Thematic Review of Software Reengineering and Maintenance.

by Andrew Le Gear

Supervisor: _____Jim Buckley

Technical Report

University of Limerick

No: UL-CSIS-04-3

11th March 2004

Contents

List of Figures

Abstract

The Software Architecture Evolution (SAE) group at the University of Limerick in conjunction with their industrial partner are currently researching a variety of software reengineering techniques to recover components from software. Reengineering towards components is the process of extracting cohesive units of reusable code from a legacy system.

The following is a review of existing research in the field of reengineering and maintenance, from which several promising and feasible research directions towards recovering components are discussed. Later in conjunction with our industrial partner, these approaches will be evaluated and a suiteable reengineering approach will be chosen.

Chapter 1

Introduction

Maintenance is viewed as a major crisis in the software industry today (Leintz and Swanson, 1980). With ever expanding volumes of code in companies, the necessity to perform software maintenance is now considered the most serious barrier to the successful application of computer technology (Bowen et al. 1993). It has been suggested that between 70% and 80% of software budgets are allocated to software maintenance tasks (Yourdon 1989) accounting for up to 80% of the effort involved in the development process (Bowen et al. 1993). These figures are projected to grow progressively worse in the future (Yourdon 1989, Rochester and Douglass 1991), however, despite this apparant importance, maintenance has remained, until recently, a neglected area of research (Bowen et al. 1993). In the scenario where the software needs to be maintained to prolong its lifetime, software development teams may have only three choices; purchase a new system, develop a new system or alter the existing system. The third choice is often the only feasible option, since the latter two routes are generally too expensive (Rochester and Douglass 1991).

One partial solution is software reengineering. Several definitions for reengineering exist (Chikofsky and Cross II 1990, Arnold 1993, Corp. 1989). These definitions differ only in emphasis on whether the behaviour of a system can be altered as a result

of applying a reengineering task. For the puporses of this document we will use the Chikofsky and Cross (1990) definition of reengineering:

> "... the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."

This definition does not explicitly exclude alteration of the systems behaviour (Arnold 1993), however it does remain ambiguous. On the other hand, a standard definition for maintenance does exist:

> "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 Std 729-1983

Thus, we could, in fact, view reengineering as an extension of maintenance (Tilley et al., 1994, Leintz and Swanson, 1980). Therefore, maintenance tasks can be said to rely upon our ability to reengineer.

The software reengineering process is described in more detail in chapter 2. The remainder of the report then proceeds to describe how reengineering is performed. Chapter 3 provides us with a detailed description of currently available methodologies that allow us to reason about and perform the various reengineering tasks. This is followed in chapter 4 by a general review of the tool support for reengineering techniques. The final chapter, chapter 5, summarizes the report and suggests research avenues for reengineering's application to component recovery.

Chapter 2

Performing Software Reengineering

2.1 The Software Reengineering Process

Several reengineering models currently exist (Aiken et al. 1993, Stoemer et al. 2003, Gallagher and Lyle 1991, Muller et al. 1993, Zuylen 1993). This document discusses a reengineering model based on the parser-viewer architecture (Chikofsky and Cross II 1990). An adapted version of this is shown in figure 2.1.

Software/ Work/ Product: The input for the reengineering process can vary in this context. Existing software, a complete product or documentation can be taken as input.

Parser/ Semantic Analyser: An analysis of the artefact is performed next. Both syntactic and semantic information can be extracted. Syntactic information can be extracted automatically using a parser. Semantic information, however, will usually involve a human element to achieve useful extraction from software (Biggerstaff 1989, Girard and Koschke 1997, Murphy and Notkin 1997, Murphy et al. 1995, Cimitile and Visaggio 1995, Biggerstaff et al. 1993). The process of design recovery relies heavily upon extracting semantic information from software (see

