied buffer that will receive a null-terminated text string. ReadString returns a pointer to the buffer containing the text data, or NULL if the end-of-file was reached.

virtual ~ CStdioFile()

Destructor. Closes the file before destroying this object.

void Dump(CDumpContext& dc)

virtual DWORD GetPosition()

Over-ridden member function - see CFile for details.

virtual BOOL Open(const char* pszFileName, UINT nOpenFlags, CFileException* pError = NULL)

Over-ridden member function - see CFile for details.

virtual UINT Read(void FAR* lpBuf, UINT nCount)

Over-ridden member function - see CFile for details.

virtual void Write(const void FAR* lpBuf, UINT nCount)

Over-ridden member function - see CFile for details.

virtual LONG Seek(LONG lOff, UINT nFrom)

Over-ridden member function - see CFile for details.

virtual void Abort()

Over-ridden member function - see CFile for details.

virtual void Flush()

Over-ridden member function - see CFile for details.

virtual void Close()

Over-ridden member function - see CFile for details.

virtual CFile* Duplicate()

Unsupported

virtual void LockRange(DWORD dwPos, DWORD dwCount)

Unsupported

virtual void UnlockRange(DWORD dwPos, DWORD dwCount)

Unsupported

Attributes

Public Members

FILE* m_pStream

stdio FILE m_hFile from base class is _fileno(m_pStream)

CString

Location: c:\petetest\mfch\afx.h

Overview

Non CObject classes Class CString A CString object consists of a variable-length sequence of characters. The CString class provides a variety of functions and operators that manipulate CString objects, making CString objects easier to use than ordinary character arrays. The maximum size of a CString object is MAXINT (32,767) characters. The CString class has built-in memory allocation capability. This allows string objects to grow as a result of concatenation operations. The overloaded const char* conversion operator allows CString objects to be freely substituted for character pointers in function calls. For examples of using this class, see the Class Library Reference for the Microsoft Foundation Class Library. To access CString, you must: #include <afx.h>

Services

Public Members

CString()

Standard constructor

CString(const CString& stringSrc)

construct from current CString

CString(char ch, int nRepeat = 1)

construct from a single character to be repeated n times

CString(const char* psz)

construct from a pointer to an array of characters

CString(const char* pch, int nLength)

construct from a pointer to an array of characters of length nLength

~CString()

Destructor. Releases allocated memory used to store the string's character data

int GetLength()

Returns the number of characters in this CString object (not including the null terminator)

BOOL IsEmpty()

Tests a CString object for the empty condition. Returns 0 if empty, non-zero otherwise

void Empty()

Makes this CString object an empty string and frees memory as appropriate

char GetAt(int nIndex)

Returns a single character specified by an index number. nIndex is a 0 based index of the character in the CString object

char operator[](int nIndex)

same as GetAt

void SetAt(int nIndex, char ch)

Overwrites a single character specified by an index number. SetAt will not enlarge the string if the index exceeds the bounds of the existing string. nIndex is a 0 based index of the character in the CString object

operator const char*()

casts the CString object as a string pointer.

const CString& operator=(const char* psz)

reinitialises the CString object with its new value the same as psz

const CString& operator+=(const char* psz)

joins a copy of psz on to the end of this CString object

friend CString AFXAPI operator+(const CString& string1, const CString& string2)

adds two CString objects

int Compare(const char* psz)

Compares this CString object with another string, character by character. Returns 0 if the strings are identical, -1 if this CString object is less than psz or 1 if this CString object is greater than psz

int CompareNoCase(const char* psz)

Compares this CString object with another string, character by character, ignoring case Returns 0 if the strings are identical, -1 if this CString object is less than psz or 1 if this CString object is greater than psz

int Collate(const char* psz)

Performs a locale specific comparison of two strings. Returns 0 if the strings are identical, -1 if this CString object is less than psz or 1 if this CString object is greater than psz

CString Mid(int nFirst, int nCount)

Extracts a substring of length nCount characters from this CString object, starting at position nFirst (zero-based). The function returns a copy of the extracted substring.

CString Mid(int nFirst)

Extracts a substring from this CString object, starting at position nFirst (zero-based), extracting the remainder of the string. The function returns a copy of the extracted substring.

CString Left(int nCount)

Extracts the first (that is, leftmost) nCount characters from this CString object and returns a copy of the extracted substring. If nCount exceeds the string length, then the entire string is extracted.

CString Right(int nCount)

Extracts the last (that is, rightmost) nCount characters from this CString object and returns a copy of the extracted substring. If nCount exceeds the string length, then the entire string is extracted.

