Effective component-based solutions to problems: reusing components and designs

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:


Metadata Record: [https://dspace.lboro.ac.uk/2134/34880](https://dspace.lboro.ac.uk/2134/34880)

Publisher: © Loughborough University

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) licence. Full details of this licence are available at: [https://creativecommons.org/licenses/by-nc-nd/4.0/](https://creativecommons.org/licenses/by-nc-nd/4.0/)

Please cite the published version.
University Library

Author/Filing Title: Hornsby, Peter

Class Mark: .................................................................

Please note that fines are charged on ALL overdue items.

FOR REFERENCE ONLY
Effective component-based solutions to problems: Reusing components and designs

By
Peter Hornsby

A doctoral thesis submitted in partial fulfilment of the requirements for the award of the degree of Doctor of Philosophy of Loughborough University

January 2003

Supervisor: Dr Ian Newman
Department of Computer Science
© Loughborough University
Acknowledgements

There are a number of people whom I would like to thank for the help, advice and (in some cases) soft, water absorbent shoulders offered during this research and in the years following.

My heartfelt thanks are extended to my supervisor, Dr Ian Newman, for his help and guidance throughout the research, as well as for the many interesting (and in most cases, tangential) conversations we have had.

I would like to express my gratitude to my Director of Research, Dr Chris Hinde, for his assistance over the ‘long haul’ and (equally tangential) conversations.

Thanks too to Dr Russel Winder for being my external examiner; it was a long and difficult road, but worth it.

I would like to thank Dr And Kemp, for his tuition and friendship over many years, but particularly for helping me develop my ideas during this work.

None of this would have been possible without the love and support of my family, Sarah and Louisa. To Mum, Dad, Gerard, Sarah and Louisa, I could not have done this without you.

Finally, to all the friends who, in so many ways have been there for me, thank you.
“...students arrive from school confident that they know very nearly everything, and they leave years later certain that they know practically nothing. Where did the knowledge go in the meantime? Into the University, of course, where it is carefully dried and stored.”

-- Pratchett et al., 2000

“No man of intelligence will venture to express his philosophical views in language, especially not in language that is unchangeable, which is true of that which is set down in written characters.”

-- Plato, Seventh Letter
Abstract

Computers are useful problem-solving tools, and they are most effective when they are programmed to address a particular problem. Programming is however an activity that is restricted to a very small group of specialists, usually with years of training. Within this specialism, component reuse is regarded as an important technique, but one that is difficult to achieve in practice. The existing development community has already invested considerable time and money in learning software development skills, and is unlikely to invest further in learning a significantly different skill. It seems reasonable therefore that effective techniques for component reuse will need to be based on existing skills, and must keep the additional workload of component reuse as small as possible. The work described in this thesis is an investigation of techniques which might meet this requirement and which are based on an understanding of the holistic human-computer problem solving system. Here, both the requirements of the computer as an information processing system, and the needs of the human problem solver are accounted for and enabled to work together effectively.

The approach adopted was a series of experiments, each of which explores some aspect of the interface between human understanding and possible technologies for improving reuse. Several useful results were obtained. Evidence gathered for this thesis indicates that the background of users influences the problem solving strategy adopted. In addition, experimental support is provided for the assertion that human problem solvers move rapidly and frequently between different levels of abstraction in the process of problem solving (refining their understanding of both the problem and the solution). Empirical evidence has been provided to show that software developers are able to understand UML in sufficient detail for it to be a useful language for describing software designs. The tools that have been produced for storing and retrieving components have demonstrated that it is possible to automatically store large numbers of components for later retrieval. It has also been demonstrated that components may be associated with multiple descriptions to increase their potential reusability in different contexts and for different users.
The approach used in the final prototype system, DesignMatcher, builds on the earlier research tools. This provides the developer with an automatically populated store of designs which are compared dynamically with the developers' ongoing work, during the design process. The design process explicates the goals and implementation of the system in a form that can be processed by both the computer and the user, providing a shared mapping between the real-world problem and a possible solution. DesignMatcher supports the microscopic iteration between problem and solution made by human problem solvers but poorly supported by current development tools and techniques. DesignMatcher uses familiar interface elements and integrates with existing tools in order to reduce the overhead associated with its use. By building upon a clear understanding of fundamental human problem solving and software development skills, the DesignMatcher approach does not add significantly to the existing workload, making the development process more efficient. A preliminary evaluation has shown that the DesignMatcher approach is acceptable to developers and can be a useful tool in the production of effective component-based solutions to problems.
13.3.2. Adding components to the database .............................................................. 165
13.3.3. Matching designs ........................................................................................... l66
13.4. Implementation.................................................................................................. 167
13.4.1. Design capture ............................................................................................... l67
13.4.2. Matching designs ........................................................................................... l68
13.5. System initialisation ......................................................................................... 170
13.6. Example of use ................................................................................................. 170
13.7. DesignMatcher client interface ......................................................................... 172
13.8. Conclusions ....................................................................................................... 174
14. DesignMatcher evaluation ..................................................................................... 175
14.1. Introduction ...................................................................................................... 175
14.2. Subject ............................................................................................................. 175
14.3. Method ............................................................................................................. 176
14.4. DesignMatcher use: analysis .......................................................................... 177
14.5. Discussion ....................................................................................................... 183
14.6. Conclusions ..................................................................................................... 184
15. The case for the DesignMatcher approach ........................................................... 185
15.1. Introduction ..................................................................................................... 185
15.2. DesignMatcher as an information retrieval system ........................................... 185
15.2.1. DesignMatcher performance ....................................................................... 186
15.3. DesignMatcher in use by individual problem solvers in an organisational context 189
15.4. Conclusions .................................................................................................... 190
16. Conclusions and further research ......................................................................... 192
16.1. Introduction .................................................................................................... 192
16.2. Further research .............................................................................................. 194
16.3. Conclusion ...................................................................................................... 195
17. References ......................................................................................................... 197
18. Appendix A: UNIX scripting experiment results ................................................... 216
18.1. Subject A: 2nd year English undergraduate ................................................... 216
18.2. Subject B: 2nd year human biology undergraduate .......................................... 218
18.3. Subject C: 3rd (final) year maths and physics undergraduate ........................ 220
18.4. Subject D: 2nd year computing undergraduate ................................................ 221
18.5. Subject E: 2nd year physics undergraduate ...................................................... 222
18.6. Subject F: 3rd year computing Ph.D. ............................................................... 223
1. Introduction

Problem solving is an activity that is central to human survival, and computers have the potential to be powerful assistants in this, as information retrieval and processing tools. However, programmers are faced with a difficult and complex task when creating software systems. From a usability perspective, they must satisfy the potentially conflicting goals of reliability, accessibility (to different user populations), and flexibility (to different problems). From a technical perspective, questions of extensibility, compatibility and reusability must be addressed. Commercial pressures also influence the development of software, with the need to fulfil (often-conflicting) goals of rapid development and maintainability. Finally, developers working within groups must coordinate their problem-solving activities.

Software development is thus a task that presents considerable challenges. Nonetheless, the importance of this task makes it critical that it is performed effectively, and for this, the support tools and environment used during the task are vital. These are particularly important given the increasing amount of computer programming that is being done by people who are not trained programmers as a support activity to their main task. The aim of this research was to produce tools that enable the task of software development to be addressed more effectively than has hitherto been possible, by users from a range of backgrounds, through the reuse of existing design and implementation artefacts.

This thesis views programming as a specialised form of human problem solving, and as such, has considered research into the problem solving activity, and particularly, the role of computers. This has led to a broad-based approach being adopted, which has examined a range of related research areas.

Computer Science is fundamentally a practical discipline, and as such, it is important to produce proof-of-concept tools to demonstrate the principles that are proposed. This approach was adopted in the work described in this thesis, and accordingly, a strategy of
iterative development was used. In addition to providing first-hand experience of the issues involved in software development, this enabled feedback to be obtained regarding the probable validity of the approach for specialists in software development and other fields. Most importantly, this approach has enabled the development of each prototype to guide the design and development of the next.

This thesis is divided into two parts. The first part presents the background information for the research, beginning with a discussion of human problem solving, and going on to examine the role of the computer as a problem solving tool. This includes existing systems development approaches and paradigms, end user programming techniques, and the role of component technologies. The underlying theme is the practical effectiveness that these technologies have in improving the process of generating solutions to problems.

Chapter 8 sets out the conclusions that were drawn from the work that was examined and proposes a methodology and a strategy for the work reported in the second (experimental) part of the thesis, which describes the implementation of the work programme. This resulted in three empirical studies and the production of three research systems, each of which is described in a separate chapter. Finally, the conclusions that can be drawn from the research are presented, together with future work to be conducted.

This thesis presents a novel approach to component reuse based on design information as well as code, which appears to be both theoretically and practically sound. The approach promotes design, has a minimal additional workload for the developer, and offers considerable potential to improve the effectiveness of the programming process, for both novices and experienced programmers. As such, it is believed to present a useful and significant contribution to the field.
"Everything is deeply intertwined."

- Theodor Holm Nelson

2. Human Problem Solving

2.1. Introduction

One approach to understanding how and why difficulties arise in using computers as tools in solving problems is to perform an examination of the psychological aspects of the task. Since the goal of this work is to make computers more accessible as problem solving tools, this chapter examines how problem solving is performed in both general terms and in the specific context of programming. A general overview of psychological research into human problem solving is presented, before specific areas are addressed later in the thesis when they are relevant to a particular aspect of the problem.

2.2. Human Problem Solving

A problem is generally understood to be a goal which cannot be achieved directly, and problem solving can thus be described as action taken in pursuit of this goal (Kahney, 1986). A significant amount of research into human problem solving has been done by cognitive scientists, viewing the human as an information processor, similar in many ways to the (Turing) Machine (Figure 2.1) proposed by Turing (1936).
The Turing Machine executes a series of transitions, which are determined by the transition table and the characters on the tape. The transition table is finite, whereas the tape is infinite (classically, the Turing Machine is thought of as having the tape bounded on the left, but extending infinitely to the right). For each transition, the machine checks the state it is currently in, and the character that is written on the tape below the machine head. Based on this, it changes to a new state, writes a new character on the tape, and moves the machine head one space left or right. The machine stops after entering the special 'HALT' state. By changing the machine table and the tape, the behaviour of the machine can be changed, and thus a Universal Turing Machine, a machine capable of solving any computable problem, may be postulated. While the Universal Turing Machine is in no way a physical analogue of the human brain, it is of interest in cognitive science since it:

- provides a complete model of what information processing is all about;
- is in principle capable of answering psychologically interesting questions; and
- provides an example of how a finite device can generate an infinite variety of behaviour (Dawson, 1998.)

The concept of a finite machine generating an infinite variety of behaviour is a difficult one, but can be understood in that although the machine itself is finitely defined, it is
allowed an infinite amount of space (the ticker tape) and time within which to work (Hodges, 1983).

In order to describe the nature of problem solving more precisely, a theoretical construct known as a problem space is utilised by cognitive scientists, after Newell (1977), who describes problem solving with regards to this space:

“The elements of this space consist of states of knowledge about the problem... Both the initial situation and the desired situation are represented as elements of this space. A problem space also has associated with it a set of operators, which, when applied to an element of the space, produce new elements.” [original italics]

Cognitive Science characterises problem solving as a search within this problem space. This concept is perhaps best understood through a well-defined problem, such as a game of solitaire, which is played with pegs in a game board. This game can have many possible states, represented by different numbers of pegs in different patterns on the board. The problem solver moves between states through operators, which in the case of solitaire are ‘hops’ over adjacent pegs. Thus, the state of the board after any given move will influence possible future states.

Researchers have mapped the possible problem states for certain well-defined problem domains. Clearly most domains are too large to map completely, such as chess, with over $10^{40}$ possible combinations of pieces (de Groot, 1965). For problems such as these where an exhaustive search of the problem space is impractical, two general strategies can be used to address the problem without this exhaustive search. An appropriate algorithm can guarantee to produce a solution, but in large problem spaces these can take significant amounts of time. More often used by human problem solvers are what are called heuristic methods, which are loosely defined ‘rules of thumb’ that have been developed using previous (and usually personal) experience. These do not guarantee a solution, but are typically easy to apply and fast to use. Human problem solvers nearly always use
heuristics because of the limitations of human cognition. These limitations include the capacity of short-term memory (Miller, 1956), the restricted rate at which the human can react to new data, and the rate at which long term memory can be built up (Baddeley, 1999). For example, a heuristic identified by Newell, Shaw and Simon (1958) was Means-End Analysis (MEA). This is a three-stage heuristic:

1. note the difference between current state and goal state;
2. create a sub-goal to reduce this difference; and
3. select an operator which will achieve this sub-goal.

A problem with MEA arises when it is no longer clear what the sub-goal should be (Gross, 1996). As will be seen in the next section, this is often the case for real-world problems.

2.3. Types of problem

In many cases, it is desirable to be able to classify problems. If a successful classification can be made, a problem solver can use strategies known to be effective for problems of that type and thus allow the problem to be solved more effectively. This strategy may be a well-defined series of steps (an algorithm), or a (less well-defined) heuristic. Cognitive Science breaks problems into two categories: well-defined and ill-defined problems. According to Kahney (1993), in well-defined problems, the solver has four kinds of information available:

1. initial state;
2. goal state;
3. legal operators; and
4. operator restrictions.

Such problems would (for instance) include the chess or solitaire problems presented earlier. It is interesting to note that computer programs imitating human problem solving in some way usually address well-defined problems (Gelernter and Rochester, 1958).
Such problems are comparatively simple to describe for computer processing since the four required types of information are all present. At the opposite end of the spectrum are ill-defined problems. Kahney (1986) defines these as problems that lack some or all of the attributes of well-defined problems. This type of problem is also known as 'wicked' after Rittel and Webber (1984), who described such problems in the context of social planning, such as the construction of a new road, where many (often-conflicting) factors must be balanced. Rittel and Webber outlined a number of criteria that define wicked problems, including:

- no definite problem formulation;
- no immediate and no ultimate test of a solution; and
- each problem is unique.

As a strategy for coping with the uncertainty inherent in problems of this type, problem solvers make assumptions or use domain (that is, subject-specific) knowledge to clarify what is needed at different stages of the problem, while still using a basic MEA solution strategy (Voss et al., 1983). As a result, the accuracy and efficacy of these assumptions can have a major impact on the perceived success of the solution. While Simon (1984) has argued that the distinction between well-defined and ill-defined problems is less clear-cut than the above description implies, the distinction is nonetheless believed to be useful in considering real-world problem-solving situations. Real-world problem solving must often satisfy criteria that are at best indirectly related to the problem at hand. Spending time thinking about problems before starting to create a solution can be a tentative indication of a ‘good’ problem solver, and design processes exemplify this: the importance of design in developing computer software is acknowledged to be a vital part of successful development. Experience can influence the solution strategy adopted, and over time, the experience of problem solvers in different fields has been codified into textbooks, well-defined processes and other resources that may be used to promote ‘good design’. However, as noted earlier, an essential precondition for the successful use of such stored knowledge is an accurate identification of the problem type.
In order to manage problem solving within an organisation, where many problem-solvers are involved, the process may be formalised, which enables criteria such as project milestones to be met. Such a process may however compound difficulties in addressing wicked problems by imposing additional constraints. Thus, it is proposed that the majority of real-world problems, as well as the majority of problems addressed through programming, are ‘wicked’. To illustrate this, Figure 2.2 illustrates how a software designer moves between different aspects of a problem, refining their understanding of what is needed:

![Graph showing a designer solving a lift simulation problem](from Guindon, 1990)

Here, the designer is attempting to create a simulation of a lift. The points on the graph show activity at different levels of abstraction within the problem domain; the connecting line indicates the movement between abstraction levels. The light bulb symbols indicate moments of insight, while the crosses ('+') indicate the identification of new requirements. By comparison, the shaded grey line indicates how the designer could move between the levels of the problem if a top-down process (representative of the formal processes used by
many software development organisations) were used (albeit preceded by a problem specification). Guindon (1990) suggests that the designer is interleaving problem specification with solution development, and notes that:

“These deviations from top-down design appear to be consequences of intrinsic features of design problems – incomplete specification of the problem, lack of a predetermined solution path, and integration of multiple sources of knowledge.”

Such deviation from a conventional view of problem solving is commonplace in human problem solvers, and in accordance with the opportunistic problem-solving behaviour suggested by Green (1989). It may therefore be seen that the designer must address problems of inadequate (initial) knowledge if a solution is to be produced. In software development, the need for this is reflected in the vast range of development strategies available, such as rapid prototyping, exploratory programming and forms of end-user involvement in the process, which enable further information on requirements and technical strategies to be gathered. As noted earlier, problems are not only wicked in their own right; they can also be considered wicked in terms of the organisational context within which the designer works. Curtis and Walz (1990) note that projects must be aligned with company goals and are affected by corporate politics, culture and procedures. Furthermore, the designer is unlikely to be working alone on a project, and group dynamics can affect its progress. As a result, the designer may adopt a particular solution for reasons other than those dictated by the problem. Finally, the designer’s understanding of the nature of the problem is heavily influenced by how it is represented. The next subsection examines the contributions made to problem solving by expertise and representation.

2.4. Problem representations

From the earliest psychological studies, the interdependency between the representation of a problem and the formulation of an effective solution has been recognised. The Gestalt
school of psychology argued that the essence of problem solving was the perceptual restructuring of the problem by the problem solver, leading to insight. Gestalt is a German word with no precise equivalent in English, but which is generally translated to mean ‘form’; that is, the structural characteristics and type of problem. The Gestalt psychologists believed that problem solving proceeded in a series of fixed stages, which Wallas (1926) described as:

1. **Preparation.** The solver has recognised that a problem exists and made some preliminary attempts at understanding and solving the problem.

2. **Incubation.** Assuming these preliminary attempts have failed, the problem solver puts the problem to one side and is assumed to be considering the problem at an unconscious level.

3. **Illumination.** This is the conscious moment of insight produced by the unconscious work.

4. **Verification.** This is usually a simple check to ensure the insight was accurate.

While artists and mathematicians supported this view, it lacks an understanding of what happens during the key problem solving stage of incubation. Despite this lack of understanding regarding incubation, the Gestalt psychologists nevertheless identified that a possible barrier to problem solving was the notion of (mind) set (also known as functional fixedness (Weiten, 1992)). This describes the human tendency for perceptions to be determined by experience. Techniques that work for one problem are often expected to work for another, which is perceived to be (in some way) similar. While this can improve the efficiency of the problem solver (as described earlier), it can also reduce performance if the set is inappropriate to the situation.

Amarel (1968) demonstrated that minor changes in the representation of problems could produce significant improvements in the efficiency with which solutions were found. This improvement was attributed to a reduction in the size of the problem space that must be searched to find a solution:
"...unfortunately, descriptions built of atomic elements have astronomical N-state spaces. Thus, we are confronted with the problem of finding the coarsest possible elements and predicates that can form descriptions that are fine enough for expressing the rules of action in the required detail."

Problem representations therefore differ in terms of their abstraction from the detail of the problem, and this can have a significant impact on the ease with which it may be solved and on the effectiveness of the solution. As Guindon (1990) noted however, problem solvers move between levels of abstraction when addressing different parts of a problem.

2.5. Expertise in problem solving

Expertise in a particular domain can be a significant factor in successful problem solving within that domain. Problem solving in the domain of chess is a popular field of academic study, since chess is a well-defined problem with a broad following. De Groot (1965) found that novice chess players and masters spend the same amount of time searching for moves, but was unable to isolate the factor which meant that the masters automatically explore ‘strong’ moves (those likely to lead to checkmate) in preference to weaker ones. De Groot also found that the masters were able to recall chess positions more effectively than novices, which he attributed to their use of chunking meaningful combinations of pieces. Chunking is a behaviour proposed by Miller (1956), who suggested that the effective capacity of (short term) human memory (which he proposed was seven items, plus or minus two), could effectively be increased by grouping information into meaningful chunks. Organisation by chunking has been demonstrated by experts in other domains, including programming (McKeithen et al., 1981). This study also noted that as expertise in programming grows, programmers move from a chunking scheme whereby classification takes place through natural language mnemonics (for instance forming reserved programming words into a story) through to chunking based on function, such as if-then-else. Chapter 4 will show that the paradigm within which the programming is performed can also influence the chunking strategy. Soloway et al. (1982) state that professional programmers "...have and use high-level, plan knowledge to direct their
programming activities.” It is interesting to note how this parallels the scientific method as outlined by Popper (1972). This proceeds through the collection of data, its examination and the production of hypotheses, and verification. It is also interesting to consider the applicability of an alternative representation of scientific thinking as it is often practised in programming: progression by falsification: the production of a program and the search for bugs. However, solutions to problems are produced in a relatively well-defined (solution) domain, via a programming language. They are based on an understanding of a real-world problem, and if a conventional software design process is undertaken, a requirements capture process is typically performed. The approach suggested by Popper therefore appears to be a more appropriate model for addressing problem solving via the development of software than progression by falsification.

Problem solving with a computer by way of the creation of software is a particularly complex task. The factors influencing the problem-solving domain are shown in Figure 2.3:
This model is reflected in Fischer's (1987) two aspects of software design tasks. The situation model is concerned with understanding the problem, for instance the rationale behind the problem and the constraints upon solving it. The system model is a set of operations that will result in the desired solution. Thus a software design task may be described as a translation from the situation model to the system model, and in the process of creating a software solution, both the situation and system models will change. The successful programming task is dependant upon an understanding of both models, since the developer must be aware of the nature of the problem and how to translate it into a meaningful solution in code. Additional complexities, such as those introduced by
interactions with other computer systems, hardware issues, as well as social factors, further complicate the situation. Pennington (1987) perhaps states the problem most clearly:

“A skilled computer programmer must understand the problem to be solved, design a solution, code the solution into a programming language, test the programs’ correctness, and be able to comprehend written programs.”

Pennington further states that knowledge of a wide range of subjects is required in programming, including:

- the real world problem domain;
- design strategies and useful design components;
- programming language syntax, text structure rules, and programming conventions;
- computer features that impact program implementation; and
- the user of the program.

The task of programming is therefore a particularly complex form of problem solving, since it entails understanding and coordinating aspects of many different fields of knowledge. Approaches to deal with this degree of complexity are examined in the next chapter.

2.6. Summary

Attempts have been made by psychologists and others to describe problem solving as proceeding in a well-defined fashion. Real world problems do not fit into this view – they are ‘wicked’. Formal design methods are available to designers in an effort to structure the solution of these problems, and to fulfil the broader organisational demands of the process. While these techniques are intended to structure the problem solving process, they do not reflect how it occurs in reality. This is less surprising when the full complexity of the programming task is understood. Expertise in general problem solving
has been studied, and appears to suggest that experts are influenced by knowledge both of the domain and of common solutions within the domain. Indeed, expertise is characterised by the development of such knowledge. The way in which the problem is represented has a major effect upon the solution strategy, and this will be addressed in more detail in later chapters.
"A complex system that works is invariably found to have evolved from a simple system that worked... A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system."

- Gall, J. (1978)

3. System development

3.1. Introduction
Programming is a complex area of human problem solving, particularly when (as in most commercial settings) groups of programmers must work together. Software development has become notorious for inaccurate schedule and cost estimates, and for producing software of inadequate quality (Pressman, 1997). To examine how and why problems occur in the development of software, this chapter will initially examine the programming task itself. Having identified the problems of a purely programming-oriented view of software development, the stages of software engineering (a more rigorous form of system development typically used in problems that are perceived as non-trivial, or where the activities of multiple people need to be coordinated) will be examined.

3.2. Programming
As well as its functional benefits, programming is an appealing creative activity. Brooks (1975) characterises programming as working like a poet, in a medium ‘...only slightly removed from pure thought-stuff’. More prosaically, (Blackwell, 1996) characterises it as planning the behaviour of a virtual machine, defined by the programming language. Programming involves both creative and logical aspects, and it is here that the true complexity of the task begins to become apparent. Bellamy (1994) notes that:
“Somehow program code has a finality about it, an illusion that it is close to a solution. When the code does not work, rather than throw it away and try to produce a better solution, programmers are reluctant to discard invested effort and tend to ‘hack’ at the existing code.”

It should be noted that the term ‘hacking’ is used in this thesis to describe an approach to programming that focuses on a single aspect of the problem to the exclusion of all others, trying different approaches without really knowing how that aspect of the problem relates to the whole system. It is however acknowledged that experienced developers also use the term almost as a badge of honour; such developers usually have sufficient experience to be aware of the larger problem context within which they are working, and may alternatively term this approach ‘exploratory programming’.

Whilst engaged in supervising laboratory sessions for both undergraduate and postgraduate students at Loughborough University, the author noted that the students who consistently performed better were those who did not begin coding immediately, but who explored the problem using (typically) pen and paper. It seems likely that mentally moving away from the implementation domain (the program editing tool or development environment) enables developers to focus on the problem that they are trying to solve rather than (as with the ‘hacking’ approach) the implementation detail. This is perhaps analogous to the ‘incubation’ period suggested by the Gestalt school of psychology outlined in the previous chapter, enabling developers to (consciously or otherwise) focus on what they are trying to do, rather than how they are trying to do it.

Much of the complexity in programming stems from the degree of interaction to be performed by the program. The program may interact with the user, the operating system, and other systems, and these interactions need to be coherent in both syntax and semantics. Particularly in an industrial setting, computational efficiency and reusability are likely to be important issues. To cope with this complexity, programmers shift between abstracting from the problem domain, and generating conceptual structures based on these abstractions to produce a solution (Blackwell, 1996). Solutions may then be evaluated
using mental simulation (Adelson and Soloway, 1985). As programmers gain expertise in a problem or technical domain, the need to perform such simulation may decrease, to be replaced by the use of plans, formed from experience, and enabling the developer to focus on novel or more challenging parts of the problem (ibid.). Since problem solving is largely opportunistic (Green, 1989; Guindon, 1990), it is reasonable to suggest that a developer will at any given time understand different parts of the problem to different levels of detail, as previously indicated by the lift simulation in Section 2.3.

Given the scale of the programming task, it is unsurprising that good developers make considerable use of external notations in the process of developing systems. A free-form medium such as paper does not restrict the level of abstraction which may be used; developers can represent the problem in any way they see fit, be it in text, diagrams, code, or any combination of these that is supported. Indeed, Petre and Winder (1988) showed that expert programmers use a pseudo-language made up of notations found to be convenient from known languages, as an intermediate step towards coding a solution. In addition, research into the mental imagery used by programmers in designing programs varies considerably (Petre and Blackwell, 1997); among other factors, developers have been tentatively identified as using imagery that varies in terms of:

- completeness;
- granularity of abstraction;
- number of dimensions (usually greater than four); and
- labels assigned to entities.

It is probable that these varying forms of imagery reflect the real-world problem domain and technical aspects of the problem, and the efforts of the developer to reconcile them in order to navigate the many levels of abstraction involved. The complexity of programming is reflected in the (often large) number of people in software development teams. This stems from the need to organise and coordinate information from the problem and solution domains, about the programming language, and about other contextual factors influencing the solution development. This is reflected in the observation of Greenbaum
and Kyng (1991) that the major activity in (team-based) software development is the teaching and learning that takes place between developers. The demands of this activity are further shown in "Brooks' law" (Brooks, 1975), which states that "Adding manpower to a late software project makes it later." A new developer arriving part way through a project must spend time effectively being taught about the system and its associated problem and solution domains, increasing the overall amount of work involved. In order to structure the problem solving process within software development, concepts have been borrowed from the engineering disciplines, and these have collectively become known as software engineering. This is discussed in the next section.

### 3.3. Software engineering

The problems of programming described above led to what became known as the 'software crisis', a term first coined at the NATO software engineering conference of 1968 (Naur and Randell, 1969). Pressman (1987) describes this crisis as a result of:

- inaccurate schedule and cost estimates;
- productivity of software developers not keeping pace with demands for their services; and
- software of inadequate quality.

Programming, particularly of large systems, is dependent upon many different factors. The developer must identify how to solve a real-world problem, and express that as a solution in a programming language. In addition to this, technical issues of maintainability, efficiency and reliability must also be addressed. Because of these inherent complexities, great emphasis has been placed on producing a design for an application, prior to implementing the system. That is not to say that producing small code fragments for the purpose of investigation is (or should be) completely eliminated. The act of programming can indicate potential sources of error with the approach being taken, which Green (1990) describes as 'fruitful mistakes'. But as Bellamy (1994) noted, a strong tendency among programmers is to create a solution which is then hacked to make
it work. This is not an effective approach to developing systems of any size, or where the work of many developers must be coordinated. Software engineering has drawn on concepts employed in conventional engineering projects, in order to formulate a sound basis for design. This is intended to enable the production of software that Sommerville (1996) suggested, should possess four attributes:

- **Maintainable**: it should be possible to make changes to the software without undue cost;
- **Reliable**: the software should be reliable at an appropriate level;
- **Efficient**: while a system should not pursue efficiency at the cost of maintainability, it should not make wasteful use of system resources; and
- **An appropriate user interface**: whatever other good qualities a system may possess, it may fail to be used to its full potential if the user interface is inappropriate.

Maintenance is important, because software typically exists within a dynamic human-computer environment. The tasks which a particular program is expected to fulfil may change once the system has been implemented, or user feedback may prompt a change to the interface of the system. Therefore, it is proposed that the system should be designed in such a way that areas of concern for maintenance are identified early, and accounted for in the design of the system. Reliability is clearly a desirable attribute of all software. However, the consequences of a word processor failing are perhaps less traumatic than if a control program for a nuclear power station fails. When a program fails, it should do so gracefully – without affecting other programs and ideally, with some possibility of saving the user’s work or shutting down critical systems. Whilst the system should not be profligate in its use of available resources, this should never be a higher priority than reliability or maintainability (Sommerville, 1996). Since the user interface is the primary means of access to the functionality offered by an application, its importance must be fully understood. It is believed that the impression formed by the user of an application is based primarily on the user interface, and for this reason, user interface design is a major field of study in its own right. Many areas must be addressed in interface design, and a number of useful reference works exist (e.g. Nielsen, 1993; Precece et al.,
While the importance of the interface is not disputed, the design of an interface is still made up of abstractions that are drawn together during the design process. These abstractions must present the user with a model of how the information required by the software application is transformed into the required output, and this model can be considered to a great extent to be independent of the abstractions used in the design of the software. As such, interface design can be considered to be a subset of the larger design task, since the abstractions used to deliver the functionality when writing the software are usually different abstractions to those perceived by the user. Therefore, interface design will not be addressed separately, but will be considered as part of the general system design task.