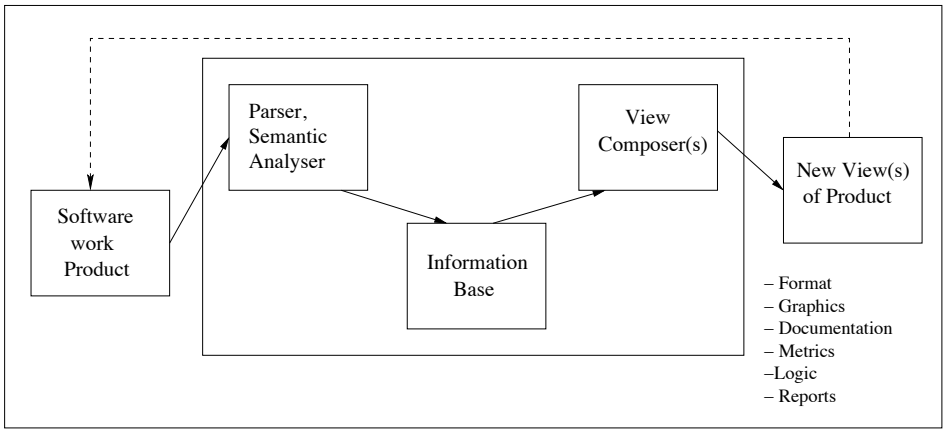

Figure 2.1: Parser-viewer tool architecture adapted from (Chikofsky and Cross II, 1990)

section 3.1.2). Extracting this information requires that the analyst understands the software to some degree (Soloway and Ehrlich 1984, Pennington 1987a). Static or dynamic analysis techniques may be employed and is usually documented afterwards by the user (see section 3.1.3). This topic is discussed further in section 2.1.1.

Information Base: The information extracted during the analysis stage is then stored in a standard format. Several representations exist for storing both syntactic (Larsen and Harrold 1996, Tipp 1995, Ottenstein and Ottenstein 1984) and semantic (Biggerstaff 1989, Biggerstaff et al. 1993, Kosche and Daniel 2003, Murphy and Notkin 1997, Murphy et al. 1995) information. These are discussed later in sections 2.1.2 and 3.1.2.

View Composers: Based upon the information stored in the information base, various representations of the source can be generated to emphasise different aspects of the system. This may include viewing structure generally (Cleary and Exton 2004), call structure (Storey et al. 1997, 2001, Girard and Koschke 1997) or data structure (Zhao 2002, Larsen and Harrold 1996, Cordy et al. 2001) computed directly from the information base. Alternately it may generate views

based upon human provided input regarding relations between software elements (O'Cinneide and Nixon 1999, Schwanke 1991, Murphy et al. 2001). Functionally equivilant but structurally different views of source at the same level of abstraction may also be presented, e.g. - automatically generated "goto" free code (Urschler 1975). The new views generated present an alternate organisation of the software to which it can be reengineered.

New Views of Product: The views composed are outputted to the user for the programmers perusal. Again this may be software, a product or documentation and may be returned as input to the reengineering model for further iterations.

2.1.1 Dynamic vs. Static Analysis Techniques

In order for assertions to be made about a software artifact, some form of analysis must be undertaken. An analysis of the software can derive information such as call relations, data flows or metrics of complexity, some of which may be necessary before other reengineering techniques like clustering and slicing can be performed.

Techniques employed to analyse software can be broadly categorised as static and dynamic (Tipp 1995, Ritsch and Sneed 1993). The difference between the two lies in the distinction between programs and processes[1]. A program is a static representation and is characterized by source code. A process is an instance of that program executing and is dynamic by nature. The scenario is analogous to a recipe and baking a cake (O'Gorman 2001); the recipe being the program and the baking of it being the process. Static analysis therefore will present information based upon the source representation of the system. A dynamic analysis will glean it's information based on source execution at runtime, i.e.-the process.

Deciding which approach to analysis is appropriate is a matter of context. This ar-

[1]This is the operating system notion of a process.

gument is especially relevant when speaking of control or data flow analysis. In the case of a static analysis the resulting data set can be program wide. This can be problematic where programs are large, yielding a massive data set after analysis. One solution would be to perform the same analysis only on the execution paths that are relevant, therefore reducing the data set considerably. However the problem then arises regarding the accuracy of your analysis; how do you ensure that complete branch coverage for the specific context was attained? . Based on our literature review, the majority of analysis techniques used in reengineering tend to be of a static nature.