CString SpanIncluding(const char* pszCharSet)

Extracts the largest substring that contains only the characters in the specified set pszCharSet; starts from the first character in this CString object. If the first character of the string is not in the character set, then SpanIncluding returns an empty string

CString SpanExcluding(const char* pszCharSet)

Extracts the largest substring that excludes only the characters in the specified set pszCharSet; starts from the first character in this CString object. If the first character of the string is in the character set, then SpanExcluding returns an empty string

void MakeUpper()

Converts this CString object to an uppercase string

void MakeLower()

Converts this CString object to a lowercase string

void MakeReverse()

Reverses the order of the characters in this CString object

int Find(char ch)

Searches this string for the first match of the character ch. Returns the zero-based index of the first character in this CString object that matches the requested character; -1 if the character is not found

int ReverseFind(char ch)

Searches this string for the last match of the character ch. Returns the zero-based index of the last character in this CString object that matches the requested character; -1 if the character is not found

int FindOneOf(const char* pszCharSet)

Searches this string for the first character that matches any character contained in pszCharSet. Returns the zero-based index of the first character in this CString object that is also in pszCharSet; -1 if there is no match

int Find(const char* pszSub)

Searches this string for the first match of the substring pszSub. Returns the zero-based index of the first character in this CString object that matches the requested substring; -1 if the substring is not found

char* GetBuffer(int nMinBufLength)

void ReleaseBuffer(int nNewLength = -1)

char* GetBufferSetLength(int nNewLength)

int GetAllocLength()

Protected Members

void Init()

void AllocCopy(CString& dest, int nCopyLen, int nCopyIndex, int nExtraLen)

void AllocBuffer(int nLen)

void AssignCopy(int nSrcLen, const char* pszSrcData)

void ConcatCopy(int nSrc1Len, const char* pszSrc1Data, int nSrc2Len, const char* pszSrc2Data)

void ConcatInPlace(int nSrcLen, const char* pszSrcData)

static void SafeDelete(char* pch)

static int SafeStrlen(const char* psz)

Attributes

Protected Members

char* m_pchData

actual string (zero terminated)

int m_nDataLength

does not include terminating 0

int m_nAllocLength

does not include terminating 0

CStringArray

Super Classes: CObject

Location: c:\petetest\mfch\afxcoll.h

Overview

CStringArray The CStringArray class supports arrays of CString objects. The string arrays are similar to C arrays but they can dynamically shrink and grow as necessary. Array indexes always start at position 0. You can decide whether to fix the upper bound or allow the array to expand when you add elements past the current bound. Memory is allocated contiguously to the upper bound, even if some elements are null. For examples of using this class, see the Class Library Reference for the Microsoft Foundation Class Library entry for CObArray. To access CStringArray, you must: #include <afxcoll.h>

Services

Public Members

CStringArray()

Construction Constructs an empty CString pointer array. The array grows one element at a time.

int GetSize()

Returns the size of the array. Since indexes are zero-based, the size is 1 greater than the largest index.

int GetUpperBound()

Returns the current upper bound of this array. Because array indexes are zero-based, this function returns a value 1 less than GetSize. Returns -1 when the array contains no elements.

void SetSize(int nNewSize, int nGrowBy = -1)

Establishes the size of an empty or existing array; allocates memory if necessary. If the new size is smaller than the old size, then the array is truncated and all unused memory is released. nNewSize is the new array size (number of elements). Must be greater than or equal to 0. nGrowBy is the minimum number of element slots to allocate if a size increase is necessary.

void FreeExtra()

Frees any extra memory that was allocated while the array was grown. This function has no effect on the size or upper bound of the array.

void RemoveAll()

Removes all the pointers from this array and deletes the CString objects. If the array is already empty, the function still works. The RemoveAll function frees all memory used for pointer storage.

CString GetAt(int nIndex)

Returns the array element at the specified index; NULL if no element is stored at the index.

void SetAt(int nIndex, const char* newElement)

Sets the array element at the specified index. SetAt will not cause the array to grow. Use SetAtGrow if you want the array to grow automatically.

CString& ElementAt(int nIndex)

Returns a temporary reference to the element pointer within the array. It is used to implement the left-side assignment operator for arrays. Note that this is an advanced function that should be used only to implement special array operators. Returns a reference to a CString pointer.

void SetAtGrow(int nIndex, const char* newElement)

Sets the array element at the specified index. The array grows automatically if necessary (that is, the upper bound is adjusted to accommodate the new element).

int Add(const char* newElement)

Adds a new element to the end of an array, growing the array by 1. If SetSize has been used with an nGrowBy value greater than 1, then extra memory may be allocated. However, the upper bound will increase by only 1.