The extent to which engineering concepts apply to software development must be understood. Feynman (1996) argues that computing as a field of study is an engineering discipline, as it deals with artificial, rather than natural phenomena. However, in the development of software, engineering concepts must be applied with care. One of the main differences between traditional engineering and software engineering is the relative costs of the stages involved. A traditional engineering project using physical construction materials is typically very expensive to build. This places considerable importance on the need for a clear, well thought out design, since the final product must be well understood before the expensive building process takes place. Conversely, building software costs virtually nothing by comparison, and as a result, a much more exploratory style of development can be (and frequently is) adopted in preference to spending time in creating a better design. This can take the form of exploratory programming, where solutions to problems are sought within an understanding of the larger goals of the system. Without this understanding, where the program evolves without thought (or indeed any real degree of professionalism), the approach is considered to be hacking.

Engineering with physical materials engages the senses in many different ways. Each material used will have physical properties that are (usually) sufficiently well understood to enable it to be manipulated easily and for its behaviour to be relatively predictable. Conversely, in developing software a programmer will be defining all but the smallest
elements from which the program will be constructed, which immediately increases the scale of the task. This has led to calls for software engineering to utilise predefined ‘components’ in an effort to realise the same benefits that they have brought to conventional engineering. This is not as straightforward as it may initially appear, and this is explored further in Chapter 7.

To structure an examination of the issues associated with software engineering, the development process will be divided into three broad stages, after Pressman (1987). These are problem identification, solution generation, and maintenance. Many different software development methodologies have been proposed, including the Structured Systems Analysis and Design Methodology (SSADM) (Downs et al., 1988), the Dynamic Systems Development Method (DSDM) (DSDM Consortium, 1995), and Jackson Structured Programming (JSP) (Storer, 1987). However, each of these approaches addresses the same fundamental stages. Rather than go into the details of each therefore, these stages will be discussed. Issues associated with the management of software development projects per se will not be considered; there are many excellent reference works on this subject (see for example Hughes and Cotterell, 1999); the management of the development process is considered to be peripheral to the main areas undertaken in this research.

3.3.1. Problem identification

In 1987, Pressman wrote:

"Poor initial definition is the major cause of failed software efforts. A formal and detailed description of information domain, function, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can only be developed after thorough communication between customer and developer."

Thirteen years later, Reifer (2000) noted that:
"We have been taught to spell out the requirements at the beginning of a project and not to change them. Experience has shown that these lessons are impractical and impossible to achieve...We have discovered that requirements development is a learning, rather than a gathering, process. [Original italics]"

These two statements have been written from different perspectives on the software development process: the former from the developers perspective, the latter from the project management perspective. However, the final sentence in both indicates that despite the gap of thirteen years, and the difference in perspectives, identifying requirements is an evolutionary process, rather than a one-time-only process. This depends upon good communication between the developer and the customer, something that develops over a period of time.

That little has changed in the essence of these two statements indicates that adequately describing the problem to be solved continues to be difficult, and that no effective methodology has been developed for problem identification at the time of writing. Correctly identifying a problem is a major step towards being able to focus resources towards its resolution. Failure to perform this process effectively can be costly, both in terms of delivery date and financial expenditure for a project. To illustrate this, Boehm (1974) reports that typically 12% of the errors identified in a large software system were due to errors in the original system requirements. Much of the work aimed at improving the effectiveness of the requirements gathering process has emphasised the involvement of the (eventual) user of the system (e.g. Constantine and Lockwood, 1999). To stimulate feedback from users about the developing system, rapid prototyping has been adopted in many organisations. This is a useful technique that produces a prototype of the system, usually with some limited functionality, enabling the customer to experience how a system will appear, and providing a basis for discussion and feedback. This can ensure that the developer has correctly understood what the customer wants. Brooks (1975) observed the trend towards increased effort being devoted to effective problem identification in projects of all sizes, and created a ‘rule of thumb’ for scheduling future software development:
• 1/3 planning;
• 1/6 coding;
• ¼ component test / early systems test; and
• ¼ system test, all components in hand.

Brooks suggests that even with 1/3 of the available time dedicated to planning, this is barely enough to produce a detailed, dependable specification - and is certainly insufficient to research new techniques.

One of the most recent approaches to identifying requirements is use case modelling (Jacobson et al., 1992). This is a particularly effective method of requirements capture since it describes how the final product will behave, and it is claimed that it can provide an effective bridge between the software engineer and the customer in discussions (Constantine and Lockwood, 1999). Use case modelling is both important in its own right, and also as a significant contribution to the Unified Modelling Language (Booch et al., 1999), which will be addressed in Section 4.5.

A variety of information types must be obtained during requirements capture. Stokes (1991) notes that six types of information are present in the majority of requirement specifications. These are:

• functionality;
• functional constraints;
• design constraints;
• data and communication protocols;
• project management; and
• environmental description / system objectives.

Structuring requirements in this way essentially captures both what the system is to do, and the environment within which it will operate. Requirements should also be both
complete (all required functionality should be noted) and consistent (requirements should not be contradictory), as Sommerville (1989) notes. As several authors have indicated (e.g. Sommerville, ibid. and Pressman, 1987), it is important for requirements to be stated in a way which enables the developer to establish when they have been met. For example, to state that a particular interface should enable telephone operators to locate customer orders quickly is insufficient. ‘Quickly’ should be defined as (for instance) locating orders within a particular time period based on specific information. This approach enables success or failure to be quantified and weak areas to be addressed.

Requirements are usually described using natural language. Whilst this is beneficial in that the requirements can then be easily discussed with clients, natural language does have inherent problems of interpretation, in that two people may disagree on what a particular requirements statement means, or on how conflicting requirements should best be implemented. While this may be partially addressed through the use of structured English, this can give the impression of coherence where none exists.

3.3.2. Solution generation

Good design is considered to be vital in producing systems of any meaningful scale (e.g. Pooley and Stevens, 1999). A design is effectively a plan for converting a description of a problem into a solution (Kitchenham and Carn, 1990). The act of design can help the developer to identify further requirements, potential implementation difficulties, and common functionality. This can lead to considerable time savings during development and ultimately, to increased acceptability of the software. Pressman (1987) effectively illustrated the need for design:
Figure 3.1 illustrates that a good design provides a solid foundation for system development. This contrasts with a hacked approach, where an initially small piece of code grows into a large, unstable lump. The work of Schon (Schon, 1983; Winograd et al., 1996) is notable in any study of design. Schon acknowledges that the design process is inherently complex, due to the range of factors that must be reconciled. In an interview with Winograd et al. (ibid.), Schon notes that there are many factors that influence a design which cannot easily be represented in a development model, and that this gives the design process an inherent complexity. Schon suggests that a designer will undertake a process of 'reflection in action', in order to try to manage this complexity. This refers to the way that designers (in all disciplines) develop an increased understanding of the nature of problems in the process of solving them. This increased understanding can be reflected in the identification of further problems or solutions, or simply increased knowledge. Particularly in software engineering, this can result in the development of requirements as the project progresses.

To illustrate this complexity of design, consider an invoicing system for a telephone sales company. Within the problem domain, the developer must identify what is needed. While management will be employing the developer, existing telephone operators must be
interviewed to identify what is needed, but management will almost certainly want to influence the system as well. Conflicts in what is wanted will almost certainly arise. The designers may have experience in performing procedural design, but an object-oriented approach will almost certainly be more effective for an event-driven system. The host organisation may have a legacy database and be in the process of upgrading it. If the invoicing system must interact with both, this can add additional complexity. For these reasons, software design problems are typically wicked problems (see Section 2.3), and as such, are subject to the same constraints and approaches outlined in the previous chapter.

To cope with this complexity, the development activity is usually divided up into a number of abstract stages. These stages form the basis for the large number of design methodologies that are available. However, it would be foolish to suggest that a significant proportion of developers follow a well-defined design process in practice. Although Chapter 2 demonstrated that formal models of software development do not truly reflect the process as experienced by software developers, a study by Basili and Reiter (1981) concluded that the use of a disciplined methodology enabled a group of developers to coordinate their activities effectively. Thus, although software development methodologies may not reflect the realities of development, the benefits derived indirectly, particularly through improved group working, appear to compensate.

3.3.3. Maintenance

Maintenance has been defined as:

"...the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment." (ANSI/IEEE, 1983)

Maintenance is typically the most costly stage of development, largely due to the need for the software to evolve to a changing environment (Bennett, Cornelius, Munro and Robson, 1991). Pennington (1987) states that over 50% of all professional programmer time is
spent on maintenance tasks involving modifications and updates of previously written programs. Thus, the importance of this stage must not be underestimated.

Brooks (1983) has proposed a theory of program comprehension based on a mapping between the problem domain and the programming domain. He argues that the developer produces these mappings, and it is the role of the maintainer to reconstruct them, a process that takes place in a bottom-up fashion. As a result, it can be useful to understand both the decisions made in a particular situation as well as the reasons for those decisions. This aspect of development is rarely preserved for inspection by maintenance engineers, and is referred to as a problem of invisibility (Devanbu et al., 1991). This can be alleviated by good documentation, which is discussed in the following section.

3.4. The role of documentation

Documentation plays a vital role in understanding both what a particular system does and how it does it. Documentation in this context refers to that which is produced and used by the developer(s) of the system for the purposes of understanding what has been done, particularly during the maintenance stage of the system life cycle.

Reeves (1992) has argued that programming is itself a design activity, and that the code listing of a system is the most accurate design document available. Whilst this is technically correct, large projects comprise an extremely large number of coding steps. The author of this thesis believes that this approach would be highly unsuitable in such (indeed, most) cases, due to the limited cognitive and memory capabilities of humans. Furthermore, code is unsuitable for easily enabling the higher-level functionality of a system to be determined. As mentioned earlier, software involves both an understanding of how to solve a real-world problem and an understanding of how to implement that solution in code. In a system of any real size, there is a vital need for developers to be able to join the project part way through and understand both what the system is trying to achieve and how it is doing it.
One approach which has been used to try to achieve this is literate programming (Knuth, 1992). Here, the code and its documentation are maintained in a single file, enabling documentation to be automatically generated, as well as maintaining an explanation of what the code does in immediate physical proximity to the code. This approach is increasingly being adopted in programming environments. For instance, the Java programming language utilises this approach in the JavaDoc tool, which is discussed in more detail in Section 5.1.15.

The nature of software makes it imperative that it can be understood from multiple perspectives (Brooks, 1983; Houde and Sellman, 1994). Devanbu et al. (1991) identify four of these:

- **A domain model view**: what is the code doing relative to the conceptual objects and actions in the problem domain?
- **An architectural view**: how does the software relate to the physical system architecture?
- **A feature view**: how do system functions relate to the features of the software?
- **A code view**: how do the code-level components relate to each other?

To enable developers to access these views, Devanbu et al. (ibid.) created the Large Software System Information Environment (LaSSIE). This was an attempt to create an integrated architectural, conceptual and code view of a large software system within a knowledge base to be used by developers. The knowledge base could be interrogated via a graphical browser and a natural language query processing system. The developers of LaSSIE used domain analysis to create the knowledge base, stating that:

"...the domain analyst looks at a variety of systems that service the same application domain and produces an external description of a set of reusable components, which can be used to build applications for that domain."
Whilst LaSSIE provides a fairly powerful knowledge base for interrogation, it does not provide a great deal of contextual knowledge. It could be argued that knowledge about the problem domain and wider organisational issues are vital in understanding how and why a system was produced, and thus to comprehending the rationale behind certain design decisions. An alternative approach to capturing this sort of contextual information was proposed by Fischer et al. (1994). Domain-Oriented Design Environments (DODE’s) are designed to capture design information throughout the lifetime of a system, making this information accessible to developers at the appropriate time. A simple DODE, produced to assist with network design, is shown in Figure 3.2:

The DODE architecture consists of five elements:

Figure 3.2: Sample DODE interface (from Fischer et al., 1994)
• a construction component;
• an argumentation component;
• a catalogue of design examples;
• a specification component; and
• a simulation component.

These are linked by three knowledge-based tools: a critiquing system, an argumentation illustrator and a catalogue explorer. By linking informal textual and graphical records by formal associations, the designer can provide design argumentation. Three main activities are involved in the use of a DODE:

• **Seeding**: the DODE is seeded with domain knowledge;
• **Evolutionary growth**: domain designers (the users of the DODE) add information to the seed as they use it to create design artefacts; and
• **Reseeding**: environment developers help domain designers to reorganise and reformulate information so it can be reused to support future design tasks.

Unfortunately, the DODE approach did not meet with a great deal of success. An evaluation of a DODE by Sumner et al. (1997) concluded that designers learned the conditions under which the critiquing system would intervene. Although less experienced designers made a relatively high number of changes in response to the critic, more experienced designers actively took steps to anticipate and avoid critic intervention. Another factor that might lead to limited acceptance was the additional workload imposed by structuring design information. Of particular importance is that, as has been demonstrated, a designer shifts between levels of abstraction in solving a problem, addressing different parts of the overall problem at different times. This leads to the question of where in the design process a developer would be expected to generate this support material. Experience with software design indicates that it is difficult enough to get people to use conventional design tools, without adding additional work.
Another system that aimed to create a source of information about the history of a design was the 'Raison d’Etre' system of Carroll et al. (1994). This was much more general in scope and used questionnaires and video recordings of interviews with developers, to create a database of information about the history of a design, identifying why particular decisions were made. This approach provided a much more personal view of the design and how it developed over time than the DODE approach, but did not relate the gathered information directly to the design and implementation. While an interesting tool in its own right, its value in comprehending the design in retrospect must be questioned, due to the volume of material generated, the lack of direct relationships with the design and implementation, and the costs associated with the approach.

While systems such as these may be valuable in understanding both what a system does and how it does it, no single system captures and integrates information from both the problem domain and the solution domain. The most effective of the systems described appears to have been the DODE approach, but in practice the value of this is primarily as an assistant to a designer, and as the empirical study of Sumner et al. (1997) showed, was useful mainly for novice designers. The workload involved in initialising systems such as these should also be considered. The creation of the knowledge base for the LaSSIE system took 0.5 person years (Devanbu, 2000, personal communication) over and above the normal development effort. This is a considerable workload to add to any project, particularly given the existing complexity of the task.

3.5. Conclusions

The development of software is a complex task, particularly when wicked problems are addressed by groups of developers. In order to structure an approach to creating software systems, software developers have adopted certain aspects of engineering. The development process entails three stages: problem identification, solution generation and system maintenance. These stages are closely related, and this is illustrated by the potential for omissions in the requirements process (itself a complex task) to lead to additional work in system maintenance. Whilst the previous chapter has demonstrated that
developers do not follow design methodologies especially closely, this chapter has demonstrated that design methodologies nevertheless play a significant role in effective problem solving, particularly for large systems, or where many developers are involved. A reasonable middle ground is to acknowledge that while there is a role to be played by exploratory approaches, particularly where new technologies are being employed or a new problem domain is involved, this should not be the primary development strategy. Rather, it should be an adjunct to a development process that generates a well-defined, maintainable, reliable design, while assisting in communication and coordination between developers.
"...and what is worse than all, a profusion of notations (when we regard the whole science) which threaten, if not duly corrected, to multiply our differences instead of promoting our progress."

- Charles Babbage

4. System implementation

4.1. Introduction

The purpose of the analysis reported in this chapter is to identify techniques that may be used to make the programming task more accessible. To achieve this goal, it is important to study programming at the language level, and in terms of the notations that support the broader development task. The creation of computer systems is challenging since it entails addressing problems in two domains. On the one hand, there is the real-world problem domain and the constraints it imposes upon the solution. On the other hand are constraints imposed by the system (or systems) upon which the solution is to be implemented or with which the solution must interact (including human systems). This introduces its own problems that can be at least partly addressed through the design of the programming language selected to implement the solution. The selection of a programming language can be a daunting task, not least because of the vast number of languages available. Programming languages have changed significantly over the last fifty years, but since all programming languages are essentially interfaces to the underlying machine operations, Ben-Ari (1996) suggests that they are all capable of performing the same tasks. While this may be technically true for most general-purpose programming languages, in practise programming languages differ on a wide range of factors that have a significant effect on their usage. These include:

- the (potential) level of abstraction from the machine;
- the (potential) level of abstraction from the problem domain;
- the extent to which the design of the language can promote reliable, maintainable code;
• the paradigm underlying the language.

Both theoretical and pragmatic concerns must be addressed in deciding upon a programming language to use and to study, and this is evident when the evolution of programming languages is examined. The rapid development in the power of computing hardware has led to changes in the emphasis that must be placed on different aspects of the programming task. A recent example of this is the 'Millennium Bug' problem, produced by the desire to conserve computing resources for date storage, by reducing the representation of years from four to two digits on both the disk storage and the CMOS (Complementary Metal Oxide Semiconductor) RAM (Random Access Memory). It is unlikely that a modern developer would generate code that reproduced this error, simply because it was made at a time when hardware resources were scarce and expensive – modern hardware is comparatively cheaper, with higher performance. Changing hardware and the Lamarckian evolution (Kelly, 1995) imposed by the success or failure of programming languages have both been significant pragmatic factors influencing their development. In addition to this are the theoretical concerns that have been both led by, and which have led, this development. It was not realistic to discuss all the programming languages available to developers, nor was it considered to be particularly useful in the context of this project. To illustrate how these changing external forces have affected programming languages, this chapter will examine a representative sample of languages and identify significant trends in development.

Conservative estimates (e.g. Ben-Ari, 1996) suggest that there are over four thousand programming languages currently in existence. However, the number of programming languages in widespread use is far smaller than this. This chapter will identify the trends in programming language development that have influenced languages still in widespread current use. In addition to identifying languages that may be useful for the development of prototype systems within this study, this will also identify languages for studying existing programming issues. In support of this, design notations will be examined in terms of the support they offer to the development process. Significant concerns in programming can then be understood in greater detail.
4.2. Programming languages

Ben-Ari (ibid.) defines a programming language in the following way:

"A programming language is an abstraction mechanism. It enables a programmer to specify a computation abstractly, and to let a program (usually called an assembler, compiler or interpreter) implement the specification in the detailed form needed for execution on a computer."

The development of programming languages is closely linked to the rapid growth in the performance of computer hardware. The processing and storage capabilities of computers in the 1950's and 1960's were extremely limited relative to those of modern computers. This placed great emphasis upon the need to create programs that made the most out of these limited capabilities, by manipulating resources at a low level. Whilst this permitted a high degree of control over the available resources, cognitively relating the program code to the function it was designed to perform became increasingly difficult as programs grew in size. This affected both the speed of development and the reliability of these systems, which in turn had a significant effect on their cost. Two of the early programming languages enabled the developer to abstract away from the machine level and express concepts closer to the problem domain. Both FORTRAN (FORmula TRANslator) and COBOL (COmmon Business Oriented Language) were designed to support particular problem domains, FORTRAN for scientific and engineering problems and COBOL for business data processing.

FORTRAN is an example of an imperative language, that is, a language in which the programmer explicitly defines the operations that are to be performed. FORTRAN was initially developed in 1957, and was arguably the first widely used programming language, but its design reflects the issues of the time, in that it has very weak checking for syntactic errors, unlike later languages. As Backus (1981) comments:
“As far as we were aware, we simply made up the language as we went along. We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler which could produce efficient programs.”

As Backus (ibid.) notes, prior to FORTRAN usage, ¼ to ½ of the computer time was spent debugging programs. As a result, it was desirable to improve the performance of the programmers as well as that of the machines. FORTRAN enabled this to be done by replacing many machine-level operations with a few higher-level commands. FORTRAN was also significant because it introduced the concept of functions and the resulting ability to structure programs more effectively than was previously possible. While functions can take many forms, the basic concept remains the same: a discrete unit of code that can be referenced from a program in order to perform a well-defined task. This enables programs to be more readable as a result of reduced code size. FORTRAN was also significant because it enabled programmers to assign variable names (initially of up to six characters). Both of these factors were important in the production of generalised function libraries, the availability of which further aided the rapid development of reliable code.

The other language that provided a major spur to the development of programming was COBOL, developed in 1960 and first standardised in 1968, as a language oriented to the needs of business data processing. An interesting issue in its development was pressure from the business data processing community to enable the use of words such as ADD, SUBTRACT and MULTIPLY rather than their equivalent symbols. This was on the grounds that COBOL was to be read (if not coded) by management and other non-specialists, and these groups could not use, or could not comprehend, the equivalent mathematical symbols (Sammet, 1981). A major factor in the prominence of COBOL was the backing it received from the US Department of Defence.

COBOL has syntax and reserved words based on English. Whilst this can be advantageous in that they appeal to many ‘non-technical’ people who view them as less intimidating than other languages, by the same token they can give a false sense of
security, as an accurate specification is still needed (Norman, 1991). COBOL is machine independent, which promotes the portability of software written in the language—something that is important from both a technical and a commercial perspective.

Both FORTRAN and COBOL were led by the demands of their respective problem domains. LISP (LISt Processing) emerged in 1962 from research into computing theory, and was subsequently of great importance in Artificial Intelligence (AI) research. LISP was the first language to be oriented around a particular data structure, the linked list. It also marked a change from programs written based around a flow of control, to programs based on a flow of data. The von Neumann architecture for computing hardware had previously been the dominant driving force for language development. This architecture developed from the relative costs of the different physical parts of the computer. Cheap and slow, long-term storage is connected to more expensive, fast, short-term storage (via the system buses). This in turn supplies a (still more expensive) Central Processing Unit (CPU) which performs the computation (Goldstein, 1972). This model imposes a view of programming where a series of instructions is performed upon data, and the program is the central concern, moving from one instruction to the next. The data oriented approach of LISP presented a major shift to this paradigm. In addition to its theoretical value as an alternative approach to the von Neumann architecture, LISP demonstrated the value of defined data types.

The importance of the Pascal language (Wirth, 1971) cannot be underestimated. Pascal was originally designed to demonstrate ideas about type declarations and type checking (Ben-Ari, 1996). These are important in that they enable the programmer to identify certain types of error at compile-time. Pascal also offers an interesting example of how languages evolve. It led to the development of Modula, a language based around the code module: a unit of code containing both data and functionality. The encapsulation of these can be a major factor in enhancing the comprehensibility of code. Pascal initially received widespread use as a language for teaching, and is still popular in various forms, including as an object-oriented language and environment called Delphi (Konopka, 1996). That the language exists in these different forms is an indication of how both technical and
commercial factors, together with individual programmer preference, have affected the adoption of different programming languages and tools. Furthermore, the strong type checking that exists in all forms of the language is a testament to the value of this idea.

C is arguably the most significant language in the history of computing. It has been described as a high-level assembly language, since it provides aspects of both high and low-level languages (Kernighan and Ritchie, 1988). Whilst it has structured control statements and data structures, it also offers low-level features such as pointers and bit-level operations (Ben-Ari, 1996). In this way, it enables programmers to address low-level performance concerns as well as providing the facilities for higher level structuring. However, while the combination of these features can be extremely powerful, it also offers significant potential for error.

SIMULA (Dahl, Nygaard and Myhrhaug, 1969), designed for discrete event simulation, was probably the first language where the programmer explicitly created a model of the real-world system of interest. This reflected the increasing abstraction away from the machine level towards representing elements of interest in the problem domain. This approach had a major influence on the Object-Oriented (OO) development style. Possibly the most important language in the development of OO was Smalltalk (Goldberg and Robson, 1983), although strictly speaking this was a language and a programming environment. Smalltalk is important because it is a ‘pure’ OO language, as everything is viewed as an object, and because its concepts influenced the design of subsequent OO programming languages. However, OO gained widespread use with the introduction of C++ (Stroustrup, 1986). This extended the C language to provide support for OO, while addressing many of the weaknesses inherent to C. For this reason it is recommended that programmers use C++ even if they are not using its OO features (Ben-Ari, 1996). C++ is not considered to be a pure OO language in the same way as the Smalltalk system, as it uses primitive data types.

Early versions of Smalltalk, like other early OO programming environments, gained a reputation for being large and slow. For instance, a version of C++ within the GPT (GEC
Plessey Telecommunications) company generated a graphical display of the “Hello World” program using proprietary X Windows libraries which occupied 27MB; this compared to a Borland C++ equivalent which occupied 9k. As the complexity of an approach increases, so does the potential for poor implementations – in this case, implementations that include non-essential class information. But despite such disparities, C++ offers a workable compromise between the advantages of OO in terms of comprehensibility and reusability, and the speed of a more traditional language. The syntax of C++ has been largely copied by Java (Gosling and McGilton, 1996), a language which addresses some of the identified weaknesses of C++, principally the problem of ‘memory leakage’ (also known as ‘heap creep’) caused by the problems inherent in the dynamic creation and destruction of variables. Java is discussed in more detail in Chapter 5.

4.2.1. Summary

The trend in programming languages has been for the programmer to be able to move away from concerns about the machine-dependant implementation details of the solution, and more towards addressing the primary issue of the real-world problem to be solved. Programming languages differ in their overall approach to how they enable a solution to be implemented, and this defines the paradigm to which they belong.

Four ‘types’ of programming language can potentially be distinguished: imperative, applicative, OO and declarative. However, it can be argued that OO subsumes imperative, and that declarative is a subset of applicative. Also, in practice, hybrids of two or more of these exist, such as SQL – an imperative language working on a declarative data set. Of these, OO is the most suitable language type to use in implementing reuse, as reuse is a fundamental part of such languages. Whilst precise figures are difficult to obtain, there is strong anecdotal evidence that programming languages belonging to the OO paradigm are widely used for the development of new systems. For example, Rational, a supplier of OO development tools has supplied these to over 26,000 companies (http://www.rational.com). OO development is used in many international organisations, including British
Telecommunications plc., IBM, the UK Defence Evaluation and Research Agency (DERA), and Ericsson. It seems reasonable to conclude therefore that OO is an important approach to modern software development. That the use of OO has continued even after the initial ‘fashionable’ phase is significant, and Graham (1991) has argued that a major reason for this is their suitability for the development of event-driven Graphical User Interfaces (GUI’s), particularly with the supporting toolkits that are available (such as Swing under Java). The remainder of this chapter will examine the essential features of the OO approach. Notations used in design will also be addressed, principally the Unified Modelling Language (UML). This is a comparatively recent development, which is claimed to be independent of a design process (Fowler and Scott, 2000). This claim is contentious, particularly since Green (1990) demonstrated that the notation selected influences the problem solving approach taken. It seems to be self-evident that the features of any notation will constrain what can be expressed, particularly if it has a well-defined syntax and semantics (as is the case with UML). Regardless of this, UML is enjoying considerable usage at present, and thus requires investigation.

4.3. Abstraction in programming languages

As has been noted, it is desirable to limit the amount of information that must be dealt with by the human at any given time. This is highlighted by Dijkstra (1972):

"...as a slow-witted human being I have a very small head and I had better learn to live with it and to respect my limitations and give them full credit, rather than try to ignore them, for the latter vain effort will be punished by failure."

Dijkstra, an early advocate of structured programming, was acknowledging that the ability to create small programs does not scale to creating large programs. Structured programming is a group of techniques employed to improve the quality of the solution, by addressing problems of abstraction and structure. A useful mechanism in promoting increased understanding of a program is the module. The concept of the module is that
code can be encapsulated into a unit that may be called upon by another program. This general principle has been implemented as (for instance) modules in Modula, and classes in various OO languages (e.g. C++, Smalltalk and Java). The use of modules enables programs to call upon functionality that is implemented separately. The programmer then does not need to understand how exactly the functionality is implemented, only to understand what the modules 'black box' behaviour is, and how to call it from the main program. This can be a useful tool in making a program more readable since the programmer does not have to include all the low-level code in the program file. This can in turn have benefits for understanding and modifying the code, particularly for maintenance.

4.4. Problems in programming

While there are known solutions for many of the major problems in programming, one of the most challenging (and most promising) areas is that of concurrent programming. The popularity of computing has led to a constant drive for greater performance. However, Hillis (1985) notes that the von Neumann architecture is extremely limited by virtue of bottlenecks in the flow of information that result from the clear division between the memory and the processor. As these operate at different speeds, bottlenecks are inevitable. Hillis’ Connection Machine typifies the ideas of parallel processing. This employs a more homogenous architecture, which combines the memory and processing in “...hundreds of thousands or even millions of tiny processing cells” (Hillis, ibid.). While this is an extreme example of concurrency, machines using multiple processors are becoming increasingly common. The goal of concurrency is to distribute the workload of processing among many processors, overcoming the ‘von Neumann bottleneck’.

Unfortunately, programming for a parallel architecture (concurrent programming) is a complex task in terms of the synchronisation that must take place between the many processing elements involved. Only a few languages are available which offer support for concurrency, including Ada, CmLisp (Connection machine LISP) and Java. Like COBOL, Ada was strongly influenced by the US Department of Defense. It was designed to support the writing of portable programs, unlike languages such as C which have
machine-dependent quirks. Ada also implements error handling; a feature often left to the underlying operating system. In addition, Ada was subjected to considerable review before standardisation. Despite these benefits however, Ada received little support outside of the defence community.

Concurrency is a significant issue in programming, but is not the only area of difficulty. As a means of enabling the programmer to manage the complexity of the task, many notations are available. These are explored in the next section.