2.1.2 Internal Representations of Subject Systems

Regardless of the reengineering task, it's final outcome is some how based upon the existing subject software. Therefore any reengineering tool must represent the subject software internally. Slicing in particular (section 3.2.3) relies upon a complete representation of software (Tipp 1995). One common implementation of this internal representation is known as the dependency graph (Ottenstein and Ottenstein 1984, Larsen and Harrold 1996) and exists in many varieties depending upon its use. An early paper from Ottenstein and Ottenstein (1984) describes the *program dependency graph* (PDG) and demonstrates how accurate slices can be computed using this representation. A PDG records both data and control dependencies for a program. As you can see from figures 2.2 and 2.3, even trivial programs can form highly complex program dependency graphs. This is the dependency graph in its simplest form. More complicated dependency graphs must be formed as we introduce tracking for:

- Inter-procedural calls.
- Object oriented paradigms.
  - Inheritance.

```
const
   last = 10;
...
begin
   for i := 1 to last do
      begin
         a[i] := 0;
         b[i] := 0;
      end;

   while not eof do
      begin
         read (code, value);
         read (dummy);
         if code = 'a'' then
            a[value] := a[value] +1;
         else
            b[value] := b[value] – 1;
      end
   writeln (a[last], b[last])
end.
```

Figure 2.2: Example Program (Ottenstein and Ottenstein 1984)

- polymorphism.
- methods and constructors.

System dependency graphs(SDG), *procedure dependency graphs, method dependency graphs*(MDG) *and class dependency graphs*(CDG) are specific variations of the original dependancy graph proposed by Ottenstein and Ottenstein (1984) that may be combined to form adequate solutions to some of these problems. Further details of their use and implementation can be found in (Larsen and Harrold 1996) and (Tipp 1995).