CString operator[](int nIndex)

CString& operator[](int nIndex)

These subscript operators are a convenient substitute for the SetAt and GetAt functions. The first operator, invoked for arrays that are not const, may be used on either the right (r-value) or the left (l-value) of an assignment statement. The second, invoked for const arrays, may be used only on the right. The Debug version of the library asserts if the subscript (either on the left or right side of an assignment statement) is out of bounds.

void InsertAt(int nIndex, const char* newElement, int nCount = 1)

This version of InsertAt inserts one element (or multiple copies of an element) at a specified index in an array. In the process, it shifts up (by incrementing the index) the existing element at this index, and it shifts up all the elements above it. nCount is the number of times this element should be inserted (defaults to 1).

void RemoveAt(int nIndex, int nCount = 1)

Removes one or more elements starting at a specified index in an array. In the process, it shifts down all the elements above the removed element(s). It decrements the upper bound of the array but does not free memory. nCount is the number of elements to remove. If you try to remove more elements than are contained in the array above the removal point, then the Debug version of the library asserts.

void InsertAt(int nStartIndex, CStringArray* pNewArray)

This version inserts all the elements from another CStringArray collection, starting at the nStartIndex position. The SetAt function, in contrast, replaces one specified array element and does not shift any elements.

~CStringArray()

CStringList

Super Classes: CObject

Location: c:\petetest\mfch\afxcoll.h

Overview

CStringList The CStringList class supports lists of CString objects. All comparisons are done by value, meaning that the characters in the string are compared instead of the addresses of the strings. For examples of using this class, see the Class Library Reference for the Microsoft Foundation Class Library entry for CObList. To access CStringList, you must: #include <afxcoll.h>

Services

Public Members

CStringList(int nBlockSize=10)

Constructs an empty list for CString objects.

int GetCount()

Gets the number of elements in this list.

BOOL IsEmpty()

Indicates whether this list contains no elements. Returns TRUE if the list is empty, FALSE otherwise.

CString& GetHead()

CString GetHead()

Gets the CString pointer that represents the head element of this list. You must ensure that the list is not empty before calling GetHead. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements. If the list is accessed through a pointer to a const CStringList, then GetHead returns a CString pointer. This allows the function to be used only on the right side of an assignment statement and thus protects the list from modification. If the list is accessed directly or through a pointer to a CStringList, then GetHead returns a reference to a CString pointer. This allows the function to be used on either side of an assignment statement and thus allows the list entries to be modified.

CString& GetTail()

CString GetTail()

Gets the CString pointer that represents the tail element of this list. You must ensure that the list is not empty before calling GetTail. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements. If the list is accessed through a pointer to a const CStringList, then GetHead returns a CString pointer. This allows the function to be used only on the right side of an assignment statement and thus protects the list from modification. If the list is accessed directly or through a pointer to a CStringList, then GetHead returns a reference to a CString pointer. This allows the function to be used on either side of an assignment statement and thus allows the list entries to be modified.

CString RemoveHead()

Removes the element from the head of the list and returns a pointer to it. You must ensure that the list is not empty before calling RemoveHead. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

CString RemoveTail()

Removes the element from the tail of the list and returns a pointer to it. You must ensure that the list is not empty before calling RemoveTail. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use IsEmpty to verify that the list contains elements.

POSITION AddHead(const char* newElement)

Adds a new element to the head of this list. The list may be empty before the operation.

POSITION AddTail(const char* newElement)

Adds a new element to the tail of this list. The list may be empty before the operation.

void AddHead(CStringList* pNewList)

Adds a list of elements to the head of this list. The list may be empty before the operation.

void AddTail(CStringList* pNewList)

Adds a list of elements to the tail of this list. The list may be empty before the operation.

void RemoveAll()

Removes all the elements from this list and frees the associated CStringList memory. No error is generated if the list is already empty.

POSITION GetHeadPosition()

Gets the position of the head element of this list. Returns a POSITION value that can be used for iteration or object pointer retrieval; NULL if the list is empty.

POSITION GetTailPosition()

Gets the position of the tail element of this list; NULL if the list is empty. Returns a POSITION value that can be used for iteration or object pointer retrieval; NULL if the list is empty.

CString& GetNext(POSITION& rPosition)

CString GetNext(POSITION& rPosition)

Gets the list element identified by rPosition, then sets rPosition to the POSITION value of the next entry in the list. You can use GetNext in a forward iteration loop if you establish the initial position with a call to GetHeadPosition or Find. You must ensure that your POSITION value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts. If the retrieved element is the last in the list, then the new value of rPosition is set to NULL. rPosition is a reference to a POSITION value returned by a previous GetNext, GetHeadPosition, or other member function call.