4.5. Notations to support programming

Design methodologies have typically been developed after programming languages, and although this is a topic worthy of study in itself, there is not the space to do it justice here. While methodologies typically use (semi-) formal notations to express pertinent concepts, notations are also used independently of methodologies (and historically, were used before methodologies). One of the earliest (widely used) form of diagram in software development was the flowchart. This can be used to represent common programming constructs, such as conditionals and loops (Dijkstra, 1972), and in doing so is similar to pseudocode in that it provides the developer with the ability to outline possible methods for solving a problem. Another approach is to describe how data flows from one logical processing unit to another (Sommerville, 1996), and this can be done using Data Flow Diagrams (DFD's), such as that shown in Figure 4.1. These represent what must be done (in a procedural sense) without describing how it is to be done.
Budgen (1994) notes that of the successful diagram types, most use only around four or five types of symbol. This may reflect the limitations on human memory noted by Miller (1956). These diagrams are not associated with a particular methodology, and have been used for many years. They are based around the control flow and data flow views of software development. However, the majority of modern software development is now based around the event-driven paradigm, which is usually driven by Object Oriented (OO) techniques. These are described in more detail in the next section.

### 4.6. Principles of Object Orientation

OO design differs from procedural design in a number of significant ways. Procedural design uses the data structures of the problem domain to form a program structure. The executable operations needed to perform the task are identified, and each operation is
allocated to a component of the program structure (Jackson, 1975). In this way, a
distinction between data and functionality is emphasised. Conversely, OO programming
languages stress the need to encapsulate both data and functionality in discrete 'objects'.
This results from the historical roots of OO in simulation languages, notably Simula.
Jacobson, Christerson, Jonsson and Overgaard (1992) describe objects as being:

“...characterized by a number of operations and a state which remembers
the effect of these operations.”

The objects that make up an OO system are software abstractions from (usually)
real-world entities and features of those entities (Shlaer and Mellor, 1988, 1992). OO
programming languages are particularly useful in implementing event-driven systems,
such as GUI's.

Objects are created from classes, which can be thought of as templates for objects. While
a class will define the basic attributes and operations of an object, many objects with
different states can be created from a single class. Although there is some disagreement
over the defining characteristics of OO, it is generally understood that an OO
programming language will implement mechanisms for encapsulation, inheritance, and
abstract data types (Henderson-Sellers, 1992). These concepts will now be described in
more detail.

4.6.1. Encapsulation

Encapsulation (also known as abstraction or information hiding) is concerned with
representing the essential features of an entity, without including unnecessary detail
(Budgen, 1994; Graham, 1991). While encapsulation is not restricted to OO, it is an
important part of the OO approach to software development. In OO terms, encapsulation
enables the functionality possessed by an object to be utilised without needing to know
how that functionality is implemented, or as Sommerville (1996) puts it: “...a design
strategy in which as much information as possible is hidden within design components”.

54
Since each object communicates by way of public interfaces, developers can modify the internal implementation of a module that performs a given function without disturbing the overall structure of the system. This can enable maintenance to be performed more easily, but places greater emphasis on the need to have a good design in the first instance. Corritore and Wiedenbeck (1999) demonstrated that developers make use of the encapsulation features of OO languages to avoid needing to understand how functionality is implemented. This is significant because it enables the developer to focus their attention less on implementation details, and more on the task being addressed.

4.6.2. Inheritance

Another major feature of OO systems is the use of inheritance to structure the relationships between classes. Inheritance enables new classes to be created based on other classes. The developer can select one or more existing classes and add to, or (in some cases) substitute, attributes and methods. While it can be a powerful tool, inheritance hierarchies produced for the same problem by different developers can vary significantly. Dvorak and Moher (1991) demonstrated that differences in domain experience led to qualitative differences in the extent to which programmers agreed on the structure of class hierarchy decomposition. While Davies, Gilmore and Green (1995) argue that communication between and within teams of designers will be facilitated if a shared understanding of a domain can be achieved, in practice this takes time to occur, and the depth of understanding will vary considerably within the group. While it seems reasonable to suggest that this will occur in any domain of work, in programming different levels of abstraction are addressed that can influence the nature of the expertise that can be applied. For example, a developer can have knowledge in identifying an algorithm to address a problem, implementing a solution according to machine constraints, identifying user requirements, and so on. Communication is thus of the utmost importance since it enables developers to establish a clear view of what is needed to implement a solution that is acceptable to the client.
4.6.3. Abstract Data Types

Most programming languages have a number of types already defined for the programmer, such as the character and integer. OO enables programmers to define their own Abstract Data Types (ADT’s) and use them as they would any other basic language type (Henderson-Sellers, 1992). This feature is critical in enabling an OO program to reflect the structure of the real-world problem that it is intended to address. Ben-Ari (1996) notes that the advent of OO languages has had a negative impact on the use of languages such as LISP, which are oriented around a particular data type. Since OO languages can create new data types, it is possible for the programmer to embed data-oriented operations within the existing language framework.

4.7. The Unified Modelling Language

In recent years, many OO development methodologies have been available, such as Shlaer-Mellor (Shlaer and Mellor, 1988, 1992); Booch (Booch, 1994) and Object-Oriented Software Engineering (Jacobson et al., 1992). Booch et al. (1999) note that between 1989 and 1994, the number of OO development methodologies increased from less than ten to over fifty. Not only does such a profusion present a problem from the point of view of using OO (since an organisation must evaluate methodologies before adopting one), but also for enabling developers to communicate within and between groups. The methods of Booch (1994), Rumbaugh et al. (1991) and Jacobson et al. (1992) are very similar, but differ in some of the minor detail (Fowler and Scott, 2000). That the notations for these languages were different caused confusion in the software engineering community, in what became known as the ‘methods war’ (Fowler and Scott, ibid.). To overcome this problem, Booch, Jacobson and Rumbaugh developed the Unified Modelling Language (UML). This is claimed to be a methodology-independent notation for design (Fowler and Scott, ibid.). As such, it enables systems to be developed using a single notation but within the framework of different (OO) methodologies, facilitating communication between developers. At the time of writing, UML is in version 1.3, a standard which has been approved by the Object Management Group (OMG), the body responsible for producing and maintaining computer industry specifications for interoperable enterprise
Modelling is important in design because it enables the designer to better understand the system under development (Booch et al., 1999). UML enables developers to model the system of interest at both a real-world level and an implementation level, in both static and dynamic terms. In both the real world and the implementation, both static and dynamic aspects of systems must be addressed. While it is not possible to do justice to UML here, some features of interest will be briefly described. A number of texts (e.g. Pooley and Stevens, 1999; Fowler and Scott, 2000; Rumbaugh et al., 1999) go into considerably more detail.

4.7.1. Use case modelling

Use case modelling is a way of describing all the functionality that the system must provide. Two entities are important in use case modelling: the actor and the use case. An actor is any entity which is external to the system, but which interacts with it in some way in order to receive some measurable benefit from the interaction. Actors initiate use cases, which Jacobson et al. (1992), describe as "...a sequence of transactions in a system whose task is to yield a measurable value to an individual actor of the system". Use cases have four roles:

- to identify the system boundary;
- to assist the developer in gaining an understanding of the problem;
- to identify possible objects in the system; and
- to provide a structure for testing.

Use cases are usually supported by documentation describing the 'Flow of events' which makes up the use case. This describes how the use case starts and ends, how it interacts with actors, and alternative sequences of actions that may occur. This is usually done as a
textual document, while the relationships between use cases and actors are made explicit using a use case diagram (shown in Figure 4.2):

![Use case diagram for a course administration system](image)

Figure 4.2: Use case diagram for a course administration system

Figure 4.2 also illustrates how use cases can be used to identify common functionality at an early stage in the project. When several use cases share behaviour, this can be abstracted out into a separate use case, which is then called by other use cases via the 'uses' relationship. In another situation, use case $x$ may 'extend' use case $y$, if $x$ is a novel occurrence of $y$ that (typically) occurs infrequently. Both 'uses' and 'extends' can be powerful tools in identifying common behaviour at an early stage.
While use case modelling is widely used, it has been argued (Chandrasekaran, 1997) that it can be weak when used in the analysis stage of system development. It is claimed that many of the examples of its use (e.g. Quatrani, 1998) focus on the solution rather than describing the problem. This emphasises the need to use UML within a well-defined development process in order to avoid such problems.

4.7.2. Classes and Class diagrams
As the central feature of OO, the identification of classes is a major part of any development. Some researchers recommend using the nouns in requirements documents to identify classes, and using verbs to identify the operations that must be performed by those classes (e.g. Stevens with Pooley, 2000). Another approach is to use CRC (Class, Responsibilities, Collaborations) cards (Beck and Cunningham, 1989). Regardless of the method used, the goal of class identification is to be able to create a class model which satisfies current requirements, and is both easy to maintain and adapt to future requirements (Stevens with Pooley, 2000). Classes are represented in UML in terms of their attributes and the operations they can perform. Each class, attribute and operation is associated with a textual description. As well as providing the functionality within the system, classes exist in relationships with other classes. Class diagrams can be used to represent inheritance, aggregation/composition, and other pertinent features of the classes in a system.

4.7.3. Sequence diagrams
The role of sequence diagrams is to model the dynamic aspects of a system. Sequence diagrams can be applied to both use cases (to graphically illustrate the flow of events) or design (e.g. to illustrate how the system implements functionality as message passing between objects). Sequence diagrams illustrate the passing of time and the interactions between parts of the system, as shown in Figure 4.3:
Some authors (e.g. Rosenberg and Scott, 1999) suggest that it can be useful for a developer to have a sequence diagram for each use case generated. This enables the developer to more easily identify how a particular aspect of the system has been implemented, relating low level functionality to high level goals.

4.7.4. Maintenance and UML

As is apparent from the aspects of UML discussed, a considerable portion of development time in developing a UML model is devoted to creating documentation for what the system is to do and how it is to do it. Each aspect should be associated with a description, and authors have suggested various structuring mechanisms for different features (e.g. Stevens with Pooley, 2000; Quatrani, 1998). While only anecdotal evidence is available at
present, it is anticipated that having this degree of documentation available, particularly documentation that illustrates how core functionality is performed (e.g. sequence diagrams), will be of considerable benefit in maintaining systems.

4.8. **Strengths and weaknesses of OO**

Many prominent authors (Booch, 1987; Rosson and Alpert, 1990; Graham, 1991; Jacobson et al., 1992) have argued that addressing problems using OO techniques can be more ‘natural’ for designers, since an OO design can reflect relevant aspects of the real world. Indeed, this is partially supported by Kim and Lerch (1992), who suggested that OO design might reduce the working memory load for designers, relative to functional decomposition. They claim that functional decomposition focuses the attention of designers on the functional aspects of the problem, and in doing so, requires the designer to remember two sets of information: real-world aspects of the problem domain, and (abstract) constructions in the solution domain. By making the two domains more homogeneous, they suggest that the workload for developers can be reduced, and indeed, many OO designs use representations of real-world objects. In studies of isomorphic problems, Kim and Lerch (ibid.) found that the emphasis placed on abstraction in OO enabled developers to identify the underlying problem structure, as well as producing a considerable time saving. Given the complexity of the programming task, it seems reasonable to suggest that any approach that enables the developer to better manage this complexity is valuable.

Unfortunately, the decomposition of a real-world problem is a contentious area. While Dvorak and Moher (1991) predict that domain experience may enhance the ability to create an OO decomposition of a configuration, OO decompositions will vary according to factors such as the features of the problem considered important and the perceived need for future maintenance. Furthermore, Dvorak and Moher demonstrate that domain experience determines the degree of agreement over a particular class hierarchy. On the other hand, Davies et al. (1995) demonstrated that the classification of code by experts is based more on functional, rather than OO, principles. Interestingly, novices appeared to
classify code in a 'purer' OO way. The implication of this is that OO is not a 'natural' approach to problem solving, as has been claimed. However, it is interesting that a similar effect has emerged with programming languages: the most popular OO programming languages are not those which rigidly constrain the developer to the pure OO paradigm such as Smalltalk, but rather languages such as C++, which utilise both functional and OO concepts. It should be acknowledged however that factors other than technical excellence are also involved, including:

- fashion;
- timeliness;
- marketing; and
- management issues.

Jette and Smith (1989) have argued that OO systems designed with reuse in mind will have a common look and feel, both for the external (user) interface and in the internal (code) interfaces. This can have benefits for the system maintenance task by enabling maintainers to work with a familiar set of components. However, other OO concepts (such as inheritance) can add to the difficulty of identifying the locus of responsibility for a particular task – something that may be addressed with appropriate browsing tools.

4.9. Conclusions

From the preceding discussion, it appears that the OO paradigm embodies many of the features that have been shown to be useful in the development of systems. Development using OO principles is not a universal solution to the issues that face the developer of modern systems. Like any other tool in computing, it is appropriate in particular contexts. It does however reflect several distinct trends in programming and design that have been found to be useful, notably abstraction and encapsulation. As the scale of systems grows, these concepts increase in value as they enable the developer to manage complexity. UML appears to be a reasonably effective approach to development in that it does not explicitly dictate a particular approach to solving problems, although as Green (1989) argues, any notation affects the problem-solving approach taken. Nevertheless, OO approaches are
currently in widespread use and since UML has already received widespread support, it seems appropriate that an OO language and UML should be used in experimentation aimed at investigating more effective approaches to system development.
"C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, it blows away your whole leg."

-Bjarne Stroustrup

5. Java and OO programming

5.1. Introduction

Java is an example of a modern commercial OO programming language that incorporates a variety of new models for implementing OO frameworks as well as supporting existing ones. Just as it is important to select an appropriate language for the study of system design, it is equally important to select an appropriate programming language for the development of research systems. This places considerable emphasis on the selection of a language that is sufficiently widely used to permit a reasonable quantity of experimental material to be obtained. Java is a good general-purpose, OO programming language, which was initially released in 1995 and has rapidly become extremely popular for systems development in both industry and academia. A significant factor in this popularity has been its accessibility, as the main development tool, the Java Development Kit (JDK), is free to download from the Sun Microsystems Web site (http://java.sun.com). Arguably the biggest single spur to the use of Java came from the attention it received as a language for providing functionality for Web pages, enabling developers to create ‘applets’, which are programs run within a Web page, via the incorporation of a Java runtime system into Web browsers.

Sun provides Java with an API (Application Programming Interface), a library of classes which enable developers to easily perform many of the programming tasks that are required in modern system development. These tasks include Graphical User Interface (GUI) development, and networking, which are often more difficult with older languages designed when these features were less prevalent in computer systems. This chapter describes Java in more detail, and looks at how it is relevant to the goals of this thesis.
This is done in three sections. First, the language itself is examined. Next, to emphasise the widespread applicability of Java to different problem domains, some of the areas of application of the language are then described. Finally, the development process for the language itself has been a major reason for its success, and will be briefly discussed.

5.2. Java

In recent years Java has emerged as a viable alternative to C++ (and other languages) for many tasks. Java offers much of the functionality that both novice and professional developers expect, and this perhaps best summarised by saying that Java is ‘buzzword compliant’:

“A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.”


Each of these terms will be described in more detail below, taking into consideration their relevance to the overall goals of this research. Except where indicated, the information for this section has been drawn from the Java White Paper (ibid.).

5.2.1. Simple

As a programming language, Java is fairly simple, with fewer keywords than either C or C++ (Horstmann and Cornell, 1999; Bronson and Walter, 1997). ‘Simpler’ languages can be more robust, since there are fewer opportunities to make hard-to-find mistakes. One comparative study (Phipps, 1999) found that far fewer mistakes were made by developers using Java than those using C++, and that bugs could be fixed faster (it is worth noting that this study used pre-1999 C++; the use of post-1999 C++ could have produced a different result). Unlike C++, Java does not have operator overloading, multiple inheritance, or pointer arithmetic, all of which can be powerful tools but which have significant potential for causing errors. Runtime garbage collection is provided within Java, in order to
simplify memory management and resolve the problems of memory leaking found in other languages such as C++. The syntax and semantics of Java was designed to be similar to C++, and it seems probable that this is one of the reasons for its success, as it enables developers to learn the syntax of the language more easily. It has been suggested, for instance, that a C or C++ developer could get started with Java in about a week (Hamilton, 1999). Java has however been found to present some problems as a ‘first language’ for teaching programming (Hong, 1998). While many of these problems resulted from difficulties in teaching OO concepts, others resulted from the large number of Java-specific concepts that must be grasped by novices at the same time, such as the main method declaration of:

    public static void main (String[] args)

In addition, Java does not provide console input as accessibly as other languages such as C++, due to the need to handle a large number of exceptions when reading data from the keyboard input. Despite such difficulties however, it is enjoying increasing popularity as a first language in computer science courses.

5.2.2. Object-oriented

Like C++, Java is not a pure OO language – it still has primitive types. C++ is an extension of the C language into the OO paradigm, and as a result, carries much of the C ‘baggage’ which does not really apply in an OO context. Java was designed more recently than C++, with the benefit of hindsight, and does not have many of the features that can make coding in C++ more error prone than is desirable.

Java implements single inheritance: a class may only inherit from one other class. Every object in Java belongs to an inheritance hierarchy, and is ultimately a subclass of the Object class. Inheritance in Java enables superclass methods to be overridden, except where this has been prevented using the final keyword. While Java does not permit inheritance from multiple classes (unlike C++), it does enable developers to create classes
which implement one or more interfaces. An interface is similar to a class, except that (Horstmann, 2000):

- an interface does not have instance variables;
- all methods in an interface are abstract: they have a name, parameters and a return type, but no implementation; and
- all methods are automatically public.

A class conforming to an interface must supply the method body for the method(s) used. Interfaces can be a powerful tool in creating a program that is strong and flexible at a conceptual level. This is an important factor in reuse, which is facilitated by interfaces without the technical difficulties introduced by multiple inheritance (Horstmann, 2000).

5.2.3. Distributed

Java has a library of classes for dealing with networking, as well as specialised tools for producing distributed applications, such as the Interface Definition Language (IDL) for use with CORBA (Common Object Request Broker Architecture). In addition, there is a Java-specific version of a CORBA-like framework called Remote Method Invocation (RMI). This enables an object on one system to call a method in an object elsewhere on a network, enabling clients to make direct calls to their servers.

5.2.4. Interpreted

Java source code is compiled to bytecodes and interpreted by the Java Virtual Machine (JVM). This forms the basis for the platform independence offered by Java, whilst providing performance enhancements over ‘portable’ interpreted languages which do not have this intermediate step. While interpreted languages are perceived as not executing as quickly as compiled code, for areas where speed is an issue, Just-In-Time compilers are available (see Section 5.2.9). These give enhanced performance (particularly on embedded systems); however ‘native’ compiled code is invariably faster. While utilities exist for compiling Java bytecodes for particular target platforms, this is achieved at the cost of
platform independence. While there is a constant drive for greater performance in computing, for many tasks, current machines and languages provide sufficient speed, and hence Java can be used in its more portable platform-independent form.

5.2.5. Robust

Like C++ (Stroustrup, 1986), Java is a strongly-typed language (Deitel and Deitel, 1997). This enables a considerable amount of checking to be performed when code is compiled to bytecodes. The reference model of Java also contributes to the robustness of the language, as it "...eliminates the possibility of overwriting memory and corrupting data." (Gosling and McGilton, 1996). While pointers in C and (particularly) C++ can be very powerful, they are also a major cause of errors, because they enable the developer to violate the encapsulation of an object and the address space of the application. This results largely from the different application domain for which C was designed – in the domain of operating system development, this level of control is necessary. Finally, Java enables developers to prevent error conditions within Java programs from crashing the system by handling exceptions. These are a mechanism designed to provide alternate control flows if a recoverable error is generated. Any method which may produce such an error (apart from simple programmer errors, such as accessing an unallocated element of an array) is typified by a ‘throws’ statement, which must be caught by the calling routine or thrown further up the object hierarchy. Java thus becomes a more efficient language within which to develop, and this is reflected in empirical studies (Phipps, 1999).

5.2.6. Secure

One of the main reasons Java has excited so much interest is its role as a 'programming language for the Internet', and its ability to run the same code on different platforms without recompilation. These are powerful concepts, but from a security perspective, they are also potentially dangerous. Java has four layers of security. The Java compiler ensures that code does not violate the internal rules of the language. It has already been noted that strong typing and an alternative pointer model to C++ are major factors in reducing errors. Secondly, the JVM verifies at run-time that all bytecodes obey the same
rules, using additional type information available in the '.class' file. Thirdly, the class
loader prevents the violation of name space and access restrictions by classes, ensuring
that classes cannot (easily) be illegally manipulated. Finally, additional security is
available, via the SecurityManager class, which may be subclassed to provide additional
control over certain potentially dangerous methods, such as those controlling file and
network access. Security in Java is assisted by the publication of the complete source code
for the JDK by Sun, which enables the growing community of Java developers to examine
the software for problems. To some extent this follows the open source model which will
be discussed in Chapter 7.

5.2.7. Architecture neutral
Java is compiled to bytecodes that are interpreted by the JVM. As a result, applications
written in Java can run on any system for which a JVM is available. With the increased
use of networking to connect disparate types of machine together, it is possible to write
programs in Java and be reasonably confident that anyone can download and run the
program, regardless of the type of machine and operating system they are using.
Furthermore, the introduction of the Java Swing toolkit enables developers to produce
systems with a consistent 'look and feel' for all platforms.

5.2.8. Portable
In addition to enabling the creation of code that can run on any system for which a JVM is
available, the JVM itself is standardised. This is facilitated in part by introducing
specifications for the size of primitive data types, which are fixed in Java, and must be
converted to the platform representations when communicating with the host platform.
This contrasts with C and C++, which are implemented in slightly different ways by
different compilers for different machines. Typically this is shown in the different
implementations of primitive types. Standardising the JVM further enhances the
portability of Java code.
5.2.9. High-performance

As noted in Section 5.2.4, an area of concern for developers is the speed of execution of Java relative to other languages. Just-In-Time (JIT) compilers are now available which offer higher performance by translating bytecodes into machine code for the particular CPU (Central Processing Unit) the application is running on. Performance is relative: the (non-JIT) execution performance of Java is adequate for many tasks. Performance must also be considered in a broader context: if a language helps a developer to produce an effective system and rapidly make this system available to users, then execution performance may be less significant.

5.2.10. Multithreaded

The use of multithreading is becoming increasingly important in modern programming. A simple example of the use of threading is a Web browser, which displays a page of information and enables a user to scroll through that page, at the same time as downloading other information. Threading enables programs to appear to perform several tasks at the same time. Consequently, programs written using threads can appear to be more responsive to the user than equivalent programs written without threads. Threading is also how Java addresses concurrency issues, which were explored in Chapter 4.

5.2.11. Dynamic

In programming languages, the way in which external libraries are bound can make a significant difference to the ease with which code is developed. With early binding, which is used in C and C++, external libraries used must be relinked into the application whenever a new version is released. Late binding calls the external libraries when the application requires them, and this is used in Java. Late binding enables faster development cycle times than with conventional development processes since it is no longer necessary to recompile the whole application when part of it has been changed, allowing the developer greater freedom to change the code.
5.3. Java application areas

In addition to the functionality available in the Java Development Kit, the 'standard toolbox' for working in Java, a number of other useful task areas (and associated API's) for the Java language have emerged. Some of the most prominent of these will now be discussed.

5.3.1. Java Servlets

The Common Gateway Interface (CGI) and technologies built upon it have long been used to provide dynamically generated information at Web sites, CGI was intended to define a standard protocol for an information server to communicate with external applications. However, each time a Web server receives a request to be handled by a CGI program, a new process must be created to run the program. This process must then be passed all the information that might be needed to generate a response, as shown in Figure 5.1:

![CGI life cycle diagram]

Figure 5.1: The CGI life cycle (adapted from Hunter and Crawford, 1998)

Needless to say, generating multiple processes uses considerable system resources. Perl (Wall and Schwartz, 1991) is commonly used to create CGI programs because of the ease with which text manipulation (a primary requirement of CGI scripting) may be performed. Perl is however an interpreted language, and so a new interpreter (in a separate process)
must be started each time a program is called, creating a considerable load on the server. While other technologies for CGI are available, these problems are fairly typical.

Java servlets make a significant contribution to resolving these problems. Scalability and efficiency are addressed through the use of a threading model, where each new request is handled by a lightweight thread rather than a new process, as shown in Figure 5.2:

![Java Servlets-based Web Server Diagram](image)

**Figure 5.2: Servlet-based server (adapted from Hunter and Crawford, 1998)**

This makes servlets more efficient and scalable than the comparable CGI implementation. Like Java itself, servlets are portable because they use a standard API. Finally, since servlets operate inside a JVM on the server, they can easily access information held by the server, so it does not need to be passed separately. Servlets are written using the same Java as other applications, just with a specific library and set of interfaces (as with applets), enabling the developer to realise these benefits without needing to learn another language.

### 5.3.2. JavaServer Pages

JavaServer Pages (JSP) enables developers to generate dynamic Web pages by combining markup such as HTML or XML with Java code. JSP is available for a number of popular Web server engines including Apache, the server which (in June 2000) supports over 60%
of the Web (Netcraft, 2000). JSP tags can be embedded within a Web page to dynamically generate the page content.

5.3.3. Swing

One of the most significant developments in Java has been the AWT and the Swing toolkit. Swing provides Java developers with the ability to rapidly create a standard, customised look and feel for applications. For many interface applications, Swing can be considered to be a replacement for the AWT (Abstract Window Toolkit), making Java GUI's far more platform independent in terms of both appearance and portability. By providing this ability, Java addresses one of the basic expectations of modern software development.

5.3.4. JavaDoc

JavaDoc enables Java developers to generate documentation from their code directly from the comments made within the code itself, in conformance to the idea of literate programming (Knuth, 1992), as discussed in Section 3.4. In Java, this is done using a special comment tag:

```java
/**
 * @author Pete Hornsby
 * This is a comment
 */
```

Here, the '@' symbol generates an 'author' tag with the text 'Pete Hornsby' in the documentation. Other standard tags are available, or developers can generate their own by creating another doclet, which is a template for creating documentation files. The standard Sun tool generates HTML documentation, although other formats such as XML can also be produced. Documentation can therefore be produced more easily, and accessed more readily, than would be the case if the documentation were only accessible from within the code.
5.4. The Java Community Process

The process by which Java has been developed is notable since it differs from the ISO-driven (International Standards Organisation) development process of C and C++. This is the Java Community Process (Sun Microsystems, 2000), which has been used since the initial release of Java in 1995 to develop the API in response to the needs of the Java community. This process enables rapid feedback to be obtained from the development community in response to changes that are proposed to the language. As Java is still a comparatively young language, certain API features may be marked as ‘deprecated’ between versions of the JDK. This means that while support for those features is still a part of the language, future releases may not support it. This enables developers to modify their code appropriately while enabling the integrity of the language to be maintained.

At the time of writing, the Java 2 Platform Standard Edition is in version 1.3, and Sun’s Web site (http://java.sun.com) claims over three million downloads for the previous version of the software (1.2). In addition, a number of Web sites are available offering free tools to use with Java, such as IBM’s Alphaworks site (http://www.alphaworks.ibm.com/), and there are a number of Usenet newsgroups for discussion by interested parties.

5.5. Conclusion

There is no such thing as a ‘perfect’ programming language. Only if the problem domain is tightly constrained can there be even a reasonable optimum. Humans are practically limited in the amount that can be learned, and hence it is desirable for a programming language to be capable of easily addressing a wide range of application areas. The issue then becomes, which programming language should the user learn in order to have the most flexible approach to solving problems? Java is a modern programming language which is probably the most platform-independent general programming language currently available, and offers developers a common syntax and development paradigm within which to produce a wide variety of software. Java is not a perfect language by any means, but it is sufficiently strong in areas that are important for modern software development to
be a strong candidate for a general-purpose programming language that can be applied to a wide range of areas, including the development of research systems.
“Simple things should be simple; hard things should be possible”

- Alan Kay

6. End User Programming

6.1. Introduction

End User Programming (EUP) has become a catch-all term for the use of different representations and structures in programming as a means of making the task more accessible. Undoubtedly, one of the biggest spurs to EUP research was the impact that the spreadsheet had on computing. The spreadsheet is a visual representation (on a computer screen) of an accounting spreadsheet. Unlike its real-world counterpart, changing a value in one cell immediately changes the values in dependant cells, enabling calculations to be performed quickly. As well as being visually similar to a real-world object, users receive immediate feedback from their actions, rather than the slower, test-debug-execute cycle of programming. Nardi and Miller (1990) note that spreadsheet users typically lack formal programming expertise, but are nevertheless able to produce functional models of their area of interest. The spreadsheet is a classic example of an EUP system, insofar as it can support users with little or no formal training in programming, enabling them to perform relatively sophisticated calculations using high-level, task-specific functions within the spreadsheet environment. Syntax errors are identified by the system, and the user can immediately see the impact of any changes. The fundamental aspects of the problem solving process itself however appears to be similar to that experienced by problem solvers in other fields:

“When beginning work on a new spreadsheet, users often do not even know what the parameters of a problem are. They only find out about all the relevant aspects of a problem in the process of actually trying to solve it.”

- Nardi and Miller (1990)
While the spreadsheet is a useful and powerful tool, it is interesting to note that a recent study found that 95% of financial models based on spreadsheets contained major errors (Chadwick, 2000). Thus while the impact of the spreadsheet is significant, it in no way represents an ideal model for EUP.

A discussion of EUP is challenging, because the degree of expertise associated with end user programmers can vary so widely. Among other things, end user programmers can be young children learning about interacting systems (Smith, Cypher and Tesler, 2000; Kahn, 2000), non-computing professionals using spreadsheets (Nardi and Miller, 1990), or systems administrators writing small programs in a language such as Perl (Wall and Schwartz, 1991). Nonetheless, EUP is an important area of computing because researchers in this field are attempting to make programming more accessible, and in doing so, are developing tools which differ significantly from those used in conventional programming languages. Researchers have defined EUP in different ways. Cypher (1991) defines EUP as:

“When end-users, who have not necessarily been taught how to write code in conventional programming languages, write computer programs.”

This definition is however of limited value since it does not identify what EUP research aims to do. Nardi (1993) describes the problem EUP attempts to address thus:

“End users...have computational needs and want to make serious use of computers, but...are not interested in becoming professional programmers...In an end user programming system, a critical subset of the functionality of the system can be quickly learned and is sufficient for getting useful work done.”