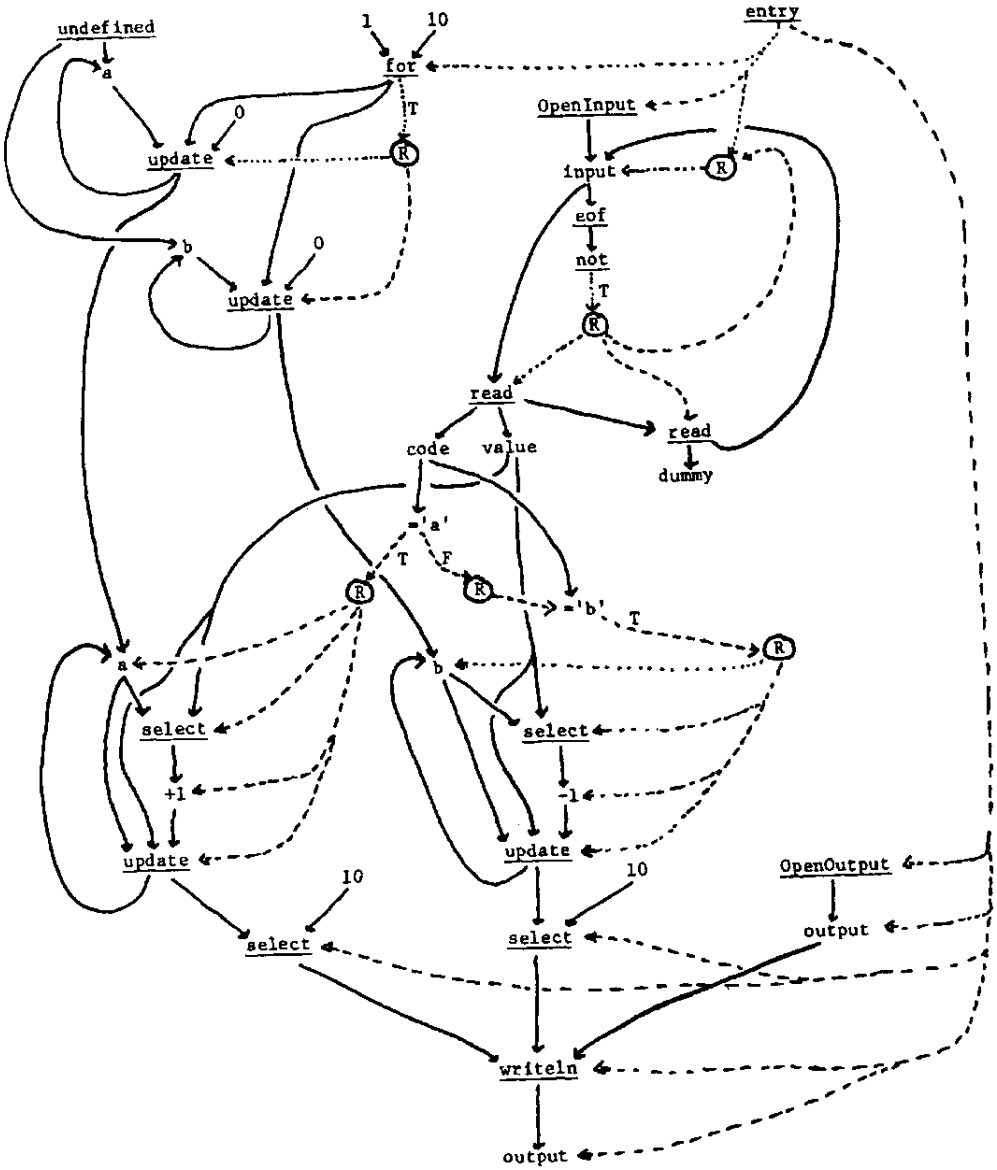

Figure 2.3: PDG for program in figure 2.2 (Ottenstein and Ottenstein 1984)

Chapter 3

Agendas for Software Reengineering

3.1 Design Recovery and Software Comprehension

This section discusses the concerns of design recovery, software comprehension and redocumentation. While the following subsections are devoted to each of these in turn, the reader should be continually aware of their logical grouping and dependence upon each other during reengineering. i.e.- Comprehending software will provide the necessary basis for the recovery of certain design aspects of a system and gaining an insight into recovered design will aid understanding of a system as well as it's successful redocumentation (Chikofsky and Cross II 1990). It is also likely that the greater the software engineers knowledge in these areas is, the greater their ability to maintain and reengineer the subject system in general will be.

3.1.1 Human Comprehension of Software

Some degree of software comprehension will usually take place during a meaningful reengineering of the subject system. Studies have shown that comprehension accounts for up to 50% of the maintenance process and up to 35% of the entire life cycle (O'Brien

and Buckley 2001). The better the understanding a programmer has of the system, the quicker, neater and more error free the maintenance solution will be (Littman et al. 1986).

Initial attempts to probe the understanding process began as early as 1984 when Soloway and Ehrlich (1984) identified the existence of "programming plans" (stereotypic solutions in code) and "rules of discourse" (coding conventions used by programmers that act as communication techniques based on their plans) as related aids to program understanding. They also suggested that, analogous to processing in other technical domains, programmers develop "chunks" as an aid to understanding. Chunks are conceptual abstractions of code into functional units. The existance of these concepts has been exploited by various design recovery tools (see section 3.1.2)(Quilici and Yang 1996, Quilici 1993, Quilici et al. 1997, Woods and Quilici 1996, Rich 1984).

These paradigms became the initial basis for software comprehension theory and have evolved considerably since. Currently accepted models of software comprehension (Mayrhauser and Vans 1995, O'Brien and Buckley 2001) identify meta-models, placing previous theories into a single combined model. These models were based upon the understanding that programmers act as "opportunistic processors" during comprehension (Letovsky 1986), changing their comprehension processes in response to cues that become available as they progress. The diagram in figure 3.1 shows Von Mayrhauser and Van's integrated meta-model (Mayrhauser and Vans 1995).

Only the main concepts of the meta-model are explained here. For a detailed explanation please refer to Mayrhauser and Vans (1995). Firstly, this model combines the **top-down** (Soloway and Ehrlich 1984) and **bottom-up** (Pennington 1987b) approaches to comprehension in a single model. Top-down comprehension uses knowledge of the programming domain to make hypotheses about the code. These hypotheses are iteratively refined until they can be mapped to implementation plans in source code, typically