CString& GetPrev(POSITION& rPosition)

CString GetPrev(POSITION& rPosition)

Gets the list element identified by rPosition, then sets rPosition to the POSITION value of the previous entry in the list. You can use GetPrev in a reverse iteration loop if you establish the initial position with a call to GetTailPosition or Find. You must ensure that your POSITION value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts. If the retrieved element is the first in the list, then the new value of rPosition is set to NULL. rPosition is a reference to a POSITION value returned by a previous GetPrev or other member function call.

CString& GetAt(POSITION position)

CString GetAt(POSITION position)

A variable of type POSITION is a key for the list. It is not the same as an index, and you cannot operate on a POSITION value yourself. GetAt retrieves the CString pointer associated with a given position. You must ensure that your POSITION value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts. position is a POSITION value returned by a previous GetHeadPosition or Find member function call.

void SetAt(POSITION pos, const char* newElement)

A variable of type POSITION is a key for the list. It is not the same as an index, and you cannot operate on a POSITION value yourself. SetAt writes the CString pointer to the specified position

in the list. You must ensure that your POSITION value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts. pos is the POSITION of the element to be set. newElement is the CString pointer to be written to the list.

void RemoveAt(POSITION position)

Removes the specified element from this list. You must ensure that your POSITION value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts. position is the position of the element to be removed from the list. inserting before or after a given position

POSITION InsertBefore(POSITION position, const char* newElement)

Adds an element to this list before the element at the specified position. Returns a POSITION value that can be used for iteration or object pointer retrieval; NULL if the list is empty. newElement is the object pointer to be added to this list.

POSITION InsertAfter(POSITION position, const char* newElement)

Adds an element to this list after the element at the specified position. position as a POSITION value returned by a previous GetNext, GetPrev, or Find member function call. newElement is the object pointer to be added to this list.

POSITION Find(const char* searchValue, POSITION startAfter = NULL)

Searches the list sequentially to find the first CString matching the specified CString. Defaults to starting at the HEAD of the list. Returns NULL if not found

POSITION FindIndex(int nIndex)

Uses the value of nIndex as an index into the list. It starts a sequential scan from the head of the list, stopping on the nth element. nIndex is the zero-based index of the list element to be found. Returns a POSITION value that can be used for iteration or object pointer retrieval; NULL if nIndex is negative or too large.

~CStringList()

void Serialize(CArchive&)

void Dump(CDumpContext&)

void AssertValid()

Protected Members

struct CNode(CStringList)

CNode* NewNode(CNode*, CNode*)

void FreeNode(CNode*)

Attributes

Protected Members

CNode* m_pNodeHead

CNode* m_pNodeTail

int m_nCount

CNode* m_pNodeFree

m_pBlocks

int m_nBlockSize

D4. Instructions on the use of ReThree-C++ for Group 4

ReThree-C++

ReThree-C++ is an integrated reverse engineering, re-documentation and reuse tool set. It can be used to extract information from C++ source code, and to create a repository of C++ classes for later retrieval. The tool set can be divided into three main functions:

1. Automatically reverse engineering C++ source code to give a visual class hierarchy representation in OMT object model format.

2. Documenting C++ source code, based on the comments contained within the code, to provide automatically generated software documentation.

3. Building, maintaining and searching a reuse repository of C++ classes which can be re-used in later applications.

This version of ReThree-C++ has a reuse repository open which contains information about the Microsoft® Foundation Classes that are available for reuse within a C++ program.

When you wish to search for a class to reuse from the currently open repository, choose the **SEARCH** menu item from the **REUSE MENU**. You will then be asked for a search term with which to search the repository. You can use *wildcard characters* (*) and *boolean operators* (&, AND, |, OR, !, NOT) in the search term.

If any classes match the criteria specified, a list of these classes will be displayed in a dialog. The Search Results dialog displays a list of the classes that matched the search criteria specified, along with an

overview of the class. The dialog allows you to view information about any of the classes. Select a class in the list box, and you will see the class overview in the box below

If you want more information about the class, use one of the three buttons on the right of the dialog to view either a class hierarchy (using OMTool), documentation (using Word) or both (using Java compatible Netscape).

View Hierarchy

This button processes the file which contains your selected class and displays a class hierarchy based around that class using OMTool to display the class hierarchy diagram.

View Documentation

This button processes the file which contains your selected class and generates documentation for the class, and any specified associated classes, which can be viewed using Word for Windows®, or other RTF compatible application.

View Web Page

This button processes the file which contains your selected class and generates a class hierarchy and documentation for the class, and any specified associated classes, which can be viewed using Netscape 2.x, or other Java compatible Web browser (Java generated class hierarchies are only available when running ReThree-C++ under Windows® 95).