It is worth noting that Nardi (like the majority of EUP researchers) views programming as primarily functional (from the users point of view; that is, aimed at satisfying goals); hence EUP environments do not support the traditional stages of the development process as
outlined in earlier chapters. The EUP user is interested in the result obtained from the program, rather than the wider concerns of professional software developers such as reliability, efficiency and maintainability. The development environment in EUP is usually considered to be responsible for providing end user programmers with a 'safe environment' within which to work. EUP might therefore be described as a tool for the development of (usually small-scale) systems, where the primary concern is solving a particular problem. Since the end user programmer is not performing an analysis of the problem domain, or going through a design process, then the language and environment through which the system is being written must enable an effective, easily understandable mapping of the problem domain (it should be noted however that users often have very good domain knowledge).

Nardi (ibid.) considers the main problem in EUP to be the provision of tools oriented around the task of the user:

“...end users will freely write their own applications when they have task-specific programming languages embedded in appropriate visual frameworks, and they will write applications in collaboration with other users.”

Nardi later claims that it is the languages in which people are asked to program that leads to difficulties – the ‘right’ language leads to success in programming. As was shown in Chapters 3 and 4 however, the process of constructing a program is a complex task, wherein the developer must move between levels of abstraction in both the problem domain and the solution domain. No single language for design or programming has been found to be ideal for all possible tasks, and to suggest that a ‘right’ language can exist for all users within a particular problem domain – even one which is narrowly defined – is at best optimistic, given the vast differences that may exist between users.

Having end users generating their own programs can be extremely valuable; in conventional software development, a significant amount of time is spent in establishing a
common understanding of the problem between the developer and the user. Such are the problems that result from this communication gap, that Figure 6.1, or variations on this theme, have become well known in software development circles.

Figure 6.1: Systems development (Huczynski and Buchanan, 1991)

As described earlier, EUP is a particularly broad area since the programming skill of the 'end user' can vary so widely. However, it is considered that approaches to programming with conventional language types and within conventional development paradigms are addressed in sufficient detail elsewhere in this thesis to enable this chapter to focus on alternative approaches to programming; specifically, the 'programming by example' and
‘visual programming’ styles. Programming By Example (PBE) enables the user to record sequences of actions to be played back at a later date, and will be discussed in Section 6.3. Visual programming systems will be considered first. Such systems attempt to engage the user primarily through iconic representations, rather than the textual mode used by most conventional programming environments.

6.2. Visual programming

Visual programming researchers claim that programming using a visual medium can make the task more accessible by bringing the external representation of the problem closer to the internal representation used by the human (Shu, 1992). It is interesting to note that proponents of object orientation have claimed similar qualities for OO (e.g. Booch, 1994). However, Chapter 2 suggested that professional programmers (like other human problem solvers) use many different internal representations depending upon the task, and identifying an appropriate visual representation for all potentially relevant concepts in computing would be practically impossible! EUP researchers frequently refer to the work of Jerome Bruner (1966), who claims that there are three modes of thinking:

- **enactive**: learning through doing;
- **iconic**: learning and thinking using pictures; and
- **symbolic**: learning and thinking using symbols.

Bruner does not make a judgement about which of these modes is ‘best’; rather, he states that different modes are appropriate at different times. In support of this, Green (1989) suggests that any notation is useful only in relation to certain tasks. This is further supported by Whitley (1997), who found that a visual programming language could improve performance if the language is appropriate both to the individual and the task. Whitley also claimed that Visual Programming Languages (VPLs) lack a strong cognitive or empirical basis for support. VPLs have however had a significant degree of success in educational programming systems, as well as some take-up of the ideas (if not the systems) in industry. The next subsection will examine a number of visual programming
environments, looking at the knowledge required to use them, and their effectiveness within those domains.

6.2.1. LabVIEW

LabVIEW (National Instruments, 1994) is a popular programming tool used in electrical engineering, primarily for the creation of simulations. It conforms to Nardi’s (1993) description of an EUP environment since it makes use of a representation (electrical notation) known to the target user population. Figure 6.2 is a block diagram taken from a LabVIEW application, designed to record and display temperature data:

The figure shows that LabVIEW programs are created by ‘wiring together’ visual representations of elements on the screen. In addition to these visual programming features, LabVIEW also offers more conventional programming concepts such as hierarchies, arrays and strings. While this may be seen as a means of offering the benefits of both a VPL and a conventional programming language, it also means that effective use of the system depends upon knowledge of both circuit notation and programming concepts. Baroth and Hartsough (1995) compared the development of the same product
by a team using LabVIEW and a team using C. In three months, the LabVIEW team went beyond the original requirements in their completed product, while the C team had not finished addressing the requirements. It is also worth noting however, that a major factor in this may have been that communications between the developers and the customers were facilitated through the use of LabVIEW, as the customers (also engineers who understood the LabVIEW syntax) were able to contribute more effectively through a greater understanding of the system under development.

6.2.2. Java Studio

The Java Studio product (Weaver and Robertson, 1998) has not had nearly the same degree of take-up by the industrial or academic community as LabVIEW, and was discontinued by Sun after only two years in favour of a generalised version of the underlying JavaBeans technology. Despite this, the concepts behind Java Studio are interesting in their own right.

Figure 6.3: Example of a program designed using Java Studio
As with LabVIEW, Java Studio (illustrated in Figure 6.3 by a simple arithmetic program) uses a visual metaphor, whereby systems are created by connecting components, the settings of which may be modified via dialogue boxes. While Java Studio is based on Java, no understanding of the Java programming language is required to produce systems. Much like a spreadsheet, changes made to one part of the system are immediately reflected elsewhere. Again, as with LabVIEW, certain aspects of programming logic are reflected in the available components, such as FOR loops and IF statements. While the Java Studio notation is not based on an existing (domain-specific) notation type, the concept of entities wired together may be useful in illustrating the underlying OO concepts of encapsulation and message passing, possibly as an educational tool.

What is interesting about Java Studio is that the environment is a ‘halfway house’ between a VPL and a conventional language, since it can be expanded by adding more JavaBeans to those already available. JavaBeans are a component technology from Sun Microsystems which are discussed in more detail in Section 7.6. Although specialised knowledge is required to create a new JavaBean, Java Studio supplies around 50 JavaBeans as standard, which may be used to create a fairly wide range of systems. A product such as Java Studio could potentially have bridged the skills gap between professional programmers and end user programmers, by enabling end users to use JavaBean components generated by professionals. However, its lack of popular support (resulting from deep technical flaws) prevented this.

6.2.3. AgentSheets and the role of analogy

One of the stated aims of EUP is to make programming easier, and as with conventional programming, reuse of existing systems would be a significant step towards this. AgentSheets attempts to address this problem through the use of analogy. End users are known to employ reuse where possible (Greenberg and Witten, 1988). In a visual environment, a major problem with implementing reuse is dealing with abstraction. Visual programming helps to make the abstract more concrete — moving the problem representation from (abstract) code to a representation which the user may be more
familiar with. Reuse is based on a different principle. The developer abstracts from the problem to obtain a representation that may be applied to different situations. So rather than saying ‘cars move on roads’ and ‘trains move on tracks’; one might say ‘objects move on surfaces’ and be able to substitute the desired objects and surfaces. However, just as Donaldson (1978) suggested that children find it difficult to work with abstract concepts in certain circumstances, so adults similarly can have problems abstracting from a problem in certain situations. This is not least because it may not be apparent what the most appropriate abstraction is, and (particularly if the future development of the system is unclear) selecting an inappropriate abstraction can make future development more difficult. This can be seen as similar to the problem in OO development of finding a breakdown of the problem that satisfies as many of the success criteria as possible. This leads to what Repenning (1991) calls the ‘Representation Cliff’—a cognitive gulf between the representations used in the EUP environment and the representations used in a conventional programming language.

AgentSheets addresses the problem of a potential cognitive gulf through the use of analogy (see Figure 6.4). In cognitive terms, analogy is a mechanism used to construct new knowledge from knowledge that has already been acquired (Repenning and Perrone, 2000). The mechanism used to create an analogy cannot be simply syntactic substitution as in the trains and cars example above, but must involve semantic information to identify the relationship (ibid.).
Figure 6.4: Describing analogy using the AgentSheets environment

Analogy can be a powerful tool, but for end users, the very nature of abstractions makes them difficult to represent visually – how can one represent an abstract moving object, or an object which can be driven? Additionally, how can it be ensured that the analogy is appropriate, in human terms and for computer interpretation?

6.3. *Programming by Example*

Programming by Example (PBE) systems (also known as programming by demonstration) are based on the idea that once the user knows how to perform a task, the system should facilitate its automation. Repenning and Perrone (2000) say that:

“Programming by Example (PBE) is a powerful end-user-programming paradigm enabling users without formal training in programming to create sophisticated programs.”
PBE systems are most commonly seen in macro recorders, such as those present within the MS Office application suite. These enable users to generate scripts (sequences of commands) by demonstration, and replay these scripts on different data sets. Scripts may be modified using the MS Visual Basic Editor, and moderately sophisticated sub-applications may be built using this approach.

Another PBE system is the EAGER tool created by Cypher (1991). EAGER operates in the background of the user's main task, highlighting screen objects when it feels that it has identified a sequence of commands being repeated by the user. Cypher noted that users of the system were uncomfortable with giving up control when EAGER took over. However, subjects did realise what EAGER was doing without being told. The principles of EAGER may be seen in the Office Assistant that is now part of MS Office, which generates assistance based on the user's actions.

6.4. EUP in teaching

EUP principles have also been used to teach about existing systems. ‘Self-disclosing’ tools present the user with a command-line alternative to mouse commands. This approach can enable the user to learn about scripts (to automate sequences of commands) while performing their primary task. Commercially, this has been implemented in the AutoCAD (Autodesk, 1992) design tool. DiGiano and Eisenberg (1995) propose six guidelines for self-disclosure:

1. disclosures should be maximally generalisable;
2. the system should facilitate experimentation with disclosures;
3. more complex expressions should gradually be revealed;
4. essential programming concepts should be addressed;
5. it should be possible to specify operations using both direct manipulation and text commands; and
6. disclosures should be unobtrusive and browsable.
Like Java Studio, the C\textsuperscript{2} programming environment augmented an existing programming language with a visual interface (Kopache and Glinert, 1988). This used a subset of the C programming language, and could represent information as textual C code or simple diagrams. Here, combinations of shape and colour were used to represent a type hierarchy. The ‘Explainer’ tool (Fischer et al., 1994) is also notable, since this provided multiple views of a LISP library of graphics functions. A view of a system could comprise a sample code listing, sample execution, component diagrams, and text.

6.5. Discussion

In its broadest form, EUP is concerned with making programming more accessible to people who are not professional programmers, but there are practical limitations to how far this may be taken. The end user must be kept aware of what the computer is doing, and how it is doing it, insofar as it relates to the domain of interest. The problem is largely one of communication: how can the user describe their problem in a way that is acceptable to the needs of both the user and the computer? Attempts have been made to use natural language as the basis for system development. Such systems typically use a small subset of natural language that must also follow well-defined syntactic rules. However, this can increase the user’s expectations to the point where they exceed the capabilities of the system and hence lead to disappointment. Natural language is unlikely to offer any solutions in the short term, since even in human-human interaction, the process is extremely time consuming, and often unreliable. This results from the fact that the nature of the human’s involvement with a problem has a major influence on how it is perceived, and thence described. As a result, any human using a programming environment must be able to understand what is happening in sufficient detail to be able to modify the system through the programming environment.

A significant motivator for PBE systems is the concept of the end user as a teacher. It has been argued (e.g. Liebermann, 1993) that the act of teaching deepens the teachers understanding of a topic. However, this can also be the downfall of PBE systems. It can be argued that the success of PBE is dependent on the selection of good examples; that is,
examples which unambiguously illustrate the idea which the user is attempting to communicate to the computer.

There is a significant body of evidence (discussed in Chapter 2) to suggest that the problem representation has a significant impact on the effectiveness with which solutions can be found. This author does not believe that 'ideal' representations for any problem or indeed, group of problems exist. As this and previous chapters have shown, humans make use of many different forms of representation depending on the task. This suggests that multiple representations for flexible, productive programming languages may produce an effective approach to making the programming task more accessible. Whereas Nardi (1993) has argued in favour of task-specific programming languages, this invokes the subsequent problem of where the boundaries of a particular task should be drawn. Few (real-world) problems fit into well-defined domains, and this is considered to be the main factor against orienting programming environments to particular problem areas.

Although reuse can be a powerful tool, it is contingent upon users understanding the program representation. Where programs are machine generated, such as with PBE techniques, none of the human clues that can make programming easier such as meaningful variable names or code comments are present, since these are created from the human problem solvers' state of understanding of the problem and its solution. Repenning and Perrone (2000) comment that:

"The general insight we derived is that a little semantics is necessary for creating meaningful analysis. This means that end users have to provide some additional up-front information to annotate their designs with minimalist semantics. It also means that this kind of semantic annotation dramatically improves the reusability of behaviour."

88
6.6. Conclusion

End user programming has demonstrated that problem solving with a computer can be made easier by using the skills and knowledge users already possess. Although this approach can be successful, it is limited to the capabilities of the particular EUP system and the ease with which ideas may be expressed in the available notation. It is in the degree of expressiveness where many PBE and visual programming systems fail. Many ideas and concepts do not have a 'natural' visual representation. Where PBE systems are used to introduce children to programming, the means by which programs are constructed rarely allows a 'natural' progression to 'regular' programming languages. The scalability of PBE systems is also questionable. If a user wishes to perform a task that is 90% possible within environment $x$ and 10% possible within environment $y$, they will not be able to create a solution if only one of the environments supports PBE.

The perception of EUP systems and components appears to be that the former are for novices, while the latter are for experts. The reality is that common ground can and does exist, such as the JavaBean-based Java Studio, or the increasing use of visual representations in conventional programming tools, such as IBM's Visual Age. Nardi (1993) argues that Apple's HyperCard with HyperTalk is a compromise between a general programming language and a task-specific language. However, this is a poor compromise since such systems have the complexity of conventional programming languages but lack speed. Additionally, they are not close enough to end users needs to provide the right kind of task specificity, nor are they general enough to be as powerful as a conventional language. Clearly, the choice of when and how to integrate is a difficult one.

The effectiveness of a visual environment is limited to information artefacts that may be visualised effectively. For certain programming concepts such as data structures, no appropriate visual representation may exist. As with any metaphor, the effectiveness of a visual programming environment depends upon the user's familiarity with the source knowledge domain. An important question is therefore whether there is an unavoidable power / accessibility trade off: whether a metaphor capable of powerful manipulations is necessarily accessible to only a small user population. One way in which this has been
addressed is through a programming environment that can represent systems as text or graphics. Davies et al. (1995) called for a similar mechanism with regard to object technology, acknowledging that different representations were relevant to different stages of a task. Recently, software development environments (such as the Rational Rose design tool) are beginning to offer the capability to view a given system in different ways.
"Specialists in every part of software have a curious vision of the world: All parts of software but his are simple and easily parameterised; his is totally variable."

- Alan J. Perlis

7. Components and reuse

7.1. Introduction

In computing, as in many other disciplines, there is very little that is new. Developers often report a sense of déjà vu when creating software; a sense that a design or code to perform a task has already been written, either by themselves or by someone known to them. This can be particularly frustrating when tight deadlines must be met or resources are strained. One way in which this problem can be addressed is through the reuse of design and code. This concept of reuse is generally associated with the notion of software components. McIlroy (1969) initially suggested these at the NATO software engineering conference of 1968 (Naur and Randall, 1969). McIlroy’s paper predated OO technology, and very much reflected the software development concerns of the time, placing great emphasis on robustness and the efficient use of resources. These concerns led McIlroy to emphasise the need to be able to tailor components to individual user requirements. McIlroy recently (2000, personal communication) said that this perceived need to tailor components was a failure of vision, noting that ‘one size fits all’ is usually acceptable. This is significant in that it reflects the changing priorities of the field of non-embedded software development, away from an emphasis on efficiency at the machine level (as noted in Chapter 4) and towards faster development of more reliable software. Ledbetter and Cox (1985), who suggested that the performance overheads of reusability were sufficiently great to demand a certain minimum hardware speed, support this view.

The importance of efficiency across the whole development process is dependent upon the nature of the software and its area of use. An application such as an embedded controller
or a safety-critical system may have efficiency concerns in terms of operational performance, a small memory size, or both. This author is therefore unwilling to concede that efficiency is no longer an issue in software development. However, it is also true that run-time efficiency on many platforms can be less stringent at present than has been the case in previous years. To address this, a broader view is suggested where efficiency is seen as a concern of the whole software development process, rather than simply a run-time issue. Whilst run-time resource efficiency can be a concern, there is a limit to how much development time can be justified in the pursuit of this goal. Herein lies a perceived trade-off between the speed of development and the efficiency of the code.

Component reuse is regarded as a powerful technique for improving the development process by the software development industry, and can create both benefits and problems for the organisation that chooses to implement such a strategy. The National Aeronautics and Space Administration (NASA) developed a component library for the Mars 'Pathfinder' missions (Condon et al., 1996). Although the creation of this library took 76,000 hours, NASA estimates that they will recoup this initial investment by the 4<sup>th</sup> mission. Despite this and other public successes, software reuse can cause problems if insufficient safeguards are in place: it was found to be a contributing factor in overdoses given to some patients treated by the Therac-25 radiation treatment machine (Leveson, 1995).

7.2. Component reuse for software engineering

It has been argued that "...components are the way to go because all other engineering disciplines introduced components as they became mature..." (Szyperski, 1998). Section 3.3 has however discussed the limitations of the application of an engineering viewpoint to software; to summarise, the process of software development involves the construction of the basic tools with which information-based problems are described. In essence, two areas of concern must be considered. There are the technical demands of the system implementation: limited machine resources, the need to design code to take advantage of the developers skills, the implementation language, and future maintainability. In
addition, the real-world problem and the constraints it imposes must lead development. Fischer (1987) describes these two areas of concern as the situation model and the system model: the real-world problems, and the operations that will result in the desired solution. The design task must therefore be concerned with both viewpoints; not least because, as Chapter 2 noted, the developer’s perception of both of these will change as the problem is addressed.

A major risk in component reuse is that the incorporation of existing components can have undesirable implications for the rest of the software system; in a worst case, this can result in a need to make major changes to the organisation. Szyperski (1998) reports that the introduction of the SAP R/3 system into the Australian Post Office caused major problems because of differences between the organisational structure of the Post Office, and the assumptions that had been designed into the software. The Australian Post Office is a federated organisation; however the R/3 software only supports a monotonic hierarchy of access authorisations. Szyperski reports that upon being told about this, SAP commented that “Our system implements best practice - why would you want to deviate from that?” This illustrates the difficulties associated with applying standard solutions to real-world problems: such problems are typically wicked (as described in Chapter 2) and thus rarely have solutions that can be so broadly applied. Is component reuse therefore a feasible goal? Successful projects that have made use of it would appear to indicate that these issues can be overcome, in some circumstances at least. However, it is important to be aware of the complexity of component reuse in software development relative to the physical engineering domains.

7.3. A definition of component reuse

Component reuse can be a complex field of study due to the claims that are sometimes made about it. Depending upon which author is consulted, component reuse can be a case of ‘Been there, done that’ (Poulin, 1999), or a technology that will succeed where OO has failed (Udell, 1994). The disparity in the perceived success of components appears to stem from the lack of a common understanding about what a component is, both within
and between the academic and industrial software development communities. Within the field as a whole, the term 'component' has been defined in many different ways:

"...a binary module of code whose interface is defined by properties, methods and events...any piece of a software that you want to treat as a standard-issue "black box" chunk of functionality at design-time."

- Konopka (1996)

"...a reusable software component:
- provides a vehicle for formally expressing data structures and algorithms
- supports the software engineering principles of abstraction, information hiding, modularity and locality
- exploits the facilities of programming languages such as ADA
- offers a mechanism for the reuse of software"

- Booch (1987)

Sommerville and Wood (1989) define six classes of component:

- Functions;
- Procedures and value-returning procedures;
- Declaration packages;
- Objects;
- Abstract data types;
- Sub-systems.

Finally, Szyperski (1998) claims that:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."
More recently, delegates at the TOOLS2000 conference (Mitchell et al., 2000) were unable to agree on a common definition of the term. While a cynic might conclude that the field is fragmented and unworthy of further study, the potential benefits of reuse are sufficiently great to continue to attract attention from both academia and industry, spurred on by success stories such as that reported by NASA.

While the available definitions of 'component' are wide-ranging, the central theme is of software development through the composition of reusable elements. Working from this description, a software component may tentatively be described as any fragment of code that may be reused with sufficient ease by the developer to justify the effort involved in the reuse. For a component to be reusable, the effort required to understand it, in addition to the time taken to perform any required alterations, must be less than the time to design and implement a custom solution. Adolph (1999) suggests that this point occurs when more than 20% of a component needs to be modified; when this limit is exceeded, it is more efficient to start from scratch. However, this is a largely subjective judgement that is contingent on the skills and experience of the developer concerned.

Earlier chapters noted that the early stages of the design process were vital for the long-term success of a development project, and that effort expended here could reduce the costs of later stages of development. By the same token, the reuse of design artefacts can be one of the most effective forms of reuse. Taking this into account, to facilitate a broader examination of the subject, the term component will be defined here as:

"Any tangible product of the software development process that a developer chooses to reuse in whole or in part, in preference to creating a functionally similar product."

Whilst this definition is broader than those quoted above, it is considered desirable to base a definition of the term on how reuse occurs in practice, rather than providing a more narrow definition that fails to encompass the complete development process, or reuse as it
is already practised in an ad hoc fashion. Functions, classes, design patterns and use cases would all be examples of components given this definition.

The remainder of this chapter will examine component-based development. It will start by looking at how market forces have influenced the field, before examining currently popular forms of component reuse. This will then be followed by an examination of the forms of design and code reuse that are possible, together with the mechanisms that can be used to support them.

### 7.4. Commercial factors in component reuse

It is held to be self-evident that commercial factors are a major driving force in the development of computing. These market forces are also a major influence on the development and adoption of component technologies because of the cost savings they (potentially) offer through the reuse of components, as well as the potential to create a profitable marketplace for components, as initially suggested by McIlroy (1969). Traas and Hillegersberg (2000) noted that despite bold predictions of a component market worth US$64 billion by 2002,

> "The Internet component marketplace is still in its infancy. The number of producers that sell the currently leading component standards (ActiveX and JavaBeans) over the Internet should be counted in dozens rather than hundreds."

Component marketplaces are available, such as Flashline.com (http://www.flashline.com) and Objectools.com (http://www.objectools.com). However, component vendors have been slow to emerge, and those that do exist are fairly broad in what they consider to be a component. The Director of Product Development for one component vendor recently claimed that selling components was difficult because a true component marketplace did not yet exist (Bunting, 2000, personal communication).
Broader commercial factors have a major influence on component development. As Adolph (1999) notes, a reusable component can cost between three and five times more to develop than a conventional equivalent created for a single set of environmental conditions. This is because to be reusable, a component cannot make simplifying assumptions about its environment. But in computing, the speed with which a software product can be developed is a major factor in its probable success. A product that is released first to the marketplace can establish itself as the standard, with a resulting impact on the profitability (and survival) of the company. While one company may spend time creating a component library to achieve long-term gains, another may gain a commercial advantage by avoiding the additional costs of developing a reuse library and so being first to market. Market forces can thus implicitly militate against the development of a component library. The NASA example described earlier took place within a non-commercial organisation, and was thus free from these pressures. To address this problem, Poulin (1999) proposes establishing a Reuse Development Team within an organisation to create a foundation of shared software before development begins. While Poulin claims average reuse rates of 23% with this approach, organisational change is frequently traumatic. Furthermore, this approach would appear to be limited to companies of sufficient size to have multiple projects underway and with sufficient staff to form a Reuse Development Team. This is therefore likely to exclude many small and medium sized enterprises.

Political considerations must also be addressed when considering standards for component reuse. Indubitably, standards are important for developing a commercial basis for component reuse since they enable components written by one organisation to be reused by another. While documented evidence is difficult to come by, anecdotal evidence suggests that intra- and inter-company politics can sometimes prevent standards issued by one company from being used by another company. One approach to address this problem is exemplified by the Object Management Group. While this has the support of many vendors, it does not have a vested interest itself in selling products, and is therefore a suitable body to manage standards for component interoperability. This is demonstrated by CORBA, which will be described in Section 7.7.2.
7.5. Types of reuse

Chapter 3 discussed the importance of the early stages of design; the same arguments have been used to note the importance of being able to reuse design components. This can be complex however, and the reuse of code has been more widely accepted. This section examines the reuse of both design and code.

7.5.1. Design reuse

Adolph (1999) notes that most software professionals maintain a ‘mental backpack’ of concepts that have previously been used in software development. Unfortunately, such personal experience can be difficult to share, and this may be reflected in the high need for developers to communicate that was noted by Greenbaum and Kyng (1991). The contribution of design patterns to the field of reuse is to enable designs described at a high level of abstraction to be shared among the software development community. A design pattern is an important, recurring design (Gamma et al., 1995). It must recur sufficiently often for its utility to be noticed, and to be able to identify the environmental considerations influencing the use of the pattern. Design patterns consist of four parts in the Gamma et al. conception:

- a **pattern name**: a means by which a design pattern can be discussed by developers;
- a **problem context**: a description of when a pattern can be applied;
- a **solution**: the elements that make up the pattern; and
- **consequences**: the results and trade-offs of applying the pattern.

A design pattern is a high level abstraction: it can be applied many different times without using the same low-level implementation. It is therefore the responsibility of the designer to determine when and how to apply a particular pattern. Patterns can benefit developers by enabling what is effectively the results of experience to be shared in a common ‘memory’, for example via conventional print media or Web sites. Patterns have also enabled communication between developers to take place more easily, by providing a
language of discourse for discussing designs. Possibly the biggest single factor influencing the success of patterns is that they have endowed developers with the ability to apply experience between disparate projects. While the effective use of patterns is largely the preserve of experienced developers, similar behaviour may be seen in novice programmers, who generalise from examples to simpler programming structures known as idioms (Anderson et al., 1984).

A related field of interest is AntiPatterns (Brown et al., 1998). These are commonly occurring problems in software development, which follow a similar structure to design patterns and similarly facilitate discussion. Brown et al. (ibid.) state that:

"An AntiPattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences. The AntiPattern may be the result of a manager or developer not knowing any better, not having sufficient knowledge or experience in solving a particular type of problem, or having applied a perfectly good pattern in the wrong context."

AntiPatterns are important in that they use the same principles of high-level, abstract descriptions of situations, derived from experience, which can be shared between developers and software development managers in order to identify and ameliorate problems in the development process. AntiPatterns differ from design patterns in that they are concerned with the management of the design process as well as issues within the design task itself. It is interesting to note that the concept of design patterns relates strongly to that of plans: mental constructs which cognitive scientists propose are used by expert programmers to structure their solution strategies (e.g. Soloway et al., 1988).

Both design patterns and AntiPatterns are notable in that there is no explicit tool support for them: their successful use is dependent upon the skills and experience of the developer. Developers can access them in three ways: from personal experience, through textbooks, or via on-line repositories such as the Patterns Home Page (http://hillside.net/patterns/).
Ultimately, it is the developer’s discretion that must be used to judge the applicability of a particular design pattern or AntiPattern to a problem, and their experience that must be used to apply it effectively.

7.5.2. Code reuse

Code reuse is possibly the most widely understood and practised form of reuse. It can be a purely ad-hoc process, a systematic approach supported by organisational processes and tools, or some intermediate form. This section will examine how reuse takes place, before some of the tools that are available to support the reuse process are described.

7.5.2.1. Ad hoc reuse

A considerable amount of reuse in software development takes place on an ad hoc basis. Developers may copy, paste and then possibly modify code, or reuse elements of existing designs, after a visual or mental search. Many developers have a ‘toolkit’ of code fragments and existing designs that they have built up in the process of creating systems, to be used in future development projects. Such reuse can be difficult to study, because it is highly dependent upon the developers experience and skill to identify opportunities for reuse based on their knowledge of the components available. The degree to which the developer has been personally involved in creating the components may well be a significant factor in promoting reuse, since the developer should understand the capabilities and limitations of the components at a low level to a greater extent than bought-in components. In the same way, the developer may (and often could) reuse elements of an existing design.

7.5.2.2. Reuse from a component library

Libraries of components are possibly the oldest semi-formal approach to reuse. Programming languages such as Java and C++ have standard libraries that are typically perceived as a fundamental part of the language and which enable many common tasks to be performed with relative ease. Other libraries of code may develop over time to perform
tasks not addressed in the standard libraries, such as the Standard Template Library (STL) for C++, which provides facilities for many basic algorithms and data structures (and which has now been incorporated into the C++ language standard). As well as the STL, commercial libraries are also available, such as that offered by the Numerical Algorithms Group (NAG: http://www.nag.co.uk). In addition, the computing community makes considerable use of 'open source' tools, such as the Comprehensive Perl Archive Network (CPAN), at http://www.cpan.org. The Open Source pages at http://www.opensource.org state that:

"When programmers on the Internet can read, redistribute, and modify the source for a piece of software, it evolves. People improve it, people adapt it, people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing."

[Original emphasis]

Making the source code for software freely available is a challenge to conventional modes of commercial software development, and by implication, to the development of a component marketplace. However, from another perspective, the easy availability of open source components is beneficial to software developers since it provides code which can be used freely and which is likely to be reliable. In addition, many open source developers provide contact details for the users of their systems so that they can become aware of problems and thus address them. This is in sharp distinction to developers of commercial systems, most of whom guard the inner workings of their products closely. Despite this disparity, 'cross-fertilisation' is taking place between open source and commercial software development. For example, in 1998, Netscape released the source of its Web browser, and IBM uses the open source Apache Web server as the core of its WebSphere product line (O'Reilly, 1999).
7.5.3. Data reuse and XML

While it is a comparatively recent development, the Extensible Markup Language (XML) makes a different and particularly significant contribution to reuse. Whereas the technologies described thus far are significant in terms of the design and coding of systems, XML offers developers the ability to reuse and share data more effectively between applications. Consider a group of three applications. If each uses a bespoke data format, then for effective data exchange, each application must be able to read data in 2 other formats. This is illustrated in Figure 7.1:

![Figure 7.1: Problems with sharing data](image)

Clearly, the more applications there are that must communicate, the more complex each becomes. XML offers a potential solution to this problem. XML is a subset of SGML, the Standard Generalized Markup Language. This is a much larger language than XML, and is correspondingly more complex to use. While XML is not the first attempt at creating a general language for information exchange, it is undoubtedly one of the most successful in recent years.