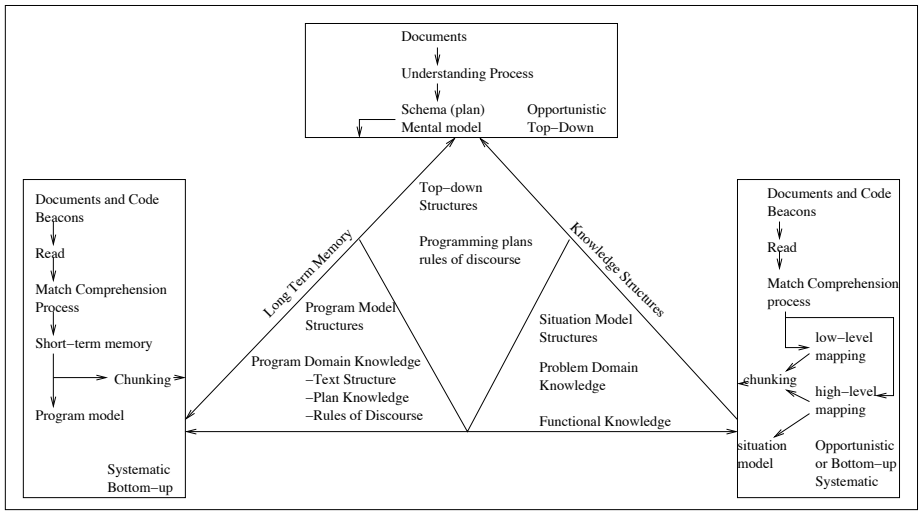

Figure 3.1: Mayrhauser and Van's Meta-Model (Mayrhauser and Vans 1995)

through shallow reasoning. Afterwards these form the basis for more detailed hierarchial units to be generated (the situation model in figure 3.1). In contrast, a bottom-up approach to comprehension will begin at source code level and a programmer will begin to form abstractions by chunking control block based units of code (the program model in figure 3.1). This continues until a single hypothesis regarding the system is reached. As shown in figure 3.1, both approaches are employed during comprehension. Reengineering tool designers have been able to take advantage of the existance of these two comprehension approaches in the Reflexion modelling technique for design recovery (Murphy and Notkin, 1997, Murphy et al., 1995, Kosche and Daniel, 2003, Murphy et al., 2001).

Comprehension processes can be described as either **systematic** or **opportunistic**. Using a systematic approach, the programmer studies the code in detail and gains a deep and comprehensive understanding of the code, which could also include runtime information. The opportunistic programmer will only attempt to understand the absolute minimum in order to achieve his task. Though a systematic approach to understanding will inevitably lead to higher quality understanding (Littman et al. 1986), this is very

often not possible with real large commercial systems.

Apart from approaches to understanding, the internal knowledge structures that programmers form are also described by comprehension literature (Pennington 1987b, Mayrhauser and Vans 1995). **Program Model Structures** describe programming specific knowledge, such as programming plans and rules of discourse. **Situation Model Structures** describe domain specific knowledge. This is a collection of real world knowledge relating to the environment that the system was intended for. **Top-Down structures** describe hypotheses formed using a top down approach to understanding. In particular, Biggerstaff's DESIRE tool (Biggerstaff 1989, Biggerstaff et al. 1993) exploits the existance of these knowledge structures to implement design recovery.

Though the meta-model in figure 3.1 provides for various different comprehension processes it does not explicitly account for differences in understanding encountered by novice and experienced programmers. The divergence between the two groups has been shown experimentally (Soloway and Ehrlich 1984, Good and Brna 2003). Experienced programmers can form far better expectations regarding the subject code and make better use of concepts like rules of discourse (O'Brien and Buckley 2001).

However interesting and useful it is to see how the process of understanding is undertaken by people, better grounding is needed to properly relate the theory to reengineering tasks (Buckley, 1994, Walenstein, 2002, Bannon and Bodker). Design Recovery, a subtask of reverse engineering (Chikofsky and Cross II 1990), better describes this relationship.

3.1.2 Design Recovery and Domain Models

Design recovery[1] is subtly different from comprehension described in section 3.1.1. Chikofsky and Cross (1990) in their taxonomy, define design recovery:

[1] Similar task to *reverse engineering* (Arnold 1993), but probably more accurately viewed as a subset of reverse engineering (Chikofsky and Cross II 1990)

> "Design recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself"

Other descriptions of design recovery do exist (Stoemer et al. 2003, Dean and Chen 2003, Sartipi et al. 2000, Malton and Schneider 2001), but they all essentially capture similar basic concepts. Biggerstaff (1989) first brought the term into the mainstream in 1989 with his accompanying tool DESIRE. Here, the inadequacies of source code alone in an understanding context are identified. Design recovery can include elements of *domain knowledge* regarding the system, the *system's context*, *documentation* supporting the system and input from an *expert* developer of the system.

Core to this topic is the concept of a **domain model**. A domain model records the expectations of a programmer regarding the real-world situation the system is modelling, during an understanding process, and attempts to match these expectations with source code, hence introducing traceability from hypotheses to source code. An attempt at automation was made in Biggerstaff's DESIRE tool (Biggerstaff 1989). The tool is analyzed further in (Biggerstaff et al. 1993) where he identifies what is known as the *concept assignment problem* of matching programming plans, expectations and hypotheses to source code. The concept assignment problem is identified here as a barrier that may not be overcome, at least not in a totally automated manner. Creating domain models automatically has proved difficult (Biggerstaff et al. 1993). Research in the area of plan detection (Quilici 1993, Quilici et al. 1997, Quilici and Yang 1996, Rich 1984, Woods and Quilici 1996), and pattern detection (O'Cinneide 2001, O'Cinneide and Nixon 1999, 2000, 2001, Heuzeroth et al. 2003) , though worthwhile, and partially grounded in comprehension theory, have not yet reached a level of practical application.

At present, the best application for automated design recovery through plan detection would seem to be in vertical domains where a far narrower range of plans and expectations would exist, thus making the solution space manageable (Quilici et al., 1997).

Given that automating design recovery is not currently practical, semi-automated approaches are being investigated as viable solutions. In recent years, semi automated approaches, such as *reflexion modeling,* have been used with very promising results (Kosche and Daniel 2003, Murphy and Notkin 1997, Murphy et al. 1995, Sartipi 2001, Tran et al. 2000, Murphy et al. 2001, Walenstein 2002). These are discussed later in section 3.2.1. Dynamic analysis techniques have also shown promise (Ritsch and Sneed 1993, Heuzeroth et al. 2003, Komondoor and Horwitz 2003, Rajlich and Wilde 2002) , and are discussed in 2.1.1.

3.1.3 Documentation and Redocumentation

Ideally, design and implementation decisions should be recorded and described in documentation. Documentation broadly encompasses architecture and behavioral descriptions in both diagrammatic and prose form and may span all levels of abstraction (Chikofsky and Cross II 1990). Existing documentation can be beneficial during design recovery and comprehension, however maintaining accurate documentation is difficult (Vestdam and Normark 2002) and it can be said that the only truly accurate form of documentation is the source code itself (Mendelzon and Sametinger 1995). Therefore documentation is viewed with caution throughout the software development community.

One suggested approach to improve documentation, Elucidative Programming (Vestdam and Normark 2002), suggests viewing code and documentation from within a web browser. Highlighting code in one panel will display related documentation in another. This provides a direct conceptual link between source and related documentation. Other

approaches that are not immediately obvious as documentation are commenting, coding conventions and style guides employed across organizations (Baecker and Marcus, 1990).

Redocumentation is carried out in situations where the original documentation is lost or has grown too inaccurate to be of use. As the programmer begins to understand the system and recover it's design, this new information is recorded in the form of new documentation. Arnold (1993) defines redocumentation as,

> "... the creation of updated, correct information about software."

The oldest technique for redocumentation is reverse engineering (Chikofsky and Cross II 1990) although this field has now grown much larger making redocumentation a subset of reverse engineering (Chikofsky and Cross II 1990). Redocumentation attempts to recover old documentation or documentation that should have originally existed (Chikofsky and Cross II 1990). The abstractions generated by redocumentation can take either a textual or graphical form (Arnold 1993), examples of which include, pretty printing, diagram generating and cross reference list generating (Baecker and Marcus, 1990).

3.1.4 Software Visualization

Another important aspect to understanding within a reengineering context is how the information gleaned from the subject software is presented to the user. Software visualisation tools typically use graph-based abstractions of source code to facilitate program comprehension. However special care needs to be taken, since a badly displayed graph will be of little use. In Purchase's paper on graph aesthetics (Purchase et al. 2002) she convincingly argues, based on empirical evidence, for several aspects of generic graph layout that facilitate browsing. These include:

- Minimising bends.

- Minimising edge crossings.
- Graph should be arranged orthogonally.
- The graphs width should be minimised.
- All text labels should be horizontal.
- The font should be the same for text throughout the graph.

One graph layout currently being experimented with are "fish-eye"-oriented views (Storey and Muller 1995). This method presents views to the user as if looking at a diagram through a fish eye lens (i.e.- the parts of the graph in context are large while the remainder of the graph is small). This allows the user to view parts of the graph in detail while still keeping track of its context in the remainder of the system. This method of visualization has now been implemented in several tools (Storey and Muller 1995, Storey et al. 1997, 2001) with promising results, though the trend seems to be erring towards a mixture of traditional views and "fish-eye"-oriented views as being the best approach. Storey et al. (1997) have suggested, based upon experimental results, that "fish-eye"-oriented views are better applied to smaller programs. SHRimP (Storey et al., 2001) implements "fish-eye" views and is now incorporated in tandem with the traditional multi-window views of the Rigi[2] tool (Muller and Klashinsky 1988)(see section 3.2).