An XML document looks similar to a document written using HyperText Markup Language (HTML). They differ in that HTML describes the logical structure of a document using a series of standard nested tags. XML uses user-defined nested tags to describe the logical data in the document, and XML documents can be readily translated from one structure into another. This should enable data described using XML to be
shared more easily with other applications than data described using a proprietary data format.

7.6. Domain analysis and reuse

One of the most popular approaches to reuse is by means of Domain Analysis (DA). As mentioned earlier, Poulin (1999) has argued that DA is the key to achieving greater than 20% levels of reusability. DA was cited as the basis for the high levels of reuse achieved in the NASA example (Condon et al., 1996) described earlier. DA is an attempt to generalise all systems in an application domain by way of a domain model that transcends all specific applications (Prieto-Diaz and Freeman, 1987). DA is similar to the systems analysis stage in software development, except that instead of developing the requirements for a single system, an attempt is made to define knowledge for a group of systems that will operate within a particular problem domain. It can therefore be seen that it is similar to the spirit of OO development in that it will ultimately create a model of the real-world elements of interest, but in this case it is creating tools that can be applied to a range of problems within a particular domain.

In operating at a higher level of abstraction than systems analysis, DA enables the identification of common (and thus reusable) aspects of a particular application domain. Neighbors (1984) pioneered this approach with the Draco system, which used DA to focus on the reuse of analysis and design information. Draco enabled systems in new domains to be described in terms of existing domains.

Domain analysis is itself a difficult and time consuming process, entailing the creation of a high-level body of knowledge. As a result, it is frequently impractical for organisations to adopt it as market forces demand that a product is created and shipped. In addition, as Neighbors (1984) notes, each domain typically exists at the intersection of other domains. Hence, it can be difficult to identify what knowledge a domain should encompass, and consequently, a major task of the domain designers becomes the organisation of these domains of knowledge.
7.7. Tool support for component reuse

Despite a lack of common agreement over a definition for the term component, a number of approaches have been claimed to be component technologies. This section briefly describes the strengths and weaknesses of two of these. The following section looks at existing solutions for the task of retrieving appropriate components for reuse. Two solutions have been examined: JavaBeans, which are components in their own right, and CORBA, which is a standard for the ‘glue’ between systems.

7.7.1. JavaBeans

JavaBeans is an approach to component-based development using the Java language (described in Chapter 5). A JavaBean can contain a number of classes, as well as other resources. They may be combined in order to create complete programs, as well as integrating with other component models such as Microsoft’s ActiveX. Software created using JavaBeans can be used in ‘containers’ such as Lotus Notes and Microsoft Word. JavaBeans may be visually assembled using assembly tools such as Java Studio (discussed in Section 6.2.2), and (like the Java language itself), can be used to create both applets and applications. JavaBeans are a particularly significant technology because they enable developers to write programs which, ultimately, interoperate across different platforms, operating systems and networks with greater ease than is currently possible. It is also possible to buy JavaBeans at a number of sites on the Internet, such as FlashLine (http://www.flashline.com) and ComponentSource (http://www.componentsource.com/java/). While this is not yet the profitable component marketplace predicted by Traas and Hillegersberg (2000), it does suggest that the JavaBeans approach is sufficiently pragmatic as a component model to be interoperable between different organisations.
7.7.2. CORBA
CORBA, the Common Object Request Broker Architecture is, like UML (described in Chapter 4), a standard of the OMG. It is an architecture designed to enable software applications (potentially written in different languages) to work together over networks. Using a standard protocol called IIOP (Internet Interoperability Protocol), a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network (OMG CORBA FAQ, 2000). CORBA is thus a technology for interoperability between platforms; it provides a 'glue' between the (distributed) components that form a system. By facilitating interoperability, the use of components is itself promoted.

7.8. Component retrieval systems
The value of any retrieval system is based upon the ease with which users may retrieve items of interest to them. By implication, this places two main demands on the retrieval system. These are:

1. To enable information to be added easily.
   - This is of fundamental importance in making the system attractive to reusers, since it directly affects the amount of information contained in the system, and thus, the likelihood that the system will contain a component of value to the user. It is of particular importance for an application area where the quantity and type of information changes rapidly.

2. To enable information to be retrieved easily.
   - Retrieval is often viewed as the most significant function, but it is contingent upon the previous point. The effectiveness of retrieval is based largely on how well the user can use the available retrieval mechanisms, and how motivated they are to use these.
Having established the main demands upon retrieval systems, two such will be discussed, each of which takes a different approach to the problems of effective retrieval within software development.

### 7.8.1. Faceted retrieval

Prieto-Diaz and Freeman (1987) proposed a classification scheme for software components based on the faceted classification method used in library science. Under this scheme, a group of elemental classes are defined, which are referred to as facets. A stored item can belong to multiple facets, any number of which may be used as the basis for retrieval. Additional facets may be added in response to changes in the needs of the retrieval system. The facets themselves are also classified, based on the application domain. Prieto-Diaz and Freeman selected six facets as a basis for component retrieval. These were:

- Function;
- Objects;
- Medium;
- System type;
- Functional area; and
- Setting.

These were chosen as pertinent to the application domain of reusability. The system uses a controlled vocabulary in conjunction with a thesaurus of terms, where the term which best describes the concept is selected as the representative term. One of the most interesting features of this approach is a weighted conceptual graph to measure the closeness of the terms in the facet, although the authors acknowledge that the construction of the conceptual graph is time-consuming. While the overall structure of the graph remains relatively stable as the size of the repository grows, some tuning in response to user feedback does take place.
Despite its apparent sophistication, the faceted approach has been found to be less effective than free-text retrieval. Mili et al. (1997) examined the problem of component retrieval both in terms of retrieval success, and the costs of setting up the component library and associated indexing systems. By using automated indexing for a free-text retrieval system, greater success at retrieval was found than was achieved with manual, controlled vocabulary indexing. Mili et al. suggest that multifaceted retrieval is at the wrong level of formality for the majority of developers, and identify two stages in searching. During the first stage, developers do not yet know the form that the solution will take. This may be seen to support the work of Schon (1983) and Guindon (1990) discussed in Chapter 2. During this stage, multifaceted searching may be too constraining, as the developer must select the categories to which the perceived requirement belongs, at a time when they may not be able to reasonably make this decision. In the second stage, when more is known about the components and their interrelationships, more information is required about components than is provided by multifaceted classification. It is interesting to note that this growth in knowledge about the components (the solution domain) reflects that described by Schon (1983).

7.8.2. CodeFinder

A purist software design view would put forward the perspective that a considerable amount of initial design work must be done in order to organise the information contained within a repository, before the implementation of a system. While this may be true for the architecture underlying the system, the relationships between the terms used in the system (that is, the knowledge structure) does not need to be defined initially. The CodeFinder system (Henninger, 1996) takes the latter view, and offers a different approach to component reuse. This is based on the concept that the understanding that a (re)user has of a problem changes as problem solving progresses. This is in sharp contrast to the view of Prieto-Diaz and Freeman (1987), who consider a functional specification to be the beginning of the reuse process. As noted in Chapter 3 however, in practical software development requirements evolve during the course of the project, and the approach of
Henninger appears to reflect this more effectively than the approach of Prieto-Diaz and Freeman.

The initial costs of setting up a repository of components can be a major barrier to their use within organisations. CodeFinder addresses this problem in two ways. In the first instance, the addition of components can be automated through a tool called PEEL (Parse and Extract Emacs LISP). This extracts components from text files and uses both interactive user support and automatic extraction to index components. Secondly, a technique called adaptive indexing is used. This collects information about the use of terms interactively, by adding terms to the representation of objects when they have been used in the context of a successful search.

7.9. Discussion

Where the CodeFinder approach succeeds over the faceted retrieval and keyword-based approaches is in its ability to evolve as new terms are identified. The weakness of systems that are dependant upon a keyword-based approach (to which faceted systems may be said to belong) is perhaps best illustrated in a paper by Furnas et al. (1987), which demonstrated that the probability of two people selecting the same term to describe a component is less than 20%. This is further noted by the comment of Curtis (1989) that an ideal component library would have an “...indexing scheme similar to the knowledge structures possessed by most programmers working in an application area.” The difficulty comes in identifying what these knowledge structures are. Whereas a particular retrieval system may appear to be based on a well-researched structure of information, the success of retrieval is dependant upon how well this can be used, which is in turn a function of the user as well as the system.

In essence then, a component retrieval system that rigidly classifies the artefacts held within it is essentially useless, unless either the classification terms are universally used, or the users are capable of bridging the conceptual distance to the mechanisms and concepts used by the retrieval system. While domain analysis is effective in operating at a higher
level of abstraction and thus at identifying higher level concepts for reuse, difficulties arise in defining the domain boundaries. As Neighbors (1984) notes, real-world domains are in actuality a network of interrelated domains.

7.10. Conclusions

Successful component reuse must adapt to user requirements, and this is highly context dependent. Novak (1995) notes that to be effective, reuse must minimise the associated human and computational costs. The former is the time required by the programmer to find the entity to be reused, to understand it, and to perform any modifications required. The computational cost is the cost of employing a reused component relative to that of a ‘hand-written’ version. While the approaches studied here address storage and retrieval issues, the human costs of understanding the component have been largely ignored by researchers.
"Luke, you will find that many of the truths we cling to depend greatly on our own point of view."
- Obi-Wan Kenobi (from the film *Star Wars*, 1977)

8. Hypothesis and work programme

8.1. Introduction

The study of the literature conducted during this research has examined both the human and technical factors that influence the problem-solving process. Within software engineering, factors external to the particular problem being solved, such as the need to communicate between participants in the process and the need to meet project milestones, have heavily influenced the methodologies and tools available to software engineers. These have some considerable merit; the success of numerous projects attests to this. However, while they are effective in addressing the organisational aspects of software engineering, they do not provide sufficient support for the human aspects of problem solving. Attempts to improve software development practices must be based on a more complete view of the problem solving process.

The goal of this chapter is to identify an effective strategy by which the process of problem solving may be improved. The chapter initially identifies a number of fundamental principles that have been established from the literature review. These are then discussed, and used as the basis for an overall solution strategy.

8.2. Fundamental principles

As discussed in Chapter 2, research into human problem solving has demonstrated that it takes place through an iterative process at a microscopic level, with rapid shifts between the problem and the solution. However, the current view of the software engineering process has led to tools that are designed to support a development approach that takes
place through a series of iterative progressions from problem to solution, with these iterations occurring at a macroscopic level. This process accounts for the broader organisational factors discussed in the introduction. While these views are in many ways divergent, both must be accounted for if the effectiveness of problem solving within software engineering is to be improved. A number of principles identified from the literature search will now be reviewed.

a) No single representation is appropriate for all problems

One of the main themes of End User Programming (EUP) research has been the use of different representations for programming constructs. For certain user populations, such approaches have been found to be successful, and one indicator of this success is the employment of visual representations into mainstream program development tools. However, the inclusion of such problem domain knowledge within a programming environment can be problematic. As Neighbors (1984) noted, identifying a discrete body of knowledge is difficult, and domains overlap significantly. This makes the use of domain-specific approaches limited in their applicability, in terms of both the human factor and the support infrastructure.

Chapter 2 established that there is no single successful method of human problem solving. Amarel (1968) identified that minor changes in the representation of problems could produce significant changes in the efficiency with which solutions were found. Good (1999) supports this in the context of software development, by showing that there is no single technique by which all computer programs may be understood.

The concept of flexibility in representation does not only apply to visual programming constructs. Where components are stored in a component library, adopting a single descriptive format can restrict their accessibility. This is largely because of the factors that affect the description of components in software engineering, which are discussed in more detail in the next point.
b) Effective component reuse must encompass multiple perspectives

Creating and reusing components may be one means of achieving more effective system development, where the term ‘effective’ encompasses concepts such as speed, reliability, and efficiency. Chapter 3 established that components in software engineering differ significantly from components in the physical engineering disciplines, because they are processing information rather than interacting with physical forces. Thus, while component reuse is believed to hold great promise, the complexity of software components is often underestimated due to their perceived similarity to physical components.

As Curtis (1989) has suggested, an ideal component library would have:

"... [an] indexing scheme similar to the knowledge structures possessed by most programmers working in an application area."

Knowledge based approaches might offer one way of achieving this. However, such approaches appear to be of limited use. Chapter 2 identified that the creation of software is involved with managing two world-models: the real world problem addressed by the program (the situation model), and the steps that must be taken to solve the problem (the system model). A component reuse strategy must account for both of these models. The problems of doing this are highlighted by Domain Analysis (DA) approaches, which relate the situation model to the available components. While DA can be successful within certain well-defined problem domains, its focus on the situation model can lead to reuse opportunities being missed. For example, while two systems may be different at a high level of abstraction, at a low level, they may have common functionality. A retrieval approach that is only concerned with the situation model will miss such opportunities for reuse.

Another component retrieval tool may focus on only the functional aspects of the system (such as faceted retrieval), but here again, the broader situation model aspects of the problem may be lost. Component retrieval tools must therefore be capable of employing
multiple perspectives in the way that they store component information. As Jaensch (1930) notes:

“The danger of one-sidedness, subjectivity and error in the fundamental questions of knowledge, is chiefly due to the fact that every structure of consciousness claims unlimited validity; but in truth each makes very wide negative abstractions of reality. We can, therefore, only penetrate reality and approach the ideal of ‘pure experience’ by successively taking up the standpoint of different mental structures.”

While knowledge bases can be a useful source of information, difficulties may arise when the information therein is fixed, due to the high costs of adding new information or managing the existing information. The result of this is that the knowledge base then represents only a single perspective. It is proposed that this is very much the case when detailed structuring mechanisms such as conceptual graphs (Sowa, 1984) are used. Sowa himself notes that:

“A closed, rigid system maintains a sense of security by giving instant answers to all perplexities. But it is a false security that is threatened by any incompatible viewpoint.”

As Guindon’s (1990) study demonstrated, in solving a problem, software engineers (like other human problem solvers) move between different aspects of a problem (at different levels of abstraction) and thus need to take different perspectives depending on the (current) area of concern. For these reasons, it is proposed that effective component reuse cannot be based on a single representation; nor can it utilise only a single perspective.

c) Effective component reuse systems must add only a minimal additional workload to be accessible

Software development is a complex task, in terms of the technical issues to be addressed, the communication that must take place between participants, and the processes that must
be followed. All of these elements must be accounted for; it is therefore proposed that even a highly effective system of component reuse may be ignored, if the effort required to utilise it is perceived by users to detract substantially from the rest of the task. This may be seen in the considerable investment that must be made in DA, which limits its accessibility to organisations which have both sufficient (financial and personnel) resources to invest in its initialisation, and which have a relatively constrained domain area. A reuse system based around a particular problem domain may well be effective, but requires a significant organisational involvement in terms of content and maintenance. This limits it to a very small number of organisations and domains, and ultimately limits its effectiveness.

d) Exploring possible solution spaces informs the capture and understanding of requirements

One theme has emerged repeatedly throughout the literature search. It is perhaps best described by Schon (1983), who identifies design as an activity where knowledge about the problem develops in parallel with knowledge about the solution. This has been supported by the empirical research of Guindon (1990), and Green (1989), both of whom identified the essentially opportunistic nature of the design process. This view is implicitly recognised in the iterative approach advocated by many software development methodologies, as each iteration enables learning about the problem domain and possible solutions to take place and be acted upon. Approaches such as rapid prototyping demonstrate an increasing acknowledgement of the value of microscopic iteration in problem solving.

8.3. Discussion

Augmenting the software development process through component reuse is widely perceived as an effective approach to improving software development, as discussed in Chapter 7. However, creating a technical and organisational support structure for component reuse is a complex task. One such technique (DA) requires a considerable investment of organisational resource; even Reuse Development Teams (proposed by
Poulin, 1999) require a minimum organisational size, and entail a degree of organisational change. Despite these difficulties, component reuse remains a topic of considerable interest and promise.

Any approach to software development requires a significant investment by the developer in learning the basic principles of the approach. It seems reasonable to assume that at least part of this learning investment may be more effectively made in an implementation language with a broad range of application areas, rather than one which is specialised to a particular problem domain (as is usually the case with EUP and DA systems). It further seems reasonable to suggest that a component reuse system which utilises an existing programming language will be incorporated into an existing development process, and should therefore be designed to conform to such existing processes and tools where practical.

To accomplish this, it should be considered that the development of software is usually accompanied by the production of a considerable amount of supporting (design) material, both formal and informal in nature. This material already provides a valuable source of contextual information that describes both the situation model and the system model, as well as providing a mapping from one to the other. The sheer volume of material generated during development is considerable, as the developer(s) must manage and refine ideas in a highly dynamic environment, adapting and balancing different and developing requirements. The need to manage this information has led to the evolution of many different types of notation and (more recently) to methodologies which assist with this process.

Taking UML as an example, the situation model is described by the analysis aspects of the development, such as the use cases, actors, flow of events and requirements. The classes and methods of the design describe the system model. Most significantly, the situation and system models may be linked by the use of sequence diagrams illustrating how the classes work together in order to deliver the functionality outlined in the use cases. UML design information provides a conceptual connection to the coded implementation of the
system (i.e. from the situation model to the system model). Thus, a ‘complete’ form of reuse, using both design and code information appears to be possible through such development information. The use of UML may also enable designs to be understood by a larger number of designers than if more domain-specific languages were used, enabling designs to be reused easily across multiple problem domains.

UML provides the facility to capture both general principles of a design, as well as problem-specific textual documentation. It is proposed that human-readable text can provide an effective descriptive mechanism that is suitable for a component retrieval system. This is supported by the use of text as the native data format within the UNIX operating system and the recent emergence of the eXtensible Markup Language (XML) as an application-independent mechanism for data description.

It has frequently been noted that design as an explicit process during development is not performed as often as might be desirable. It is suggested that utilising design material as the basis for a component reuse strategy may encourage the use of a design process in system development, as well as potentially making a component reuse strategy accessible to many more developers, regardless of the size or resources of the host organisation. This addresses one of the most substantial barriers to widespread component reuse.

8.4. Work programme

To facilitate component reuse by utilising design information, where a component can be either a design or code element, a number of factors must be addressed. In terms of the situation model, the solution strategy must account for how the (re)use of components is affected by the way in which they are described. It is also necessary to ensure that developers are capable of using existing design materials as a guide to modifying ongoing designs. Both of these aspects are most effectively addressed through empirical studies.

It is also important to address the system model aspects of the approach. It must be technically feasible to populate a data store with both design and code components, as well
as to have a means of effectively retrieving design and code components during the design process.

Finally, it must be possible for a component retrieval system to account for the mapping between the situation and system models. This should encompass both the technical factors of component retrieval as well as the broader environmental factors that have been identified as important in software development. These include the limited capability of developers to invest time and effort in learning additional retrieval skills, the extra effort required in retrieving components during the development process, and so on.

As discussed in Point 8.2d, the problem solving process is characterised by a growth in knowledge about the problem as solutions are developed. Therefore, an iterative development strategy was adopted which involved the development of a number of research systems, each of which addressed a part of the overall problem area as identified here. It was believed that this would reduce the risks inherent in a single (large) development, and so enable deeper insight into the problem through the creation of possible solutions.

8.5. Summary

A work programme has been established based upon the principles identified in the literature survey. This programme addresses the problem area in a holistic fashion, accounting for a number of significant factors in the problem domain. The approach proposed should therefore provide an effective basis for producing a satisfactory resolution to the problem.
"Everything should be made as simple as possible, but not simpler."
- Albert Einstein

9. A study of component reuse

9.1. Introduction
This chapter describes an experiment which explored the way that subjects from different backgrounds (in this case academic disciplines), approached the task of combining components to produce solutions to well-defined problems. The experiment was designed to provide subjects with descriptions for a number of components, to explain how these components could be combined, and to observe the subjects performing a series of simple tasks using the components. This type of experiment was required before any proposals could be made for component reuse systems, since it provides a more detailed understanding of how subjects may use existing components to address simple problems.

Previous research (identified in the literature review) highlighted several difficulties that needed to be avoided in the experiment:

- inability to use the retrieval system due to inadequate descriptions;
- lack of knowledge of the problem domain; and
- lack of knowledge of the solution domain (what components might be available and methods of combining them).

In response to these difficulties, the hypothesis of this chapter is that:

If:

A1: subjects were provided with a few simple components, each of which had a clearly described purpose;
A2: there was a single, straightforward mechanism for combining the components; and
A3: the problem domain was readily understood and appreciated by the subjects;

Then:

C1: subjects should be able to combine the components to provide solutions for specified
tasks; and
C2: subjects should be able to generate a relatively common and consistent set of solutions
to the problems.

A suitable component basis may be found in the UNIX shell scripting mechanism (A1). This employs components that can be used independently to perform simple tasks, or in combination to perform more complex tasks (A2). As such, shell scripting provides a moderately 'clean' implementation of the component reuse concept, as problems are solved using tools constructed through composition. However, the UNIX environment is often perceived as 'unfriendly', particularly by computing novices, and this would have introduced confounding factors into the study. To address this issue, components modelled on the UNIX approach were generated, which could be used without interacting with the UNIX environment or, indeed, a computer (A3). Subjects were provided with a number of cards, each of which represented a component, and which was associated with a short description. Subjects could then combine these component representations in order to produce solutions to simple problems.

The UNIX approach is an interesting implementation of component reuse in its own right, and for this reason is discussed in the next section. This provides the basis for the report on the study itself.

9.2. UNIX shell scripting

Programs within the UNIX shell environment present a simple interpretation of the component concept, as they are self-contained programs that are useful in their own right.
A 'software toolkit' approach is used, where each tool can be used independently, allowing users to acquire familiarity with tools in isolation. Each tool has an associated help file available describing what its purpose is, and how it is to be used. Furthermore, two or more such tools may be combined to perform more complex operations, as the output of one can be used as the input of another using the pipe ('|') mechanism. The pipe connects the output of one tool to the input of another and manages the synchronisation of the associated tools. This mechanism has revolutionised shell programming under UNIX, and many programs have been written using shell scripts rather than conventional programming languages (Kernighan, 1984). As Arthur (1990) notes:

“Each function, although trivial when viewed as a single entity, becomes vastly more important when combined with other singular functions to do virtually any kind of activity.”

Although the performance of shell scripts may make them unsuitable for very complex tasks, or tasks requiring great efficiency (Arthur, ibid.), the conceptual simplicity of the scripting process makes them appropriate for use and study on a small scale.

### 9.3. Experimental study: small-scale component reuse

An observational study was performed to examine how subjects from different disciplines could reuse simple components to perform set tasks. The goal of the study was to identify human problem-solving issues relevant to the description of components and the impact that this had on the overall process of component reuse.

#### 9.3.1. Subjects

One of the goals of this research is to develop software development tools for a broad spectrum of users, which includes users who are not software development professionals. The implications of such a broad 'user population' are that a wide range of problem-solving approaches will be used, which in turn will influence the design of support mechanisms. Paid subjects were used, who responded to advertisements placed
around the University. Six subjects were selected, in order to provide a broad sample of problem solving approaches. This follows the approach used by Lawson (1979) in a study of problem solving in the field of architectural design. This study demonstrated that the problem solving strategy adopted by the subjects was affected by the discipline to which the subjects belonged, as well as the length of time that had been spent studying within the discipline.

Although it could be argued that the study was biased towards those with a university-level education, the approach was nonetheless more rounded in terms of the problem solving strategies used than if only subjects from one or two disciplines participated.

The subjects were:

- A final (third) year computing Ph.D. student researching multimodal interaction
- A second year undergraduate Physics student
- A second year undergraduate English student
- A second year undergraduate Human Biology student
- A second year undergraduate Computing student
- A final (third) year undergraduate Maths and Physics student

None of these subjects were familiar with the UNIX operating system.

9.3.2. Method

Subjects were presented with a number of cards, each of which was labelled with the name of a component for processing information. Each component had a UNIX shell tool equivalent, but the name on the card was changed to a succinct English equivalent that had been determined by the experimenter, in consultation with a UNIX systems administrator, to adequately describe its functionality for non-computing professionals. The exceptions to these were the pipe ('|') and the redirection commands ('>' and '>>'). These were left
in their original form for two reasons. Firstly, their English equivalents were felt to be both ambiguous and verbose, particularly for the pipe tool. Secondly, it was felt that leaving some of the tools in a non-textual form would be a useful means of determining the acceptability of symbolic representations for a component reuse system. The UNIX tools and their chosen natural language equivalents are given in Table 9.1:

<table>
<thead>
<tr>
<th>UNIX tool</th>
<th>Experimental equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>Display</td>
</tr>
<tr>
<td>grep</td>
<td>Find</td>
</tr>
<tr>
<td>sed</td>
<td>Replace</td>
</tr>
<tr>
<td>ls</td>
<td>List files</td>
</tr>
<tr>
<td>cp</td>
<td>Copy</td>
</tr>
<tr>
<td>rm</td>
<td>Delete</td>
</tr>
<tr>
<td>mail</td>
<td>mail</td>
</tr>
<tr>
<td>mv</td>
<td>Rename</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>&gt;&gt;</td>
</tr>
<tr>
<td>w</td>
<td>Show users</td>
</tr>
<tr>
<td>*</td>
<td>All files</td>
</tr>
<tr>
<td>l</td>
<td></td>
</tr>
</tbody>
</table>

*Table 9.1: Study tools*

It should be noted that although all of the subjects were computer literate, they were not expected to be familiar with the concept of sequencing, and so the sequencing operator was not included.

Subjects were also provided with a sheet that provided a brief (one-sentence) description of each component. The components and their descriptions are presented as Table 9.2:
<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display <code>&lt;filename&gt;</code></td>
<td>Displays the contents of <code>&lt;filename&gt;</code></td>
</tr>
<tr>
<td>Find <code>&lt;pattern&gt;</code></td>
<td>Displays the line containing <code>&lt;pattern&gt;</code> in the input</td>
</tr>
<tr>
<td>Replace <code>&lt;existing pattern&gt; &lt;new pattern&gt;</code></td>
<td>Replaces <code>&lt;existing pattern&gt;</code> with <code>&lt;new pattern&gt;</code></td>
</tr>
<tr>
<td>Mail</td>
<td>Emails <code>&lt;user&gt;</code></td>
</tr>
<tr>
<td>List files</td>
<td>Lists all files</td>
</tr>
<tr>
<td>Copy <code>&lt;existing filename&gt; &lt;new filename&gt;</code></td>
<td>Makes a copy of <code>&lt;existing filename&gt;</code>, and calls it <code>&lt;new filename&gt;</code></td>
</tr>
<tr>
<td>Delete <code>&lt;filename&gt;</code></td>
<td>Deletes <code>&lt;filename&gt;</code></td>
</tr>
<tr>
<td>Rename <code>&lt;existing filename&gt; &lt;new filename&gt;</code></td>
<td>Renames the <code>&lt;existing filename&gt;</code> with <code>&lt;new filename&gt;</code></td>
</tr>
<tr>
<td>Show users</td>
<td>Displays other users currently logged in</td>
</tr>
<tr>
<td><code>&gt;</code> <code>&lt;filename&gt;</code></td>
<td>Takes the output and puts it into <code>&lt;filename&gt;</code></td>
</tr>
<tr>
<td><code>&gt;&gt;</code> <code>&lt;existing filename&gt;</code></td>
<td>Adds the output to an existing filename</td>
</tr>
<tr>
<td><code>*</code></td>
<td>This symbol indicates all files</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
</tbody>
</table>

Table 9.2: Tool descriptions

Examples of use for each of these tools was available. Subjects were presented with an instruction sheet that outlined the task they were to perform. These instructions were:

“You have been presented with a computer that contains a number of files, in a single directory. You have available a number of tools which can be used to perform tasks on these files. Each tool is represented by a card containing the name of that tool. By rearranging these cards and combining
them with the pipe (`|`), you must indicate how you would use the tools available to perform these tasks.”

The subjects were asked to imagine that they were working on a computer in order to put the task in context. While it would have been technically possible to perform the experiment on a computer, it was not considered desirable, as this would introduce confounding factors for those subjects unfamiliar with computers.

For example, to check if there were any files called “test.doc”, the following script could be used:

```
List files | Find test.doc
```

Subjects were asked to describe their thought processes as they performed each task, in order to provide insight into the progression of the problem solving process.

Each subject took part in two consecutive sessions, which were separated by a five-minute break. Each session comprised the same series of tasks. However, the order of presentation was randomised between sessions in order to remove some practice effects. In the first session, subjects were asked to generate a solution as quickly as possible, although this was not enforced by the experimenter. In the second session, subjects were presented with the same tasks but given as much time as they needed to produce a solution, which they could choose to change from their initial answer. This approach was intended to capture an ‘intuitive’ response when little reflection on the materials was possible, and then to see how (and if) the solution changed when more time was available, and the subjects were able to consider the questions and tools in more detail.

### 9.3.3. Results

While the raw data from the study is provided as Appendix A, this section provides a presentation and analysis of the results. The experimental design had provided the
subjects with a small number of tools represented by labelled cards. It was hypothesised that these could be readily combined (using the available descriptions) in order to provide solutions for a set of fairly simple tasks (C1). However, the behaviour of the subjects during the study did not take advantage of this aspect of the experiment design. Contrary to expectations, only one of the six subjects manipulated the cards in creating their solutions. The cards were provided in order to enable subjects to spatially manipulate the tools, in much the same way as allowed by some EUP environments. It was anticipated that this would enable subjects to easily consider different combinations of tools, without putting undue load on their (human) memory. Rather, subjects preferred to use the information sheet that provided descriptions of the tools and write down the resulting scripts.

Although the students were not scripting in the UNIX environment itself, it was felt that identifying suitable solutions would be useful as a benchmark against which to compare the solutions generated by the subjects, in terms of the tools used as well as their structure. Accordingly, sample solutions, based on the tools and descriptions used in the experiment, were generated for each task in conjunction with a UNIX systems administrator. These solutions are presented as Table 9.3:

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mail the file &quot;Joe.txt&quot; to &quot;Fred&quot;</td>
<td>Display Joe.txt</td>
</tr>
<tr>
<td>Give the file called &quot;work.doc&quot; the name &quot;play.doc&quot;</td>
<td>Rename work.doc play.doc</td>
</tr>
<tr>
<td>Put the contents of all files into a file called &quot;New&quot;</td>
<td>Display * &gt; New</td>
</tr>
<tr>
<td>Make a duplicate of the file &quot;old_data.txt&quot;, calling it &quot;new_data.txt&quot;</td>
<td>Copy old_data.txt new_data.txt</td>
</tr>
<tr>
<td>Check if the file called &quot;nursery&quot; contains the text &quot;Mary had a little lamb&quot;</td>
<td>Display nursery</td>
</tr>
</tbody>
</table>

Table 9.3: Sample solutions
This comparison generated some useful results. As anticipated, the discipline to which the subjects belonged appeared to have an effect on the approach that was taken to problem solving. For example, the subjects from English and Human Biology, who did not use computers on a regular basis, expressed concerns that they might damage the ‘computer’. Even though they had been told only to imagine that they were working on a computer (to place the task in context), this still appeared to affect their proposed solutions. Based on the think aloud protocol obtained from these subjects, this was shown in a far more defensive style of scripting than that exhibited by the other subjects. For example, in addressing the task “Give the file ‘work.doc’ the name ‘play.doc’”, the third year Computing Ph.D. subject suggested:

Rename work.doc play.doc

The Human Biology subject, on the other hand, initially suggested:

List  
Display filename  
Rename work.doc play.doc

In the second session, the subject realised that this was doing ‘too much’, and removed the “Display filename” command. However, the “List” command was still present. This is described as “Lists all files” by the component information sheet, and therefore does not appear to be an integral part of the task at hand. This subject used the list command for all but one of the other tasks, apparently to verify that the files to be used were available (based on the think aloud protocol). The range of interpretations for what initially appeared to be a straightforward set of tools and tasks was considerable, contrary to the second hypothesised consequence. For instance, the task “Check if the file called ‘nursery’ contains the text ‘Mary had a little lamb’” produced the following results (collated from all subjects):

List
These solutions show a number of different interpretations of the task and the stated purposes of the tools. For example, whereas some solutions utilise both the 'Display' and the 'Find' tools to address the problem (as proposed in the sample solutions in Table 9.3), the first and third solutions utilise either one or the other. It is also interesting to note that the pipe tool was used by only two subjects. One was the student studying Maths and Physics, the other was the Computing Ph.D. student. From the talk-aloud protocols, it appeared that three of the subjects confused the purpose of the pipe ('|') with the redirection tool ('>').

Overall, the different solutions proposed by the subjects varied significantly. Some would not have provided an effective solution to the proposed problem, and even the ones that would have worked varied significantly in terms of the efficiency with which they would address the task in a real-world situation. For example, the first and second approaches listed above differ in that the first approach would require the user to visually examine a (potentially very large) file for the desired string, whereas the second would avoid this additional effort. While efficiency was not an explicit concern of the tasks given to the
subjects, the range of solutions provided for all the tasks is considerable, particularly when the simplicity of the original tool descriptions is considered.

9.4. Discussion

A considerable amount of effort had been expended in planning the experiment to create a context that removed much of the real-world complexity of the problem domain. In doing so however, the feedback from the real-world context was also removed, and this appears to have been more significant than was originally anticipated. Many of the solutions proposed by the subjects would have been inappropriate, either because they would not work at a technical level, or because they potentially increased the amount of work required of the user. However, it was also felt that to implement a feedback process in the experiment (which could have addressed this) would potentially encourage a hacking approach; as one (Computing) subject remarked in the second session, “My way would be to fiddle with it until I got it right”. This is in sharp contrast to the subjects from Biology and English, both of whom were unwilling to ‘fiddle’ with the tools due to the perceived risk of damaging the ‘computer’. The English subject said that she “wanted to avoid overloading the computer”, while the Biology subject tended to use the same tools repeatedly, to avoid “getting in a muddle”. The approach taken by the Biology student was generally more defensive than that of the other subjects; for example, she would verify that the requisite files and users (to be emailed on one of the tasks) were available before trying to use them. As with the English student, this may have been due to a lack of experience with computers (particularly in a programming context).

While the subjects generally found the correct tools to use in producing solutions, the order in which the tools should be used was noted as a major issue, particularly where there was a need to use the symbolic tools to link the (textual) tools together. It was interesting to note that only the subject from Mathematics and Physics and the postgraduate subject from Computing made use of the pipe tool. The undergraduate Computing student noted that he kept forgetting the meaning of the pipe, and had to keep reminding himself. The general lack of use of this tool by other subjects may therefore
have resulted from its lack of a descriptive name, or a lack of familiarity. This latter explanation appears more likely because the redirection (‘>’) tool was used by most of the subjects, and while this is also not in widespread use, the symbol is commonly used in mathematics (albeit with a different meaning), a discipline with which all but one of the subjects had recent experience.

Despite the difficulties encountered, the experiment provided evidence that the respective academic disciplines of the subjects influenced the problem solving strategies adopted. One of the subjects from Computing and the subject from Mathematics approached the problem solving situation in an explicitly top-down manner, initially identifying the tools that seemed appropriate, and then combining them using the symbolic tools. The Physics student adopted an approach of silently contemplating each task and the available tools (in one case for two minutes) before writing down a fully formed answer. This subject was also notable for modifying only two of the five tasks during the second session; consequently their initial strategy appears to have been perceived as a successful one.

Another factor that was considered to be significant was the use of the descriptive sheet over the cards. Cards were provided on the understanding that subjects would find them a useful memory aid in the process of forming solutions. On the contrary, all but one subject appeared comfortable with attempting different solutions without such external aids. This did not appear to be a significant problem for tasks of this scale, but for larger tasks would undoubtedly be an issue as a result of the cognitive limitations outlined in Chapter 2.

In an informal discussion with subjects after the study, it emerged that there was a lack of understanding of the problems and available tools, as well as the filter / pipes approach to solving the problems. One solution that emerged from these discussions was that of using a range of descriptions for the problem and the tools. Although it would not be possible to describe all the problems that might exist, it would seem feasible to be able to provide sufficient descriptions to increase the probability that subjects would respond to at least one of them. A fundamental basis for component reuse is that the same tasks are
performed repeatedly. Given this, it is reasonable to suggest that the number of common uses for a component (and hence the associated descriptions) is relatively small. For example, finding a particular string in a file is a common task that is implemented in many applications, from low level operating systems (e.g. the `grep` tool usually available as part of the standard UNIX toolset) to high level applications such as Web browsers and word processors. This command is most often called 'Search' (in the TextPad text editor) or 'Find' (in most Microsoft products). However, a user might also consider the terms 'locate', 'explore', or 'detect' when contemplating the same functionality. By capturing descriptions (and hence some measure of context) from a relatively small number of users of (and uses for) a component, a useful component retrieval mechanism employing these descriptions for future use could be created. This suggests that an evolutionary approach might be taken, where, each time a component is used successfully, a description of how it is used is captured. Relatively quickly, a useful library of descriptions may then be created, which could be searched and an appropriate component retrieved.

9.5. Conclusions

This experiment attempted to increase the accessibility of a number of component-based tools to problem solvers, by reducing the context associated with the tools and the problem domain. However, applying a moderately wide range of problem-solving strategies within this domain has demonstrated issues with both removing context from problems, and with removing the interaction between the problem domain and the solution domain. Furthermore, it has suggested that the degree of technical knowledge (system model knowledge) required by the user of a component-based system is of greater importance than has hitherto been acknowledged, even for systems that are perceived as being relatively simple. This was particularly apparent in the subject errors that could reasonably be attributed to the lack of feedback in the system. The usefulness of feedback would however be dependent upon a greater degree of system domain knowledge than was available to the experimental subjects. As a result of this experiment, an alternative approach that addresses these issues may be considered. This would build relationships between the problem and solution domains using a positive feedback approach, capturing,
and allowing others to build upon, successful problem solving strategies that used the available components by multiple users, in a context-rich environment. It has also suggested that it may be necessary to construct a situation in which the reuse of components was positively favoured, in order to obtain multiple component descriptions from users. Once a more detailed technical understanding of the problem has been obtained, a more robust foundation from which to address the problems experienced by end users will be available.

The possibility that multiple descriptions could be captured from users by getting them to use existing components relates to the results of Henninger (1996), described in Chapter 7. The next chapter describes a second prototype that was developed and implemented to investigate the practicability of these ideas.
"As we mentioned, the old name for the del code was "Rubout", and you may see this name when you examine the settings for your communication program. All that it means is that the program was designed by an old person."

- Harley Hahn

10. UNIX Script Retrieval Mechanism

10.1. Introduction

The experiment described in the previous chapter suggested that it would be desirable to have multiple descriptions for each component and also that such descriptions might be acquired by component reusers adding descriptions at the point of reuse. It also suggested that it would be desirable to give users feedback on the result of executing the component combinations that they chose.

This chapter describes the design, development and heuristic evaluation of a system that would enable these suggestions to be explored further. The objective was to produce a system which allowed components and descriptions to be added to a component store. The system would allow components to be retrieved from the store by matching descriptions to requests; and the results of using retrieved components could be shown to the user, so that they could gain an improved understanding of the nature of each component.

UNIX shell scripts were again chosen as components, for the reasons discussed in Chapter 9. On this occasion, however, the system was explicitly computer-based in order to allow feedback on the use to which the components were put. In order to provide an easily accessible environment for users of the system it was decided that it should have a Web-based interface. It was felt that this should also help reduce (through familiarity)
possible fears of 'damaging' the computer reported by some of the subjects in the previous experiment.

The problem domain being addressed here is 'wicked' (initially described in Chapter 2). It is moving the component retrieval problem from a 'one to one' mapping, with one component having a single description (as used in the majority of component retrieval systems) to a one to many mapping, with one component having potentially many descriptions. These descriptions would be obtained from the multiple areas to which the component could be applied, given the vastly increased size of the problem domain and the increased context base. In practice however, many to many relationships will exist, as a single description may also apply to many components, as different ways of performing the same task will be identified. Additional complexity will result from the problem solver moving between the problem and solution domains. This transition should ultimately become a source of increased descriptive power over the components, as additional uses are identified. The final judge of the suitability of a particular component for a task will always be the user; however the descriptive information available should assist the user in selecting a component.

The remainder of this chapter describes this research system, called the UNIX Script Retrieval Mechanism (USRM). An examination of the system design and implementation is followed by a case study of the tool in use. The evaluation of the system is then discussed, identifying the merits and weaknesses of the approach.

10.2. System design and implementation

In order to investigate problems of component description and retrieval it is necessary to use some actual components. Suitable components would be ones that are readily available and offer useful functionality. Accordingly, UNIX shell scripting components were used in this experiment. The system was designed to retrieve these components through natural language descriptions. This was done following the work of Mili et al. (1997), who found free-text retrieval to be an effective approach. Another advantage of
this approach was that it should increase the accessibility of components to users from a
range of disciplines. The system was designed to enable many descriptions to be
associated with a given shell script. This was central to the design since it would increase
the likelihood that the terms associated with (at least) one of the descriptions of the script
would be similar to those used by the retriever. The system was also designed to enable
scripts to be run, in order to provide feedback on the results of a script, as a valuable
source of information for the user about how the different components operate.

It was apparent that some form of database would be needed for managing the information
stored by the tool. As a research system, it was desirable that this database be flexible in
the way that data is stored in terms of the data types and structures used, since these were
likely to change as the system developed. Therefore, the pgm indexing system was
chosen. This is an in-house tool, written by the MRRL (Midlands Regional Research
Laboratory)/AIMS (Advanced Information Management Systems) research group in the
Department of Computer Science at Loughborough University. Some of the details of
pgm, which forms the basis for the GENIE (Global Environment Network for Information
Exchange) System are covered in Smith et al. (1997). Pgm is designed to provide
indexing facilities for very large data sets, but is also notable in that it enables the
developer to easily change the structure of the data during or after development. Thus, it
is an extremely powerful tool that enables ideas to be explored with far greater flexibility
than was possible with the other database systems that were readily available. Access to
pgm from within another program was possible using a number of software libraries that
had been written using C. These libraries were used in order to rapidly implement the
system using pgm, and to allow possible data structures for a component library to be
explored. The system was implemented using C++, as the author was familiar with the
language and it made interaction with the pgm libraries easier. In addition, as discussed in
Chapter 4, coding in C++ can be more effective than coding in C.

As with the database, it was desirable that the system interface be easily modified as the
underlying functionality changed. The interface was therefore written using HTML, to
provide a Web interface to the system. As UNIX is a networked operating system, it was
appropriate that the interface for a tool designed to support scripting was easily accessible over the network. In addition, the ubiquity of the Web made it highly probable that users would be familiar with using this style of interface.

10.2.1. Implementation

The system consists of three main elements: the pgm data store, a Web interface, and the C++ code that provided the functionality. The system offers three main functions:

- Storage of scripts and their associated descriptions;
- Script retrieval based on entered descriptions; and
- Test execution of scripts.

This section will describe how the system can be used for each of these functions.

10.2.2. Script creation

Figure 10.1 shows the USRM interface used for creating new scripts:
Figure 10.1: USRM script creation interface. Users can enter in scripts that will be associated with the descriptions they enter,

It was felt that clarity and simplicity in the interface design would enable the attention of users to be focussed on the functionality of the system, as well as enabling development changes to be made easily.

This interface enabled the user to enter their personal identifier (usually either a name or login code), together with a description of the script and the script itself. Within pgm, each script is associated with a name, date (obtained from the system clock) and the descriptions(s). This approach provides the components with additional metadata so that it can be managed more effectively. While it is not necessary to capture the users name, open source projects usually enable developers to associate their names with their work as a reward. Similarly, the seti@home project encourages friendly competition between
organisations and individuals by displaying their names and the number of units processed. These approaches do not reward contributors financially, but rather with recognition for their contribution to the wider community. Similarly, USRM could acknowledge the contribution made by authors to their community.

Once the user has submitted the script, it is executed on the host computer. The results of the script are then displayed to the user, as shown in Figure 10.2:

![Script Results](image)

**Figure 10.2:** Script results produced by USRM. This allows users to confirm that the description accurately describes what the script does, and that the script is generating a reasonable set of results.

In the database, the script is checked against the contents of the data store and if no syntactically identical script is found, it is added. Where new scripts already exist within the system, the description is checked, and if this is new, then the new description is associated with the existing copy of the script within the database. This approach results
in multiple descriptions being associated with the same script, allowing a more complete record of the capability of that script to be established.

The other major functional element of the system is script searching, which is discussed in the next section.

10.2.3. **Script searching**

Figure 10.3 shows the interface used to search for scripts that have been stored in the system:

![Script Searching Interface](image)

**Figure 10.3:** USRM Script searching interface. This enables users to search for scripts based on author, words used in the description, or date.
In order to provide flexibility in the approach taken to retrieval, information about the author, description, date, or combinations thereof can be used. This enables individual users to effectively retrieve scripts based on a range of descriptive data. USRM could thus be used as a personal store of components.

Within the data store, the search data is compared against the terms known to the system. The matching routine itself is straightforward as it uses the in-built facilities of pgm, which enable an input phrase to be split into its constituent words, and to perform matching on these words (in a case insensitive manner) or on groups of words. Where multiple search criteria are entered, the system attempts to perform a logical AND with the search results, and if no results are retrieved, then an individual search is performed on each search term.

This generates results similar to those shown in Figure 10.4:

![Search Results](image)

**Figure 10.4:** Search results returned by USRM for the search data entered in Figure 10.3.
At this point, the user can select a suitable script from the search results, and retrieve it from the USRM. When the script is returned, the user can of course execute it, but is also presented with the opportunity to add an additional description to the script, based on the (potentially) new context within which the script will be used. At the server side, the number of retrievals for each script is noted for administrative purposes.

10.3. Evaluation

The development of USRM was designed primarily to demonstrate the feasibility of evolving descriptions. This has been accomplished successfully. USRM was intended to be a research tool; however, it was nonetheless appropriate to perform an evaluation on the system as implemented in order to identify potential areas for improvement. With this in mind, a heuristic evaluation was performed. This is a form of evaluation designed primarily to identify usability problems (Nielsen, 1992). The main advantage of heuristic evaluation is that (particularly if evaluators skilled in the problem domain are used), it enables a majority of problems to be identified at low cost, and was therefore appropriate for use in the environment within which USRM was developed. The goal of the evaluation was to identify problems with the basic principles underlying the (research) system, rather than to evaluate it using the same criteria as would be applied to a production system. Accordingly, three members of the departmental support staff, all of whom used UNIX shell scripting in their work and had been involved in the evaluation of information systems over a number of years, performed the evaluation. These were specialised users with a high level of subject knowledge, thus fulfilling the 'problem domain expert' role identified by Nielsen (ibid.). Their comments on the system design are reported below, together with more specific comments on areas where the system is felt to be lacking.

General comments

On the whole, the evaluators felt that the basic concept of capturing multiple potential solutions to a given problem was a valid one. Furthermore, they endorsed the use of a Web interface as it made the system accessible from any network, something which would
be important if it were deployed as a systems support tool. This accessibility was also important in terms of giving users feedback on the operation of the script. The evaluators also agreed that providing users with a Web interface as a means of accessing scripts in a safe execution environment would provide the illusion of a familiar context which should help overcome potential user fears of ‘damaging the computer’ reported in Chapter 9.

While adding scripts was noted as an additional workload, the evaluators commented that developers already freely contribute to publicly available software libraries. However, it was also suggested that it might be useful to initialise the USRM with scripts and associated descriptions to provide the ‘critical mass’ that would be necessary to attract the initial interest in the system, possibly by retrieving scripts from system logs. Simple descriptions could then be associated with individual components.

No means of rating the scripts
Although information was collected regarding the number of times a particular script had been retrieved and submitted, it was suggested that some means of scoring the scripts would have been useful, particularly for complex scripts where one solution might be perceived by an expert as a more effective approach.

Insufficient descriptive information
Although a reasonable amount of meta-information was provided with the scripts, the evaluators suggested that (particularly for more experienced users) a hyperlink to the UNIX manual (man) pages would be useful, as it would stimulate directed learning by users.

User must explicitly enter the descriptions
While the motivation to provide descriptive information was understood, it was felt that this was not an ideal approach, as it interrupted the normal workflow. However, it was difficult to identify how this might be improved.
No indication of how the script performed the task
Although the tool was a useful retrieval mechanism, it was unable to provide an explanation for how a script performed a task. As with the redundant tasks performed by some of the subjects in Chapter 9, this was perceived as a long-term weakness of the tool. A possible means of addressing this problem would be the inclusion of a link to the UNIX manual pages suggested earlier (provided that suitable pages are available).

10.4. Conclusions
The comments of the evaluators have supported the hypothesis: the approach of capturing multiple component descriptions as a means of increasing the reusability of the components is likely to be successful, particularly when the system incorporates a degree of feedback. While shell scripting has provided a useful base from which to study the problems of component comprehension and retrieval, it has a very small user base, and technical issues limit the scale and complexity of the problems to which it can be applied. In addition, each shell script and its associated descriptive information exists at a single level of abstraction. As discussed in previous chapters, this can be problematic, as many levels of abstraction must be addressed in the development of most systems. From the comments of the evaluators, whilst there is a desire to have a large amount of descriptive information available, the provision of this information is perceived to be a time-consuming process. Where scripts of moderate complexity are added, the provision of this effort may be considered to be too much. The concerns expressed by the evaluators suggest that an alternative approach will be needed to obtain descriptive information for components, that does not require so much effort on the part of the users. One approach to this may be to integrate component capture and component discovery more closely into the design and development process, an approach that is explored more thoroughly in Chapter 13, where the final research system is described.

While the systems addressed in the experimental work so far have been fairly accessible to non-specialist users, it has become increasingly apparent that the question of the problem—solution mapping needs to be addressed at a deeper level than that of taking
simple prewritten components and combining them in scripts. To develop systems, there must be a balance of capabilities between the system developer (end user or specialist) and the computer support provided for this. The research described so far has indicated that addressing the problem primarily from the perspective of end users necessitates dealing with an extremely broad range of (user) backgrounds with little in the way of common knowledge or experience which can be built upon. This does not invalidate the value of an understanding of human problem solving in the process of software development; the studies performed so far have shown that this is of fundamental importance in an appreciation of the complete human—computer software development system. However, it does suggest that a more fruitful area of research may lie in the study of support systems for more 'conventional' software development languages and tools, as these are where the greatest gains (across the whole field of software engineering) may be made, by building on existing skills and tools.

This therefore requires two major problems to be addressed. The first is the population of a component store with more conventional components for reuse than have been described in this chapter. The second, and more significant problem, is a resolution of the problem of mapping between the situation model and the system model for software engineers. The next chapter will address the first of these problems.
11. CORE: the COmponent Retrieval Engine

11.1. Introduction

The evaluators of the USRM had suggested that providing component descriptions was an additional task that broke up the normal workflow. They had also suggested that it would be desirable to create and populate a component store 'automatically' before asking users to add to it, in order to provide such a system with a degree of initial 'momentum'. This chapter describes the design and development of a tool that will create a database of components from existing code, in order to investigate the practicability of these ideas.

The two experiments described in Chapters 9 and 10 focused on the use of UNIX shell scripts as components, since these were felt to be simple and to require no knowledge of programming in order to understand the ideas of component retrieval and composition. However, in practical terms, most systems are currently built by specialists who are familiar with programming concepts. Furthermore, most sets of existing components that could be used to populate a component reuse system will have been developed by such specialists and be intended for their use: the domain knowledge of the people who developed the components is likely to be very close to that of the people who are reusing the components. Thus, the system described in this chapter was designed to utilise components written in the Java language. Java was chosen because, as discussed in Chapter 5, it is a modern language which addresses many of the needs of developers; it is (largely) machine independent; there are many components readily available from the standard API; and the JavaDoc tool provides developers with the ability to easily create documentation.

In this study, the primary purpose was to confirm the practicability of generating component descriptions from an existing set of components, as a step towards producing a more comprehensive component retrieval system, which would be integrated into the
design process. It was therefore decided to use existing components in the development wherever possible and to concentrate on functionality rather than interface issues.

### 11.2. Requirements

CORE was designed to enable the storage and retrieval of Java components. Rather than design the system to store components from the Java component architecture (JavaBeans, outlined in Section 7.7.1), the decision was made to store 'ordinary' Java classes and elements of Java classes from the API, treating these as components according to the definition given in Section 7.3. It was decided to use the same language for both developing the system and for the components stored therein.

As a research tool, it was desirable to develop CORE relatively quickly, rather than spending significant amounts of time adding additional functionality or sophisticated user interfaces. This implied that as much use as possible should be made of existing tools. In line with this approach, the system was to be hosted on a Sun workstation running a proprietary version of the UNIX operating system (Solaris 6), and using pgm for data storage. Functions for interacting with pgm from within C code were already available, and these were modified to provide a series of native bridge classes to Java code. As development progressed, these classes were inherited from, in order to add additional functionality without duplicating existing code.

As described earlier, two main functional elements were required to support reuse: component storage, and component retrieval. Requirements pertinent to each of these will now be discussed.

#### 11.2.1. Component storage

CORE was intended to provide a mechanism for the storage and retrieval of components, and a significant factor in the (perceived) value of a component library is the number of components therein. This is strongly influenced by the ease with which components may be added. Consequently, a major requirement for CORE was that it should make the
addition of components straightforward. Unlike the USRM, wider descriptive information (such as documentation) was not collected about the components. Such situation model information can already be collected using the JavaDoc tool, and hence to supply this functionality was felt to be a duplication of a proven system. Instead, the code itself was the main focus of interest, as (in the longer term) it was expected that contextual information would be collected via the design information. Nevertheless, it was felt to be important to provide as much information as could be extracted from the code to facilitate reuse at the system model level. Accordingly, the following information was to be extracted from each class added to the system:

- Package name;
- Class name;
- Class type (applet, application, etc.);
- Class header;
- Class code;
- Classes imported;
- Classes extended;
- Method name;
- Method header;
- Method visibility; and
- Method code.

According to the definition provided in Section 7.3, these provide a reasonable subset of what a component may be considered to be.

11.2.2. Component retrieval

Although the ease with which components could be added was expected to be one important determinant for an effective system, it would clearly be insufficient unless it was also possible to retrieve components easily. This would be determined by:
• the physical accessibility of the system;
• the type of information stored within the system; and
• the ease of use of the interface.

A further design criterion for the system was that it should be possible to easily modify the interface in response to changing functionality or feedback from evaluations.

### 11.3. Design

The basic system architecture was a client-server model, using a UNIX Web server and an MS Windows PC client. To a large extent this was dictated by the pgm data store used, which was housed on the UNIX server. The GENIE system (Smith, Newman and Parks, 1997) had demonstrated that several pgm-based systems could interoperate to provide multiple servers that nevertheless all provide the same data view. For the purposes of this research, it was only necessary to produce a single data store; however, the use of pgm provided the potential for significant future expansion. Thus if desired, the basic data store underlying the system was sufficiently flexible to allow the creation of a distributed information management system. While for a research system this was not a significant design goal, it was nonetheless important for any future development.

To provide ease of access and modification, a Web user interface was used for retrieval. The server-side functionality was supplied by Java servlets (discussed in Section 5.3.1). Servlets require a servlet engine within which to run, and as the host platform (a Sun workstation) was already equipped with the Apache Web server software (version 1.3.6), the jserv Java servlet engine (version 1.0b5) was obtained and installed.

It was important that the mechanisms to perform retrieval using a Web page were straightforward. It was however felt that adding components via the same medium would be excessively complex, particularly for large numbers of files. Therefore, component addition was performed using a Java program running on the server side only (via a command-line interface).
Parsing the code was the most complex part of the functionality. While the Java API has a set of classes (the Reflection API) which enable the extraction of class information, these were not effective for the purposes of CORE. This was because while the Reflection API enables class and method signatures to be obtained, it can do this only for compiled (i.e. machine-readable) code. For a system such as CORE, this is not sufficient, since the human-readable code needs to be captured. Therefore, a set of classes to extract the required elements within the code was created (it was later realised that there were alternative ways of doing this without writing new code). While this approach did not parse the code with total success, it nonetheless enabled the data store to be initialised with a significant number of Java classes, with approximately 80% accuracy, a score obtained using classes from the Java API as an evaluation.

11.4. Sample interaction

The main functionality of CORE is supplied by the search interface shown in Figure 11.1:
Searching within CORE is a straightforward process. The user enters the desired term in a text entry box, and then identifies that term as belonging to a particular type of stored item. The user can also specify the type of information to be returned, as either class details or method details. For example, a user can search for a class with the name ‘Dimension’, and choose to retrieve only the details of the methods it possesses.

If successful, this initial search will return a page of results (Figure 11.2):
The user may then review the results, and if a suitable component is identified, it can be selected and retrieved. At this point, the user can select one or more of the results and the full class or method details will be returned (Figure 11.3):
The data retrieval page displays the full code, enabling the user to further evaluate the code which may (for instance) then be incorporated into a system under development.

11.5. Discussion

CORE is a system that is capable of storing and retrieving Java components (in this experiment, classes). CORE demonstrates that Java code can be parsed to extract implementation information, as well as enabling this information to be stored and retrieved. One of the most challenging issues in the development of CORE was the production of routines to extract the raw code from the `.java` files. While these routines were largely effective, further development work is warranted. The underlying data store can easily be adapted to include additional forms of information, such as code from other
programming languages. Together with the adaptability of the interface, this is felt to be a significant design feature in enabling the system to be extended.

As mentioned earlier, CORE is not designed to provide class and method documentation. To generate this information would duplicate the functionality already offered by the JavaDoc tool. However, it would be feasible to augment CORE with JavaDoc functionality, providing a more comprehensive information source via a common (Web browser) interface. This would be a useful tool in its own right, particularly for teaching.

11.6. Conclusions
The design and implementation of this system has established a sound basis for the development of a generalised reuse tool, providing a facility for indexing and retrieving components that is suitable for use by Java developers. The next developmental step was therefore to use the experience gained in developing USRM and CORE as a basis for producing a more integrated reuse tool, DesignMatcher, described in Chapter 13, which addresses a greater part of the system development process. Before this could be done, an experiment was conducted into the effectiveness of the UML as a basis for understanding existing design materials. This is described in the next chapter.
"I use not only all the brains I have, but all I can borrow."

- Woodrow Wilson

12. UML Study

12.1. Introduction

In Chapter 8, it was suggested that design information could be used to identify opportunities for reuse during the design process, when the developer is establishing the relationships between the problem domain and the solution domain. In order to build a reuse tool to test this hypothesis, it is first necessary to identify a suitable design language or notation that can be readily understood by developers. While no studies could be found to support the many claims made for the Unified Modelling Language (UML), the degree of industrial and academic support it was receiving when this thesis was being written, provided a strong case for it to be used as the design notation in a reuse tool, to describe the problem and the solution. Before this could be done, it was felt that it was necessary to perform a study to ensure UML was appropriate for this purpose. Accordingly, a study was conducted to examine how developers used UML to understand an existing design. It was hypothesised that:

If:

A1: subjects were able to use UML in understanding the design of the system;

Then:

C1: they would be able to successfully modify that system.

This chapter describes how this study took place, and discusses the results and their implications for the use of UML as a basis for component reuse.
12.2. Background

As noted in Chapter 7, UML is rapidly becoming a major standard for OO development. UML has the potential advantage that it is claimed to be independent of a specific methodology. In principle, this should mean that UML designs could be created using a range of different design support tools (by different user populations), while maintaining a consistent notational standard. This would enable UML-based designs to be shared more widely than if multiple design notations were used, due to both the technical investment in support tools as well as the learning investment required for developers to understand different designs.