3.2 Software Restructuring and Refactoring

Software maintenance may be classified into four distinct categories; *corrective*, *adaptive*, *perfective* (*Lientz and Swanson, 1978*) and *preventative maintenance* (Glass and Noiseux, 1981). Of notable interest to reengineering is preventative maintenance, whose

[2]A software reengineering tool developed to present intra and inter hierarchial views of software.

primary aim is to improve the maintainability of the subject software in the future. Restructuring and refactoring are forms of preventative maintenance (Buckley 1994) and are important, semantic preserving (Buckley et al. 2003), reengineering activities. The terms restructuring and refactoring may be unambiguously defined as follows:

Restructuring: This involves performing alterations on code without affecting the systems external behavior(Chikofsky and Cross II, 1990). This process might include modularizing code, removing goto statements or replacing global variables with scoped variables.

Refactoring: The definition for refactoring is very similar to restructuring. Again, the external behavior of the system is unaffected. However, it generally refers to *changes to object-oriented systems* as clarified by Mens et al. (2003):

> "The key idea here is to redistribute classes, variables and methods, in order to facilitate future adaptions and extension."

The following subsections expand upon the topics of restructuring and refactoring techniques and aids.

3.2.1 Clustering

One of the most common techniques used as part of design recovery and software comprehension is clustering. Clustering is a reengineering technique that groups regions of code together based upon their dependencies. It is used to help identify modules, subsystems or components within code as a process of architectural recovery and abstraction, or restructuring. How this analysis is interpreted depends very much upon the clustering method employed. Many approaches to clustering currently exist (Kosche and Daniel 2003, Murphy et al. 1995, Tran et al. 2000, Muller et al. 1993, Girard

and Koschke 1997, Cimitile and Visaggio 1995, Schwanke 1991, Chiricota et al. 2003, Ricca and Tonella 2003). The major distinct approaches are discussed here.

Similarity Measures and Mavericks

Similarity measures attempt to algorithmically identify regions of code that resemble each other. A measure of similarity is calculated by comparing common features[3] in procedures; some clusterings of code are accepted as similar and some rejected by the software engineer making this semi-automatic. The Arch tool in (Schwanke 1991) employs a similarity measure based upon measurement theory from cognitive science. The measure is based on a simple heuristic:

> "if two procedures use several of the same unit-names[4], they are likely to be sharing significant design information, and are good candidates for placing in the same module." (Schwanke 1991)

Arch also implements maverick analysis. This is the opposite to clustering where procedures that do not belong with each other in modules are identified.

Weighted Graph Edges

Weighted graph edges are often used to determine interconnection strength (IS) between procedures or classes. Figure 3.2 illustrates what IS can be. Four call relations exist between procedures A and B while only one relation exists between A and C. Using call relations as a measure of IS we can determine that the IS between A and B is four and the IS between A and C is one.

The resulting weighted graph produced can be seen in figure 3.3. An acceptance threshold (AT) is then chosen to determine what procedures will be clustered. In this

[3] Often the shared variables, classes or functions called. Not to be confused with program features as utilised by (Wilde and Scully, 1995) during software reconnaissance (section 3.3).

[4] Analogous to features.

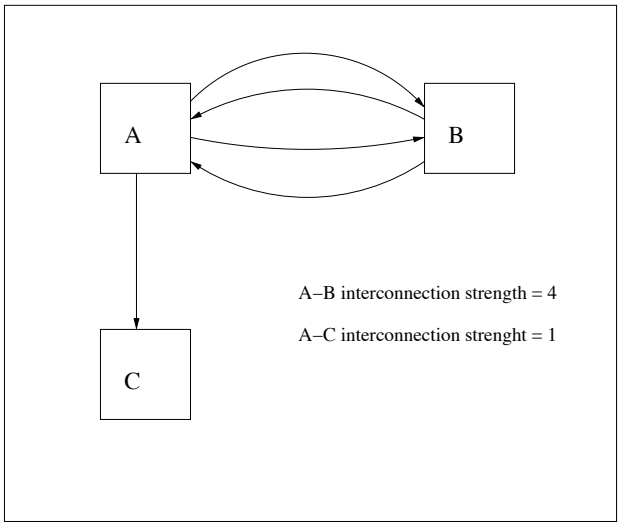

Figure 3.2: Call relations as a measure of interconnection strength

example, if we set our AT to three, A and B would be clustered together and C would be rejected. Examples of IS based clustering can be found in (Chiricota et al. 2003) and (Muller et al. 1993).