The goal of the study was to examine how developers used UML to understand existing designs. To present developers with a situation where understanding was required, the experimenter designed an example system using UML, and implemented this using Java code. The system was designed to read student course results from a file, and calculate whether the students had passed or failed a course, based on their grades. The purpose of the system was to provide a structure for understanding the supporting UML documentation, so to an extent, the nature of the system itself was irrelevant. However, by using concepts from a problem domain which would be familiar to academics, the likelihood that subjects would be able to focus on the UML documentation rather than on understanding the functionality of the system itself was increased.

It was felt that the implementation of a graphical interface for the test system would add substantially to the complexity of the design and implementation, presenting subjects with considerably more design information to understand than for the equivalent text-based interface. This would increase the amount of time taken to understand the system design, and hence require more knowledge and time commitment on the part of the subjects, without any significant gain in the goals of the study. Thus, to simplify the design of the test system, a command-line interface was used, which is shown in Figure 12.1:
As with the study described in Chapter 9, a problem domain was chosen with which the subjects would be familiar, in this case a student records system. This should enable the subjects to focus on understanding the design, since they should already be familiar with the domain.

The system was designed to manage student records. Each record consisted of the student type (undergraduate or postgraduate), the student name, and three course grades. The system enabled student records to be added, removed and modified, and course grades to be calculated based on these records. Records could be read from and written to a file, separated by a <CR> within the file, such that each record occupied a single line. Within the program, records were maintained within a linked list. This data structure was familiar to all the subjects and was functionally suitable to the task. The test system was implemented using eight classes (written in Java), which employed relatively basic programming constructs. Here again, the primary concern was with the ease with which the design materials could be understood.
12.3. Subjects

Three paid subjects were used. The subjects were:

- a computer science lecturer specialising in OO technology;
- a computer science research assistant; and
- a temporary lecturer in computer science with considerable industrial and academic experience.

Each subject had over five years experience of OO analysis and design, as well as at least one years experience with Java and UML. While the backgrounds of the subjects did not reflect the same degree of variability as the subjects used in the UNIX scripting experiment (described in Chapter 9), this was unavoidable due to the dearth of available subjects familiar with Java and UML at the time the experiment was conducted.

12.4. Method

Subjects were given an overview of the experimental task they were to perform. All subjects were told that the task was expected to take between one and two hours, and were given the opportunity to ask questions of the experimenter. The subjects were presented with a PC giving access to electronic copies of the design (using Rational Rose 98i) and code (using TextPad version 4.4.1, a Windows-based text editor with in-built macros to compile and run Java programs, obtainable from http://www.textpad.com). In addition to this, paper-based hard copies of the design and code materials were made available, and subjects were able to run the application itself.

Subjects were asked to spend some time examining the application and the design materials, running the application if desired (A1). They were tasked with modifying the design and implementation, in order to enable a user to restore a single record that had been deleted during the current interaction session. It was made clear that only a single record needed to be restored, as this simplified the task considerably and made it more
likely that the experiment would be completed within the allocated time. Modifications to enable this additional functionality were to be made using TextPad and Rational Rose.

As with the UNIX scripting study described in Chapter 9, a think aloud approach was used, in order to identify subjects' interpretations of the task and materials available. They were also asked to modify the design to reflect the changes they were making. The experimenter was available to provide assistance in using TextPad and Rational Rose, but not to give guidance in the completion of the task itself. Subject comments were recorded on audio tape, and the changes made to the design and code materials noted by the experimenter.

12.5. **Results**

All of the subjects began by running the application, and using some of the functionality, such as reading the student data file and calculating the course results. They then looked at the use case information, often transferring between the UML design and the application itself. Based on the verbal protocols, this was done to gain an understanding of the goals of the application. The next stage was to use the class information available in Rational Rose to understand how this functionality was achieved. This involved studying the classes used, either using the class name and description information, or by examining the code (beginning with the `CourseManagerMain` class) and tracing the functionality supplied by the classes, using the UML documentation. Only one subject made more than cursory use of the sequence diagrams; perhaps surprisingly, subjects did not use them if they could see an alternative means of getting hold of the information required.

In acquiring knowledge about how the application worked, subjects generally preferred the use of the available electronic media rather than the hard copies provided. Although the verbal protocols did not indicate why this was the case, later discussions with the subjects indicated that they had not even considered using the hard copy, and that it felt more 'natural' to maintain interaction in a single medium.
A number of approaches to the task were considered by the subjects, all of which involved modifying the existing method of deleting the student record. These approaches were:

1. marking the record as deleted rather than deleting it;
2. setting up a temporary linked list for deleted files; or
3. setting up a temporary object for the deleted student record.

In the existing design, deletion was performed using the CourseManager.removeStudent() method, as shown in Figure 12.2:
The `removeStudent()` method therein accessed the linked list, which returned a `Student` object. In the existing class, this was then discarded by the `removeStudent()` method. The approach adopted by all the subjects was to create a data member within the `CourseManager` class, which contained an object of type `deletedStudent` (or (as suggested in Chapter 9) some similar name). In accordance with the instructions, a
method was also created to restore the deletedStudent object. All three of the subjects were therefore able to generate an effective, workable solution (Cl).

12.6. Discussion

Subjects appeared to initially obtain a high-level understanding of the functionality available (A1). This was done in the first instance through the use of the application itself, then the use case information, then the classes themselves, either via the UML, the code or (most commonly) both. Subjects did not make use of the hard copy of the design and code information, which was surprising, but could have been due to the small sample size. Based on the comments made by the subjects and the navigation behaviour observed, it is suggested that although subjects would have been able to compare aspects of the system at different levels of abstraction simultaneously (using the hard copies), the ability to do so did not appear to be important to them. This may be seen to reflect the desire to work with only a single medium shown by subjects in Chapter 9, who eschewed the cards in favour of using the information sheet describing the components. It may be that this would change if the scale of the system were increased. Further to the comments on the behaviour of experienced developers described in Section 3.2, it should be noted that all the subjects made extensive use of paper notes for outlining ideas before committing design or code to electronic form.

It was interesting that only one of the subjects made significant use of the sequence diagrams to gain an understanding of how the functionality was provided. However, in the light of the level of experience of the developers, it may be surmised that the relatively small size of the application led to subjects feeling sufficiently comfortable with using the class information available within the UML design to examine the code files directly.

The successful completion of the task, together with the range of potential solutions identified, indicates that the hypothesis has been demonstrated: subjects were able to use UML in obtaining an understanding of the system in order to successfully modify it (Cl).
At the end of the study, subjects were asked to describe their impressions of UML. The general feeling was that UML was useful in 'fleshing out' the way in which the application delivered the functionality, as well as for identifying the location of the code to be modified. However, as (in this instance) the application itself was available to use, this provided an important first step in obtaining a high-level framework for understanding what the system did.

12.7. Conclusion

This study has provided empirical evidence for the suggestion that UML would be an appropriate design notation within a reuse tool. It has demonstrated that UML is a suitable notation with which to describe designs in such a way as to enable developers to gain understanding at a range of levels of abstraction. Furthermore, it has provided a degree of support for the UML notation that was not found in the literature. From a wider perspective, the study also provided further verification for the findings of Green (1990) and Guindon (1990) regarding the development of software as an opportunistic process, in the way that subjects used the design and code material in order to comprehend the problem and solution. While it would have been useful to examine group interactions in a similar study, these are not the primary focus of the work but may be studied in the future. This may include a more in-depth study of this type to further evaluate UML as a design notation in comparison with other such notations.

This chapter has therefore demonstrated that UML provides an appropriate design notation. The next chapter describes an investigation into the practicability of a system which will facilitate the storage of design information and enable it to be retrieved automatically to provide a system designer with opportunities for reuse.
"Those who cannot remember the past are condemned to repeat it."

- George Santayana

13. **DesignMatcher**

13.1. **Introduction**

While the systems implemented thus far have provided the capability to store and retrieve implementation components (UNIX shell scripts and Java components), the design aspect of system development has not yet been addressed in a research tool (except in the previous chapter where it was demonstrated that UML could provide an appropriate design notation for the reuse of design information). The importance of design in system development has been noted throughout this thesis, together with the disparities that exist between the formal view of software design and the realities of human problem solving. It is, therefore, argued that opportunities for component reuse could be identified during the design process. In the context of the developers' changing understanding of the problem and (possible) solutions, the design provides a mapping between the situation model and the system model, between the problem and the solution. While this process takes place as a fundamental part of human problem solving, software design presents a tremendous opportunity to capture the developers changing understanding of the problem and solution in a form which could be processed by computer. This in turn could allow an automatic process to take place, of identifying opportunities for component reuse based upon a 'snapshot' of the developers current understanding of the problem (as reflected in the design). **DesignMatcher (DM)**, the final tool that was developed in this research programme, was designed to explore this possibility. This chapter describes the design and implementation of the main functional aspects of the DM tool, which was intended to provide a (UML) design acquisition, storage, matching and retrieval system. An example case study is then presented to place the tool in context. The following chapter presents a preliminary evaluation of the DM tool.
As noted in Chapter 7, a component could be a design artefact, a code artefact, or a combination of these. The proposed goal for DM was to enable designs to be continually reviewed during the development process in order to identify potential reuse opportunities based on similarities between the ongoing design and previous (stored) designs. It was postulated that greater reuse efficiency would be promoted, as developers would be able to identify opportunities for reuse early in the development process, by being able to reuse aspects of an existing design, and (ultimately) by being able to retrieve code from the system which would implement the desired functionality. In this context, a design can be considered to be a 'map' of the developer's understanding of the problem and a solution. This map describes the problem, describes the solution, and illustrates the relationship between the two. So a design can therefore be used to move from a high-level description of the problem (the situation model) to the low-level implementation of a solution in code (the system model). A system has already been designed for the retrieval of components derived from code (CORE); DM was therefore focussed on reuse via the storage and retrieval of components derived from the design process.

### 13.2. Requirements

The primary goal of DM is to provide the developer with access to a store of designs that may be automatically retrieved during the development process, in response to design decisions made by the developer, which are reflected in the changing design stored by the computer. DM was intended as a research system, therefore it was felt that it should (in principle) be capable of use by many developers working in different locations, to be capable of interacting with a range of design tools and (potentially) alternative data stores, to reflect the context of software development by teams of developers.

To achieve this:

1. the system needed to be capable of storing and retrieving designs or parts of designs described using UML. As such, there was a need to be able to parse computer-based design information effectively; and
2. the interface to the system needed to represent the design information in a way which enabled the developer to both manage the amount of information available, and to rapidly identify possible reuse opportunities.

While performance issues for a research tool are felt to be less significant than for a production system (as it is the principles that are being demonstrated), poor performance is nonetheless likely to be a confounding factor in any evaluation. Consequently, it was felt to be desirable to minimise the performance overhead for the developers computer, as well as the workload overhead for the developer.

13.3. Design

13.3.1. System architecture

As with CORE, DM was to be based on a client-server model, for two reasons. As a research system, the most effective approach to address ease of access issues was to maintain a single, centralised design database and to access this from the users machine. This would avoid many of the synchronisation issues that would be associated with multiple databases potentially located on different machines.

In functional terms, the client aspect of DM would be responsible for providing the user with access to the designs, displaying matching information, and identifying files to be retrieved from the data store (Requirement 2). The server aspect of DM would be responsible for parsing and storing the designs (Requirement 1), performing the matching operations, and retrieving matched designs. This server-side functionality was expected to be considerably more processor-intensive (performance criteria).

While the system was designed to be adaptable to alternative information sources and databases, it was necessary for DM to interact with specific tools. Rational Rose 98i was selected on the basis that it was easily accessible within Loughborough University. The accessibility of Rational Rose also made it likely that possible evaluators of the tool would be familiar with it.
For development, the client machine was an Intel PII 300MHz with 64MB of RAM, running Windows NT and Rational Rose 98i. The server was a Sun Microsystems Ultra 10, with 128MB of RAM, running SunOS 5.5.1 (Solaris). The code for both systems was developed using the Java Development Kit version 1.2. The in-house indexing system pgm (which had also been used for CORE) was used to store and manage the data. While USRM and CORE were designed primarily as proof of concept systems, ease of use for DM was one of the fundamental principles of the approach. Research into End User Programming (discussed in Chapter 6) identified that user interfaces which employ concepts and styles with which the target user population is already familiar are likely to enjoy greater acceptance than those based on an entirely new set of concepts. In accordance with good Human-Computer Interaction principles, the user interface for DM was designed to use the same interface elements as Rational Rose (Requirement 2) in order to try to minimise the cognitive distance between the design interface and the DM interface.

13.3.2. Adding components to the database

Chapter 7 discussed the idea that a major factor in determining the success or failure of a component store was likely to be the quantity of components held therein. This in turn would be affected by the ease with which new components could be added. As established in Chapter 8, components can be complete designs, elements of designs, or code referenced from these designs (although, as noted earlier, the reuse of code components would not be addressed directly by DM). Following common practice in industry, it was decided that the file or files comprising a design or design fragment would be indexed by the user name, version number and a system name when they were added to DM. The developer (working on the client machine) would need to supply this meta-information. This information would then need to be submitted to DM for processing, and the meta-information and the design would then be added to the database.
As discussed in Chapter 4, UML can provide a rich source of design information. However, for the preliminary investigation presented in this thesis it was decided that only text-based information (the name and descriptions associated with design entities) would be extracted from the design. It was felt that this would be suitable as text offers a rich source of information, describing the design at a range of levels of abstraction.

13.3.3. Matching designs

Design matching is the main functional aspect of DM, and can be separated into two main elements:

1. to manage the amount of information that the user is presented with; and
2. to manage the interface in such a way that the user does not need to alter their (mind) set to comprehend the information they are being presented with.

The user of DM will potentially have access, through the system, to hundreds or thousands of components. There was therefore a need for DM to act as a preliminary filter on these components, to direct the user to those that appeared useful in the context of the work the user was doing, rather like a trusted aide in system development. A loose analogy may be drawn with the eye: Unless a signal is strong enough, it does not get passed along to the brain for further processing. In a similar way, DM should alert the user only to those components that appear to warrant closer inspection. Whether a component is 'good' or 'bad' is dependant entirely on the context within which the user is working, and hence it is not appropriate for DM to enforce judgement criteria that are too rigid (given the difficulties of describing this contextual information). For example, a component may be appropriate and suitable for immediate use. It may be functionally suitable, but slow, in which case the user may be able to adapt it to meet their needs. But the user is the best judge of whether a component is appropriate or not, since this decision is based upon the problem, the available components, the developers skill, the other elements of the system, and other factors. Accordingly, DM requires a means of performing a first-pass judgement about whether a component is likely to be useful or not.
13.4. Implementation

13.4.1. Design capture

XML was discussed in Chapter 7 as a means of describing data in an application-independent manner. Hence, it appeared to be appropriate to use XML in describing design information to be added to the component store, as this would enable a degree of platform independence within the system. Rather than define a bespoke XML data format, the XMI (XML Model Interchange) design interchange standard (version 1.0) was initially used. Like UML, this is an OMG-managed standard, with considerable industry support. However, as a relatively new standard, only one tool was found which would generate the XMI describing a UML design generated within Rational Rose. This was the XMI Toolkit from IBM’s AlphaWorks Web site (http://www.alphaworks.ibm.com). Unfortunately, a problem in using this tool became apparent during development. Version 1.05 of the XMI toolkit was used, which generated XML for the ‘Logical View’ aspect of the design only, although there was a shaded checkbox for the ‘Use Case View’, implying that it was (or would be) also possible to generate XML for this aspect of the design. However, it later appeared that this should not have been included in the original release, because there were no plans to include support for generating XML for the ‘Use Case View’ (Huang, an AlphaWorks engineer, personal communication).

This presented a considerable problem. Whilst the XMI toolkit processing was reliable, it was incomplete. To change the solution strategy and extract all the required information from the Rational Rose file would require considerable effort that could not be justified at that stage in the research. Although this was not a desirable design choice to have to make, the decision was made to extract ‘Use Case View’ information from the Rational Rose files, and ‘Logical View’ information from the (derived) XML files.

A further problem in using the XMI toolkit was that it was initially released as a PC-based system. However, the conversion of the Rational Rose file to XML took a considerable amount of processing time, particularly for large files. This was not ideal, as it conflicted
with the need to minimise the processing on the client machine. Therefore, the XMI Toolkit was moved to the (Solaris) server. Fortunately, although the AlphaWorks site recommended a PC as the host platform, the XMI Toolkit was itself written in Java. This enabled the application to be easily ported to the Solaris server, and some useful feedback was provided to IBM as a result.

The DM user interface was designed using the Model-View-Controller architecture enabled by the Sun classes within version 1.2 of the JDK. This approach promoted a clear separation between the visual aspects of the user interface and the underlying functionality. As such, it enabled changes to be made easily to either aspect.

13.4.2. Matching designs

The main feature of DM was intended to be its ability to compare ongoing designs in order to identify similarities with stored designs. A design in this context was considered to be the set of UML-based descriptive information associated with a system development. DM would store design information that had been described using UML. Matching on designs would then be performed at a textual level, with the text associated with different abstraction levels of the ongoing design (e.g. 'Use Case View', 'Logical View', classes, actors, and so on) compared with similar abstraction levels from existing designs. It was postulated that the most common situation would occur when a design in progress was to be compared to designs already stored in the system. After the ongoing design had been parsed, each data item of a given type would be compared with all other known data items of the same type. To aid this comparison, component names (e.g. class names) could be parsed into their constituent words, using the tools available within pgm. For instance, many developers use known shorthand notations for method and class names when designing or coding, distinguishing individual words using underscores or camel notation. DM would need to break this notation down to identify the constituent words that describe an entity, and use these individual words to match on terms already known to the system. Similar matching would be performed on the description text.
For example, two classes intended to process files might be called FileOps (using camel notation) and File_Reader (using underscore notation). DM should process these strings into “File Ops” and “File Reader” before performing a comparison operation. Each word in the search string would then be matched against each word in the stored string. If a 'hit' was made, matching against the next word from the search string would then be used. Similar matching would be performed on the descriptive information associated with each entity. Duplicate words would be ignored, thus reducing the effect that the repetition of the same words (particularly in large strings) could have in skewing the results. When matching was complete, the number of matches could be divided by the length of the match string, to generate a percentage score. So if the search term FileOps were compared to the stored term File_Reader, a score of 50% similarity would be achieved, as half of the (constituent) words in each string match. This scoring approach was chosen because some form of metric was required that would give the user a means of establishing whether DM itself had identified a possible useful match. This was in line with the concept of DM as a ‘trusted aide’ in development, indicating potentially useful components to the user.

This approach is similar to that used by Google and other Web-based search engines. For example, Google reads in search terms from users, and performs matches based on the titles of Web pages and the terms that have (sometimes) been included in the page as short descriptions. In the same way, DM would use the names of components and their associated descriptive information to match what the user is working on against known components.

As noted earlier, no matching on diagrams would initially be performed. Whilst this would undoubtedly have been an interesting field of study, the use of text as the basis for matching in this prototype was chosen because a suitable text processing and matching tool (pgm) was already available to undertake a preliminary investigation of the concepts.
13.5. **System initialisation**

For DM to be useful, it needed to be initialised with existing designs. This required both a source of designs and a means of adding them easily. Unfortunately, efforts to obtain designs from external commercial sources failed (due to reasons of commercial sensitivity). However, in the academic year 1999/2000, a new module was started in the Computer Science department at Loughborough University, teaching OO design and programming using Java and UML. The coursework for this (from 140 students) presented a useful source of designs, which had associated code. Whilst these designs addressed the same problem at a high level, there were differences in the detailed understanding of the problem, and the low-level implementations varied considerably.

In order to verify the parsing routines that had been developed for the UML data, a large-scale test was performed using the student coursework. This used a command-line based tool to add all the designs to the design store. Whilst this indicated some minor problems with the parsing routines, these were related to the use of different versions of Rational Rose that had been used by the students.

13.6. **Example of use**

For the purposes of explanation, it is assumed that the developer would follow a conventional design process, and produce a statement of requirements. This would then be elaborated upon in an iterative analysis and design process, using the UML design tool. Initially, actors and use cases would be defined, each of which should have a brief description associated with it. Taken as a group, these actors and use cases illustrate the basic functionality of the system, and identify the (external) entities with which it would interact. As development proceeds, the way that these initial use cases are implemented would then be described in increasing detail, via classes, methods, interfaces, and so on.

At each stage of development, reuse opportunities may be identified. The role of DM would be to use the current state of the design to notify the developer of similar (stored) designs that may provide opportunities for reuse. When the DM client is started, the user
would indicate to the system the location of the file containing the (developing) UML design via a file dialog. At this point, the DM client would read the file and send it to the DM server. This would compare the design with those known to the system and attempt to identify matches based on the available text information.

![Diagram](image)

Figure 13.1: Use case diagram for environment management system

To illustrate the proposed operation of the system, Figure 13.1 shows use cases and actors associated with the design of an environment management system. This is the design upon which DM will perform the analysis. The developer has given DM the name of the operating file containing this design, which will be read by the client as requested by the developer. This is then sent to a ‘holding area’ on the server. A process running on the server identifies that a new or updated design is available, and performs matching on it.
The results of the matching process are then returned to the client and displayed as shown in Figure 13.2:

Figure 13.2: The DM interface showing the results of matching. The ongoing design is in the left-hand pane, while the matches are shown in the right-hand pane.

13.7. **DesignMatcher client interface**

While DM is, like USRM and CORE, primarily intended as a research system, interface issues were of considerable importance, as ease of use throughout the development and component reuse process was a key design consideration – it was essential that component reuse be perceived as a natural part of system development. As may be seen in Figure 13.1 and Figure 13.2, the DM interface displayed design information in a similar format to that used by Rational Rose, with the names of entities shown in a tree format, next to the score that that entity had been awarded by DM. This should enable developers to focus
on the entities which were likely to be useful. As may be seen in Figure 13.2, the relationship between the ongoing design and the matched design(s) was made explicit through visible links. The DM was capable of displaying more than one design and a match simultaneously, however, it was anticipated that the practicalities of screen area and usability issues would limit the number of matched designs that could usefully be displayed. Each time a match was found for an item in the ongoing design, the matched item was given its own entry in the ‘ongoing design’ pane (this may be seen in Figure 13.2 with the ‘getTemperature’ method).

When a developer selects a matched entity, its description information is displayed in the part of the pane below the tree, together with the match score. In this way, the developer should be able to assess the suitability of the match to the ongoing task. In order to minimise the potential for visual confusion in the information displayed in the interface, a number of additional filters were provided. These enabled the developer to display only items that matched within a particular score range (such as scores from 70% to 100%), to display only items that matched the selected item, or only items that were currently visible within the tree.

If a match was found that the developer wished to retrieve, the item could be selected, and the design component then retrieved from the design store. The design would be written to a temporary file for a more detailed examination by the developer. Given a suitable match, the developer was thus able to retrieve and modify an existing system (in whole or in part) rather than generate an entirely new system. In this way, reusing portions of existing systems could save development effort by identifying existing designs that had addressed similar requirements (as shown in use cases) early in the design process, through to lower levels of abstraction in the design, through classes, methods, interfaces and so on. It would be the designers decision whether to retrieve such components, and incorporate them into the ongoing design. Once the system had been developed, the completed design could be added to DM for future reference and potential reuse.
13.8. Conclusions

The DM tool was intended to enable developers to retrieve designs with minimal effort, enabling them to focus on their primary task. However, DM was very much a research tool, and this was reflected in some of the implementation decisions that were made. For example, Rational Rose was produced by a company that employs several of the developers of the UML standard, and it was believed that this made it probable that Rational Rose would itself support the standard consistently. Unfortunately, experience with the tool indicated that this was not the case; however by the time this had been determined, much of the system implementation had been designed around Rational Rose. In addition, a considerable amount of time was spent attempting to parse the Rational Rose file format to extract use case information when the XMI tool failed to do so effectively. This was later addressed by using a more recent version of Rational Rose, which had a scripting mechanism that enabled all the required information to be written directly to a file. While at the time of writing DM operated with a specific design tool, the format of the extracted data was similar to XML. The production of a utility to transfer the design data from the tool into the application-independent XML format would be required to enable DM to communicate with other design tools.

This chapter has described the design and implementation of DesignMatcher as a tool to promote reuse by making potentially reusable components from previously generated designs available to developers as they create new designs. It has demonstrated that a tool of this type can be constructed, and has shown how the issues of acquiring design-based components for the design store and providing design information can be addressed without requiring a designer to do much additional work. These were the key objectives for this stage of the investigation into facilitating reuse. Nevertheless, it was felt that it was essential to at least carry out a preliminary evaluation of DM since no tool can be claimed to have any practical value unless it can be used effectively. The next chapter, therefore, describes the preliminary evaluation that was performed.
14. DesignMatcher evaluation

14.1. Introduction

This chapter describes a single-subject evaluation that was conducted with DM in order to establish a prima facie case for the principles that underlie the prototype. It was therefore similar to the study in Chapter 9 in that it was intended to establish the suitability of DM as a basis for component reuse. The hypothesis of Chapter 9 may thus be adapted for this situation:

If:

A1: DM provided the evaluator with access to existing components at the point at which they were relevant to the current problem;
A2: DM avoided overloading the evaluator with information not immediately pertinent to the problem;
A3: the evaluator was capable of modifying the ongoing design; and
A4: the problem domain was readily understood and appreciated by the evaluator;

Then:

C1: the evaluator should be able to combine appropriate aspects of the existing designs stored in DM with the ongoing design to generate a solution that is either felt to be more complete, or that is generated more quickly, than if DM had not been used.

14.2. Subject

The evaluation was performed by a software engineer (hereafter referred to as ‘the evaluator’) known to the author who was experienced in UML (mentioned in Section 12.3). The ability of the evaluator to use UML designs had been demonstrated in the
UML study reported in Chapter 12, thus demonstrating that they were capable of modifying the ongoing design (A3).

14.3. Method

The problem to be addressed was the creation of a simulation of a temperature control system for a building. The project brief for this system stated that:

"The simulation should be designed to enable the independent control (via a thermostat) of the temperature of each room on each floor, sourced from a single centralised heating centre. Primarily, heating will be controlled by a timer, similar to that used in a domestic heating system. The only exception to this is if a manual override is activated, when temperature control will be on independently of the timer, but still subject to thermostat control.

Provision must be made for both heating and cooling. The system should be modular and easy to use. As part of this modularity, individual rooms may be bought or rented by an organisation, which may place its own restrictions on the maximum and minimum temperatures for the room. As this is a simulation, all events must be written to a log file."

This was a problem that had already been addressed by first year undergraduate students taking a module in the Department of Computer Science at Loughborough University on OO programming. There were therefore around eighty designs available with which to initialise DM. As the designs had been taken from first year student coursework, the quality of the designs could not be expected to be as uniform, or as high, as if the experiment were conducted in a commercial environment. Six solutions were therefore selected (by the author and the lecturer taking the course) on the grounds of:

- meeting the requirements;
• demonstrating an effective understanding of OO principles (as outlined in Section 4.6).

These designs were then parsed and added to the DM database (A1). The addition of each design to the database took approximately one minute.

For the analysis phase of development (the ‘Use Case View’ within Rational Rose), design matching was performed relatively late in the process, after the evaluator had expressed a degree of satisfaction with the actors and use cases identified. After reviewing the results of design matching, the evaluator then made a number of changes to their design. For the design phase of development (the ‘Logical View’ within Rational Rose), matching was performed at an early stage after only a few classes and methods had been identified. The results of this early matching process were then used to influence the ongoing design, before a third iteration of matching and modification was performed on the results.

The evaluator was instructed that there was no need to include details in their design of a graphical interface to the system they were developing, as this was extraneous to the primary goal of the evaluation, of identifying opportunities for functional reuse using DM.

14.4. DesignMatcher use: analysis

The problem specification was examined by the evaluator. On the basis of this, they then identified use cases and actors, and described them as a UML design using Rational Rose. One of the recommended approaches (by the software development community) to this task was adopted, by studying the requirements document and making (paper-based) notes on probable candidates for use cases and actors. The evaluator was familiar with the problem domain under consideration, and did not request further information, demonstrating that they understood and appreciated the problem domain (A4). A single iteration of analysis was performed, taking around 20 minutes. DesignMatcher was then used to evaluate the analysis against the stored analyses. This was done in order to identify the effectiveness of the tool in the early stages of development, once the evaluator
had begun to establish an understanding of the problem, but before describing a possible solution. Based on the work of Guindon (1990) discussed in Section 2.3 it was expected therefore that the evaluator would have an understanding of the problem, but that DM would provide further insight (and possible solutions) from other peoples perspectives. The partial design (created by the evaluator) upon which matching was performed is shown in Figure 14.1:

![Figure 14.1: Initial analysis of heating simulation using UML in Rational Rose. The actors and use cases identified are shown, together with the relationships between them.](image)

Figure 14.1 shows the basic use cases and actors which appear to be important in the simulation have been identified in the evaluators' design. The relationships between the actors and their associated use cases have also begun to be established. Based on this data, a design matching process was carried out and the results are shown in Figure 14.2, with elements of the ongoing design in the left-hand frame, and the design matches in the right-hand frame:
For the purposes of the evaluation, only results which achieved a match (based on item name or description text strings) of 40% or better were shown. It may be seen that the icons and structure of the information is very similar to that used in the left-hand pane of the Rational Rose interface (Figure 14.1). By designing the interface in this way, the evaluator was not overloaded with information not immediately pertinent to the problem (A2).

The evaluator spent some time reading the descriptions from the matched items, comparing them to the existing items. On the basis of the matched results, a number of modifications were made to the existing use cases. The evaluator placed their existing ‘Engineer’, ‘Manager’ and ‘User’ actors in a subclass-superclass relationship to reflect their common functionality, after the evaluator had read the description information associated with an existing ‘Engineer’ actor from a stored design. In addition, a temperature control system was introduced. This had been referenced in the descriptive
information for the 'Manual override' use case from an existing design (shown in the right-hand pane of Figure 14.2). It was noted that the spelling in some of the retrieved matches was poor, and the evaluator commented that this was a potential problem in terms of the probable retrieval success.

Having performed an initial analysis of the problem and revised it based on the results from DM, an initial set of classes (together with associated attributes and methods) was then produced by the evaluator. This initial attempt was concerned only with the identification of classes from the available data rather than with establishing relationships between the identified classes. This is shown in Figure 14.3:

![Figure 14.3: The evaluators initial class identification, showing the attributes and methods of the identified classes](image)

The matching process was performed on this data and generated the results shown in Figure 14.4:
The evaluator then made a number of changes to the existing design based on a review of the retrieved results:

- the issue of control over which actor could influence different aspects of the system was identified as needing to be addressed after the evaluator reviewed the RMShowGUI class that had been retrieved;
- the need for a unique identifier for each room in the simulation was established;
- in functional terms, the need for a time controller, as well as a means of identifying the heating and cooling rate, was identified and added to the ongoing design as a result of reviewing the matches;
In the ongoing design, a `setTolerance` method was used in the `Thermostat` class to establish the sensitivity of the `Thermostat` readings (and thus to establish the time delay in any temperature change). In the matches, there was a method called `tempLimitState`; this was identified by the evaluator as an alternative approach to performing the same function.

After identifying and making these changes, the matching process was executed for a third time. The results of this are shown in Figure 14.5:

![Figure 14.5: Matching on identified classes following the changes made in response to an analysis of the design matches identified in Figure 14.4.](image)

Although more results were returned on this occasion (61 matches rather than 44 with the matching process shown in Figure 14.4), fewer changes were made by the evaluator as a result of the matching process. Furthermore, these were smaller changes related primarily to refining the existing approach, rather than introducing new classes.
After making these changes, it became clear that the emerging design would provide a satisfactory solution to the stated problem.

The evaluator concluded that they were able to combine appropriate aspects of the existing designs stored in DM with the ongoing design to generate a solution more quickly, than if DM had not been used (C1). They were not willing to conclude that the design was of higher quality due to the source of the designs that were used, but the design was produced more rapidly and fulfilled the design brief.

14.5. Discussion

The evaluation has raised a number of areas of concern. While some of these are minor (largely interface) issues, others are more significant and will need to be addressed in more detail.

Poor spelling might unduly restrict the number of matches, and a lack of design documentation could make it difficult to reuse a design. Both of these concerns related to the quality of the designs which had been captured in the tool. The problem could, potentially, be overcome in practice if entries were verified after being captured. This could be addressed by verification when the design data was initially collected from the design tool. It was noted that one of the reasons for this may have been the source of the stored designs being student coursework. While work from experienced software engineers would probably have been of higher quality (due to the existence of coding standards and quality control), it was felt that spelling was still an issue. This was considered to be a minor problem that could be addressed by an element of preprocessing.

The evaluator noted a feeling of satisfaction from comparing their proposed design decisions to the components stored in DM. The matching process was felt to be a useful approach to gaining a broader perspective on the many factors to be considered in producing a design. This could offer a safe(r) design space for the developer, while retaining a high degree of personal choice. In particular, DM helped to identify design
omissions early in the process. The evaluator did not feel that they were simply copying stored components, but rather that DM was acting as an assistant during the design process, highlighting other approaches to addressing a problem.

14.6. Conclusions

Although evaluation by a single evaluator cannot enable any firm conclusions about the value of the DM approach to be drawn, it has indicated that the tool can be used successfully. It has demonstrated that when initialised with similar designs, the prototype DM tool enabled an experienced developer to use existing designs to produce an effective design in a shorter period of time than would otherwise have been possible. It was important that the system left the evaluator feeling in control of when DM suggested changes, to avoid the possibility of the system simply being ignored through unwanted interruptions in the workflow.

It has also confirmed that, at least as far as the evaluator was concerned, the tool achieved its second aim of not placing an excessive additional workload on the developer. An unexpected outcome of the evaluation was the observation that the matching process might identify extra features of, or alternative approaches to, the problem which the developer had not previously considered.

The previous chapter described the design and development of a tool which could facilitate design reuse and this chapter examined the preliminary evaluation which had been performed. The next chapter describes in greater detail the rationale behind the DM approach.
15. The case for the DesignMatcher approach

15.1. Introduction

DesignMatcher is the most comprehensive proof-of-concept system developed in the course of this research, in terms of its integration with the design process. The DesignMatcher approach builds on the concept of multiple component descriptions that was initially demonstrated by USRM (described in Chapter 10) and complements the (code) component retrieval tool demonstrated by CORE (described in Chapter 11).

Early on in this work, it was argued that an approach to component reuse which was dominated by the technical factors (e.g. Devanbu et al., 1991; Fischer et al., 1994) or the human / organisation factors (Carroll et al., 1994) would be less effective in a real-world context than an approach which sought to build on human problem solving strengths (described in Chapter 2) and the strengths of computers as problem solving and information retrieval systems. The DM approach therefore seeks to address the problems of component reuse at a technical level and at a human level. This chapter puts the case for the DesignMatcher approach, considering it both at a technical level as an information retrieval tool, and in the broader context of how it is to be used by individual problem solvers within an organisational context.

15.2. DesignMatcher as an information retrieval system

According to the definition of Lancaster (1968), DM is an information retrieval system, since it “does not inform (i.e. change the knowledge of) the user on the subject of his enquiry. It merely informs him of the existence (or non-existence) and whereabouts of documents relating to his request”. DM conforms to this description since it presents components considered to be relevant to the user (based on the users' understanding of their problem as expressed in their design). This enables the user to decide which of the retrieved components is useful. DM may therefore be considered to act as an aide in the software development process, highlighting elements of interest to the user, but letting the...
user, with their deeper knowledge of the problem, make the final judgement over whether, and how, to use any retrieved component.

Cleverdon (1978) identifies two major factors in the evaluation of an information retrieval system: performance and cost. These factors will be addressed separately.

15.2.1. DesignMatcher performance

The performance of information retrieval systems is based around the notions of relevance and recall. Relevance is "...a measure of the contact between a source and a destination, i.e. between a document and its user" (Chowdhury, 1999). Recall is "...the extent to which the retrieval of wanted items occurs" (Chowdhury, ibid.). To assess the relevance and recall of the components identified by DM, an experienced software engineer ('the expert') was given the same task as DM (described in Chapter 14). This was to go through the evaluators' design at each of the 3 stages when matching was requested, and to identify which of the stored components they considered relevant. The expert was presented with the users design as it stood when DM performed matching, and went through the design store noting components that they judged to be relevant to the design at that point. The results are shown in Table 15.1:

<table>
<thead>
<tr>
<th>Components judged relevant by</th>
<th>DM and expert</th>
<th>Available components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expert</td>
<td>18</td>
<td>73</td>
</tr>
<tr>
<td>DM</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Logical view match</td>
<td>26</td>
<td>746</td>
</tr>
<tr>
<td>DM</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Table 15.1: DM matching versus expert matching for the same stage in the users design

For both the user case and logical view matches, DM judged as relevant 38.9% of the same components as the expert. The expert was then shown the components that had been identified as relevant by DM, but which they (the expert) had not initially identified as relevant. They were then asked to identify whether they felt these were relevant. The results are shown in Table 15.2:
These measures may be expressed more clearly using the concepts of precision and recall. Precision is defined by Cleverdon (1978) as retrieving "only those items that were relevant", and Chowdhury (ibid.) states the general formula for calculating precision as the number of relevant items retrieved divided by the total number of items retrieved, expressed as a percentage. Recall is defined as the extent to which the retrieval of wanted items occurs, and is calculated by expressing, as a percentage, the number of relevant items retrieved divided by the total number of relevant items in the collection:

<table>
<thead>
<tr>
<th>Use case match</th>
<th>Logical view match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recall</td>
<td>Precision</td>
</tr>
<tr>
<td>39%</td>
<td>88%</td>
</tr>
<tr>
<td>78%</td>
<td>54%</td>
</tr>
</tbody>
</table>

Ideally, recall and precision for an information retrieval system will both be 100%; however a number of information scientists have argued that there is an inverse relationship between recall and precision: the more items are recalled, the less likely it is that the items will be relevant. In most real-world applications, users want a manageable number of responses to any query, but with a high degree of precision. Chowdhury (1999) suggests that most information retrieval systems are designed to perform at a moderate level of recall and precision (around 50-60%). These figures are therefore indicative that the system is operating successfully as an information retrieval tool.

For DM, the components which were relevant and not identified, or which were irrelevant, were examined in more detail. The identification of irrelevant components, or the failure
to identify relevant components, can be broken down into syntactic issues and problem solving issues. The syntactic issues were:

- **poor notation style:** As discussed in Section 13.6, DM breaks down underscore and camel notations to identify the component words used in a system. In some instances, 'aSampleComponent' had been written as 'asamplecomponent', leaving DM unable to break the words down and thus perform a matching operation;
- **poor spelling:** in a number of instances, the spelling of terms was incorrect; and
- **differences in terminology:** For example, the ongoing design had identified a 'Manager' actor. There was also a stored actor component that was termed 'BuildingManagerUser' which did the same task. However, all the actors in the design from which the 'BuildingManagerUser' came had the 'User' suffix.

These syntactic issues may be at least partially addressed with known techniques. At a technical level, spell checking and thesaurus tools would go some way towards addressing terminology and spelling issues. However, it is also worth noting that within a team of software engineers, it is probable that there would be normalising factors which would lead to strong similarities in writing style, which would aid component matching and retrieval.

As discussed in Section 2.3, there are inherent semantic differences between individuals in their understanding of problems, particularly in terms of the low-level details of any problem. In addition, software engineers are likely to separate system functionality between different classes in different ways depending on their understanding of the problem, their level of experience, and other factors. These issues are not felt to be weaknesses of the DM approach; its strength comes from being able to identify components that may be useful, but which will have been generated by software engineers with different perspectives. Consider a 'perfect match': a component from the design store that perfectly matched a component that had been described by the user. It could be argued that this component was of no value, since it did not in itself add anything new. However, the (design) context within which that component was developed would almost
certainly be of interest to the user, since it would suggest that a problem (or sub-problem) under consideration had been addressed elsewhere. The DM approach would give the user the ability to retrieve that contextual information, and thus to benefit from components that had already been developed.

15.3. **DesignMatcher in use by individual problem solvers in an organisational context**

Lancaster (1979) identifies a number of factors when considering the cost of use of an information retrieval system:

1. Cost incurred per search
2. Users’ efforts involved
   - In learning how the system works
   - In actual use
   - In getting the documents through document delivery systems
   - In retrieving information from the retrieved documents
3. Users time
   - From submission of query to the retrieval of references
   - From submission of query to the retrieval of documents and the actual information

Conventional information retrieval systems are predicated on the notion of the user providing the system with a set of search terms: that is, identifying to the system details of what is to be retrieved. However, DM is based on the idea of these ‘search terms’ being generated *in the course of the users’ work*. Rather than add a (time and effort) cost to development by asking the user to step outside the work process to formulate a query, DM is based on the notion that the product of the work itself (the design) forms an accurate, detailed representation of the users’ understanding of the required components.

This approach is supported by the cost incurred per search being very low: the user selects an icon, which triggers DM to perform the matching operation and return the results. The
effort required of the user to analyse the results and understand what is being presented is similarly low: the system has been designed to use the same interface style as the design tool Rational Rose, which itself uses the standard UML icons. This minimises the cognitive distance between the design tool (e.g. Rational Rose) and the matching tool (DM), enabling the user to retain the same mindset when identifying components to reuse as when designing a system. As no shift in mindset is required, the system is likely to be regarded as extremely easy to use, and this has been the case with the formal and informal evaluations conducted so far.

The amount of time involved in matching components is kept very small. This results from the matching process being performed on a server, enabling the performance of the users’ machine to be kept at a high level. The matching process itself is extremely quick; the pgm indexing system was designed to manage data sets of far greater size than those being used here. If necessary, better performance could be achieved through code optimisation and by upgrading the server. As noted in Chapter 7, one of the major factors in the success of any component retrieval system is likely to be the ease with which new components may be added. DM was at one stage initialised with around fifty designs, each of which took less than one minute to add. While adding a new design (i.e. set of components) to the component store does require the user to enter the user name, version number and a system name, the addition process itself is fully automated, requiring no further human intervention.

15.4. Conclusions

The DesignMatcher approach is based on an understanding of human problem solving supported by appropriate use of technology. DM is an information retrieval system that has been integrated into the problem solving process, retrieving components based on a comprehensive view of the information required, that has been produced as part of the human problem solving process (within the broader organisational work process). It acts as an aide to the user in the development process, providing a filter for a large number of components within the store, to allow the user to focus their attention on components that
are likely to be of relevance to the ongoing design. As the matching process is performed on a different computer to the design process (and takes place extremely quickly), the user is able to continue working without needing to take time out from the design process.

This approach, together with the interface and interaction design, has enabled the overall cost to the user to be minimised. Initial indications suggest that DM is a useful component retrieval tool, but that a number of modifications can be made in order to improve its performance.
16. Conclusions and further research

16.1. Introduction

This chapter evaluates the research and suggests areas for future development. The goal of the work was to investigate the production of a more effective support environment for the problem-solving activity undertaken by the human using a computer. Supporting human problem solving activity is important commercially due to the amount of effort currently required to develop computer-based systems. In order to address this, work was carried out to examine how a strategy of component reuse might be supported. This has been a desired strategy in software development for some time, but has not been implemented widely. This thesis has proposed that a possible reason for this is that the conventional view of software components is fundamentally flawed, since it does not account for the subjective nature of the information which components process, or the flexibility of the human problem solver. Many existing support systems appear to have been designed to meet the perceived organisational requirements for problem solving, but have failed to support the iterative nature of the human problem solving process.

Four guiding principles were derived from the literature search and explained in Chapter 8. These are examined below in conjunction with the prototype systems which were implemented to explore how they might be addressed.

a) No single representation is appropriate for all problems

Research (discussed in Section 2.4) has shown that problem solving is strongly influenced by the representation(s) used. The DesignMatcher and CORE tools together demonstrate the storage and retrieval of design and code components at a range of levels of abstraction. DesignMatcher also demonstrates that a design process that employs multiple representations with which to manage abstraction at different stages of the problem-solving process could be supported.
b) **Effective component reuse must encompass multiple perspectives**

The study of component reuse described in Chapter 9 suggested that the background of users contributed to their perceptions of what a component could be used for. This concept of multiple perspectives (resulting in multiple descriptions) was implemented in the USRM, which demonstrated the concept of multiple descriptions for a single component, as well as multiple components for a single description. This approach was also used by DesignMatcher, which demonstrated that a rich source of information about components could be provided by employing design information that described components at multiple levels of abstraction.

c) **Effective component reuse systems must add only a minimal additional workload to be accessible**

Previous attempts to incorporate component reuse into the development process, such as Domain Analysis (DA), have been limited in their accessibility. While they have proven successful in restricted problem domains, they require a significant investment in time and effort to establish. For component reuse to be a realistic approach to software development by smaller organisations and individual programmers, it must be accessible in terms of the time and skill demanded. In practice, the way that components are added, processed and ultimately examined by the developer must require as little overhead as possible. DesignMatcher demonstrates that it may be possible to produce a system that can be used effectively with very little effort from the developer, by building on existing skills.

d) **Exploring possible solution spaces informs the capture and understanding of requirements**

Component reuse strategies such as DA require the problem solver to know what problem is being solved in some detail. Research into human problem solving has shown that in practice, this rarely happens. Rather, problem solvers move (often rapidly) between an exploration of one aspect of a solution and an increased understanding of the problem. The evaluation of the DesignMatcher tool indicates that it supports such a problem-solving
strategy and suggests that this may form the basis for an effective support mechanism for practical component reuse.

16.2. Further research

The approach described in this thesis appears to be an effective basis for successfully implementing a practical component reuse strategy in a wide range of software development contexts. This section describes a series of developmental steps that offer a possible route towards verifying the accessibility and acceptability of the approach.

I. Evaluation of the DesignMatcher tool

Although DesignMatcher has been successfully implemented, time constraints have prevented a more thorough evaluation of the system than the prima facie case discussed in Chapters 14 and 15. In addition to this detailed evaluation, further avenues of investigation would examine the use of the system across a number of problem domains and user types, which are shown in Table 16.1:

<table>
<thead>
<tr>
<th>User type</th>
<th>Problem domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experienced user</td>
<td>Familiar</td>
</tr>
<tr>
<td>Experienced user</td>
<td>Unfamiliar</td>
</tr>
<tr>
<td>Inexperienced user</td>
<td>Familiar</td>
</tr>
<tr>
<td>Inexperienced user</td>
<td>Unfamiliar</td>
</tr>
</tbody>
</table>

Table 16.1: Proposed evaluation for DesignMatcher

This would enable the effectiveness of the approach as a basis for assisting developers of different skill levels in different problem domains to be assessed. The interface and support mechanisms for the tool can then be modified appropriately.

II. Implementation of the DesignMatcher tool within well defined problem domains

While the weaknesses of domain-dependent approaches have been discussed extensively in this research, another approach to evaluating the DesignMatcher tool in use would be to
employ it within a specialised problem domain, where the language used to describe problems and solutions is relatively consistent. This would constrain the size of the problem space and thus enable fine-tuning of the component matching process, allowing the human role within the approach to be studied in more detail.

III. Investigation of the use of diagrams within development
At present, all of the component information that is stored and processed is textual in nature. This clearly does not utilise all of the available information; diagrams play a significant role in enabling developers to map between the situation and system model aspects of a design, and to manage abstraction. Investigation into the use of diagrams within development is likely to be a valuable source of information into the problem solving process and the way that information is managed within it.

IV. Implementation of alternative interfaces for the DesignMatcher tool
While the current user interface of DesignMatcher is believed to be effective for the current target user group of software engineers, the literature search established the potential benefits of representations tailored to a particular problem domain (such as those used in EUP). Research is needed to identify the benefits of implementing alternative interfaces tailored to specific user groups or problem areas, while maintaining a common underlying data structure to enable sharing between different groups.

16.3. Conclusion
This thesis has investigated an alternative approach to problem solving using computers by offering a method for component reuse that endeavours to accommodate the technical, human and organisational factors involved. A series of prototype tools has been developed to investigate principles identified from the literature search, demonstrating the storage and retrieval of design and code components at a range of levels of abstraction, representing and illustrating how multiple perspectives and descriptions of a reusable component might be made available. At the human level, the approach was designed to support the human problem-solving process to require only a small additional effort from
the developer. A preliminary evaluation of DesignMatcher suggests that the approach is acceptable to developers and can be a useful tool in the production of effective component-based solutions. Ways of further checking the validity of the approach and developing it have been proposed, and are believed to address the range of factors involved.
17. References


Booch, B. W., 1974, *Some steps towards formal and automated aids to software requirements analysis and design*, Proc. IFIP 74 Congress, North-Holland.


Chandrasekaran, P., 1997, *How use case modelling policies have affected the success of various projects (or how to improve use case modelling).* In: Conference on Object Oriented Programming Systems Languages and Applications.


Devanbu, P., 2000, personal communication.


Huang, H-L., 1999, personal communication.


McIlroy, M.D., 2000, personal communication.


18. Appendix A: UNIX scripting experiment results

This Appendix presents the results from the experiment discussed in Chapter 9. The solutions are presented here as the subjects provided them.

18.1. Subject A: 2nd year English undergraduate

Session 1

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mail the file “Joe.txt” to “Fred”</td>
<td>List * Mail user</td>
</tr>
<tr>
<td>Give the file called “work.doc” the name “play.doc”</td>
<td>List * Rename work.doc play.doc</td>
</tr>
<tr>
<td>Put the contents of all files into a file called “New”</td>
<td>List * Copy existing_filename new_filename Rename existing_filename New</td>
</tr>
<tr>
<td>Make a duplicate of the file “old_data.txt”, calling it “new_data.txt”</td>
<td>List * Copy old_data.txt new_data.txt Rename old_data.txt new_data.txt</td>
</tr>
<tr>
<td>Check if the file called “nursery” contains the text “Mary had a little lamb”</td>
<td>List * display nursery Find “Mary had a little lamb”</td>
</tr>
</tbody>
</table>

Session 2

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put the contents of all files into a file called “New”</td>
<td>List * Display * &gt; New</td>
</tr>
<tr>
<td>Check if the file called “nursery” contains the text “Mary had a little lamb”</td>
<td>List * display nursery</td>
</tr>
<tr>
<td>Give the file called “work.doc” the name “play.doc”</td>
<td>Rename work.doc play.doc</td>
</tr>
<tr>
<td>Make a duplicate of the file “old_data.txt”,</td>
<td>List * Display old_data.txt</td>
</tr>
<tr>
<td>calling it &quot;new_data.txt&quot;</td>
<td>Copy old_data.txt new_data.txt</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>Mail the file &quot;Joe.txt&quot; to &quot;Fred&quot;</td>
<td>List * display</td>
</tr>
<tr>
<td></td>
<td>Show users</td>
</tr>
<tr>
<td></td>
<td>Mail user</td>
</tr>
</tbody>
</table>
### 18.2. **Subject B: 2nd year human biology undergraduate**

#### Session 1

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mail the file &quot;Joe.txt&quot; to &quot;Fred&quot;</td>
<td>List</td>
</tr>
<tr>
<td></td>
<td>Display Joe.txt</td>
</tr>
<tr>
<td></td>
<td>Copy existing_filename new_filename</td>
</tr>
<tr>
<td></td>
<td>Mail user</td>
</tr>
<tr>
<td>Give the file called &quot;work.doc&quot; the name &quot;play.doc&quot;</td>
<td>List</td>
</tr>
<tr>
<td></td>
<td>Display work.doc</td>
</tr>
<tr>
<td></td>
<td>Rename work.doc play.doc</td>
</tr>
<tr>
<td>Put the contents of all files into a file called &quot;New&quot;</td>
<td>List</td>
</tr>
<tr>
<td></td>
<td>&gt; New *</td>
</tr>
<tr>
<td></td>
<td>&gt; rename existing_filename new_filename</td>
</tr>
<tr>
<td>Make a duplicate of the file &quot;old_data.txt&quot;, calling it &quot;new_data.txt&quot;</td>
<td>Copy old_data.txt new_data.txt</td>
</tr>
<tr>
<td></td>
<td>Rename old_data.txt new_data.txt</td>
</tr>
<tr>
<td>Check if the file called &quot;nursery&quot; contains the text &quot;Mary had a little lamb&quot;</td>
<td>List</td>
</tr>
<tr>
<td></td>
<td>Display nursery</td>
</tr>
</tbody>
</table>

#### Session 2

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put the contents of all files into a file called &quot;New&quot;</td>
<td>List</td>
</tr>
<tr>
<td></td>
<td>&gt; filename *</td>
</tr>
<tr>
<td></td>
<td>&gt; rename existing_filename New</td>
</tr>
<tr>
<td>Check if the file called &quot;nursery&quot; contains the text &quot;Mary had a little lamb&quot;</td>
<td>List</td>
</tr>
<tr>
<td></td>
<td>Display filename</td>
</tr>
<tr>
<td>Give the file called &quot;work.doc&quot; the name &quot;play.doc&quot;</td>
<td>List</td>
</tr>
<tr>
<td></td>
<td>Rename work.doc play.doc</td>
</tr>
<tr>
<td>Make a duplicate of the file &quot;old_data.txt&quot;, calling it &quot;new_data.txt&quot;</td>
<td>List</td>
</tr>
<tr>
<td></td>
<td>Copy old_data.txt new_data.txt</td>
</tr>
<tr>
<td>Rename old_data.txt new_data.txt</td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td></td>
</tr>
<tr>
<td>Mail the file “Joe.txt” to “Fred”</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td></td>
</tr>
<tr>
<td>Show users</td>
<td></td>
</tr>
<tr>
<td>Mail user</td>
<td></td>
</tr>
</tbody>
</table>
### Session 1

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Give the file called “work.doc” the name “play.doc”</td>
<td>List</td>
</tr>
<tr>
<td>Rename work.doc play.doc</td>
<td></td>
</tr>
<tr>
<td>Make a duplicate of the file “old_data.txt”, calling it “new_data.txt”</td>
<td>List</td>
</tr>
<tr>
<td>Rename existing_filename New</td>
<td>Copy</td>
</tr>
<tr>
<td>Check if the file called “nursery” contains the text “Mary had a little lamb”</td>
<td>Display nursery</td>
</tr>
<tr>
<td>Put the contents of all files into a file called “New”</td>
<td>Display * &gt; New</td>
</tr>
<tr>
<td>Mail the file “Joe.txt” to “Fred”</td>
<td>Mail Joe.txt</td>
</tr>
</tbody>
</table>

### Session 2

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put the contents of all files into a file called “New”</td>
<td>Display * &gt; New</td>
</tr>
<tr>
<td>Mail the file “Joe.txt” to “Fred”</td>
<td>Display Joe.txt</td>
</tr>
<tr>
<td>Mail Fred</td>
<td></td>
</tr>
<tr>
<td>Check if the file called “nursery” contains the text “Mary had a little lamb”</td>
<td>Display nursery</td>
</tr>
<tr>
<td>Give the file called “work.doc” the name “play.doc”</td>
<td>List</td>
</tr>
<tr>
<td>Rename work.doc play.doc</td>
<td></td>
</tr>
<tr>
<td>Make a duplicate of the file “old_data.txt”, calling it “new_data.txt”</td>
<td>List</td>
</tr>
<tr>
<td>Rename existing_filename New</td>
<td>Rename existing_filename New</td>
</tr>
</tbody>
</table>
### 18.4. Subject D: 2nd year computing undergraduate

#### Session 1

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put the contents of all files into a file called “New”</td>
<td>*</td>
</tr>
<tr>
<td>Mail the file “Joe.txt” to “Fred”</td>
<td>Mail user</td>
</tr>
<tr>
<td>Check if the file called “nursery” contains the text “Mary had a little lamb”</td>
<td>Find pattern</td>
</tr>
<tr>
<td>Give the file called “work.doc” the name “play.doc”</td>
<td>Rename work.doc play.doc</td>
</tr>
<tr>
<td>Make a duplicate of the file “old_data.txt”, calling it “new_data.txt”</td>
<td>Copy old_data.txt new_data.txt</td>
</tr>
</tbody>
</table>

#### Session 2

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put the contents of all files into a file called “New”</td>
<td>*</td>
</tr>
<tr>
<td>Mail the file “Joe.txt” to “Fred”</td>
<td>Mail user</td>
</tr>
<tr>
<td>Check if the file called “nursery” contains the text “Mary had a little lamb”</td>
<td>Find “Mary had a little lamb”</td>
</tr>
<tr>
<td>Give the file called “work.doc” the name “play.doc”</td>
<td>Rename work.doc play.doc</td>
</tr>
<tr>
<td>Make a duplicate of the file “old_data.txt”, calling it “new_data.txt”</td>
<td>Copy old_data.txt new_data.txt</td>
</tr>
</tbody>
</table>
### 18.5. Subject E: 2nd year physics undergraduate

**Session 1**

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put the contents of all files into a file called “New”</td>
<td>&gt;&gt; New</td>
</tr>
<tr>
<td>Give the file called “work.doc” the name “play.doc”</td>
<td>Rename work.doc play.doc</td>
</tr>
<tr>
<td>Check if the file called “nursery” contains the text “Mary had a little lamb”</td>
<td>Display nursery Find “Mary had a little lamb”</td>
</tr>
<tr>
<td>Mail the file “Joe.txt” to “Fred”</td>
<td>Display Joe.txt Mail Fred</td>
</tr>
<tr>
<td>Make a duplicate of the file “old_data.txt”, calling it “new_data.txt”</td>
<td>Copy old_data.txt new_data.txt</td>
</tr>
</tbody>
</table>

**Session 2**

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put the contents of all files into a file called “New”</td>
<td>List &gt;&gt; New</td>
</tr>
<tr>
<td>Give the file called “work.doc” the name “play.doc”</td>
<td>Rename work.doc play.doc</td>
</tr>
<tr>
<td>Check if the file called “nursery” contains the text “Mary had a little lamb”</td>
<td>Display nursery Find Mary had a little lamb</td>
</tr>
<tr>
<td>Mail the file “Joe.txt” to “Fred”</td>
<td>Display Joe.txt Mail Fred</td>
</tr>
<tr>
<td>Make a duplicate of the file “old_data.txt”, calling it “new_data.txt”</td>
<td>Copy old_data.txt new_data.txt</td>
</tr>
</tbody>
</table>
### Session 1

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mail the file &quot;Joe.txt&quot; to &quot;Fred&quot;</td>
<td>Mail Fred</td>
</tr>
<tr>
<td>Check if the file called &quot;nursery&quot; contains the text &quot;Mary had a little lamb&quot;</td>
<td>Display nursery</td>
</tr>
<tr>
<td>Make a duplicate of the file &quot;old_data.txt&quot;, calling it &quot;new_data.txt&quot;</td>
<td>Copy old_data.txt new_data.txt</td>
</tr>
<tr>
<td>Put the contents of all files into a file called &quot;New&quot;</td>
<td>* &gt; New</td>
</tr>
<tr>
<td>Give the file called &quot;work.doc&quot; the name &quot;play.doc&quot;</td>
<td>Rename work.doc play.doc</td>
</tr>
</tbody>
</table>

### Session 2

<table>
<thead>
<tr>
<th>Task</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check if the file called &quot;nursery&quot; contains the text &quot;Mary had a little lamb&quot;</td>
<td>Display nursery</td>
</tr>
<tr>
<td>Put the contents of all files into a file called &quot;New&quot;</td>
<td>* &gt; New</td>
</tr>
<tr>
<td>Mail the file &quot;Joe.txt&quot; to &quot;Fred&quot;</td>
<td>Joe.txt</td>
</tr>
<tr>
<td>Make a duplicate of the file &quot;old_data.txt&quot;, calling it &quot;new_data.txt&quot;</td>
<td>Copy old_data.txt new_data.txt</td>
</tr>
<tr>
<td>Give the file called &quot;work.doc&quot; the name &quot;play.doc&quot;</td>
<td>Rename work.doc play.doc</td>
</tr>
</tbody>
</table>