Dominance Analysis

Dominance analysis is a concept from graph theory that has been applied to subsystem identification in software. Figure 3.5 helps explain the notion of Strongly Directly Dominant (SDD) and Directly Dominant (DD) relationships. A is SDD over B because it is the last node that you must travel through, from the root node (A), to reach B. A is SDD over C for the same reason. However, A is only DD over D because A is the last node you must travel through, from the root node (A), to reach D, but there still remains more than one route to D from A. We can apply dominance analysis to subsystem clustering if we represent the call structure of a software system in a graph; nodes being procedures and edges being the call relations between them. Mutually recursive relationships (figure 3.4) are first identified and proposed as candidate modules, followed by the construction of a call dominance tree (figure 3.5, B). SDD edges reflect potential

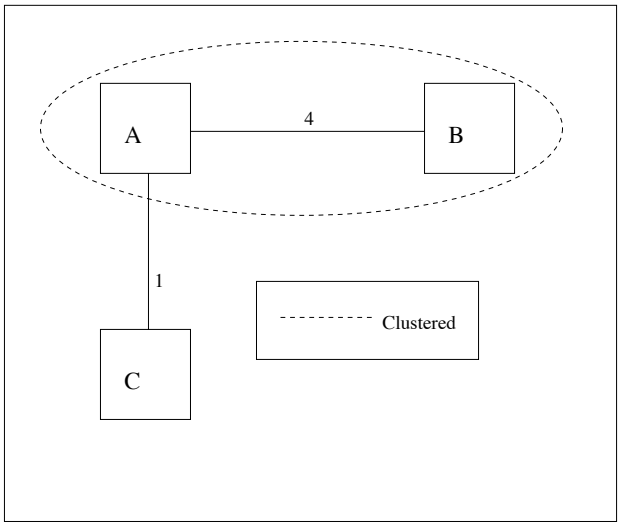

Figure 3.3: Weighted graph

"is-a-component-of" relationships, as A being the only user of B implies that B can be seen as a component of A. This is illustrated in figure 3.5 where we see that modules "B" and "C" offer exclusive sevices to A and therefore may be scoped together. DD edges reflect "uses" relationships. Again this is illustrated by figure 3.5 where see that "D"'s use is not exclusive to any particular procedure, but all clients of D are at a level subordinate or equal to A which imples that the "uses" relation ship can be scoped to A and its subordinates. This process is described in more formal detail in (Cimitile and Visaggio 1995). The engineer is in complete control and may choose to accept or reject the suggestions as he sees fit. A more complete process for dominance analysis is described in (Girard and Koschke 1997).

Artificial Intelligence(AI) Support: Though not a method by itself, other clustering techniques may apply AI methods in order to fine tune acceptance thresholds or similarity measures. Arch (as refered to in section 3.2.1), for example, incorporates a tuning method that over time adjusts its similarity measure to better suit the engineer who uses the system.

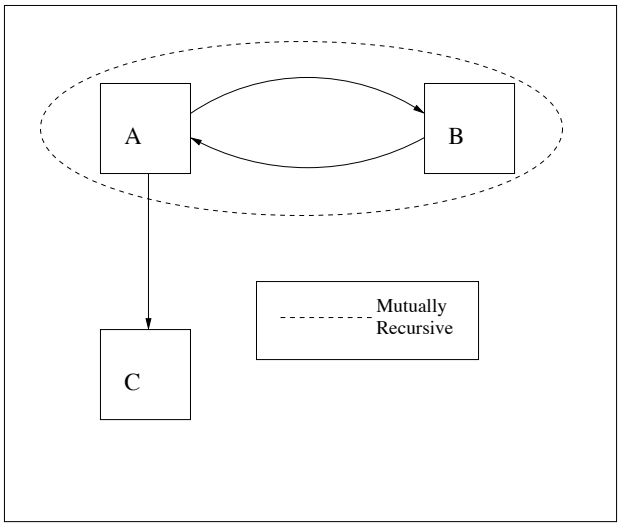

Figure 3.4: Call graph showing and example of a mutually recursive routine.

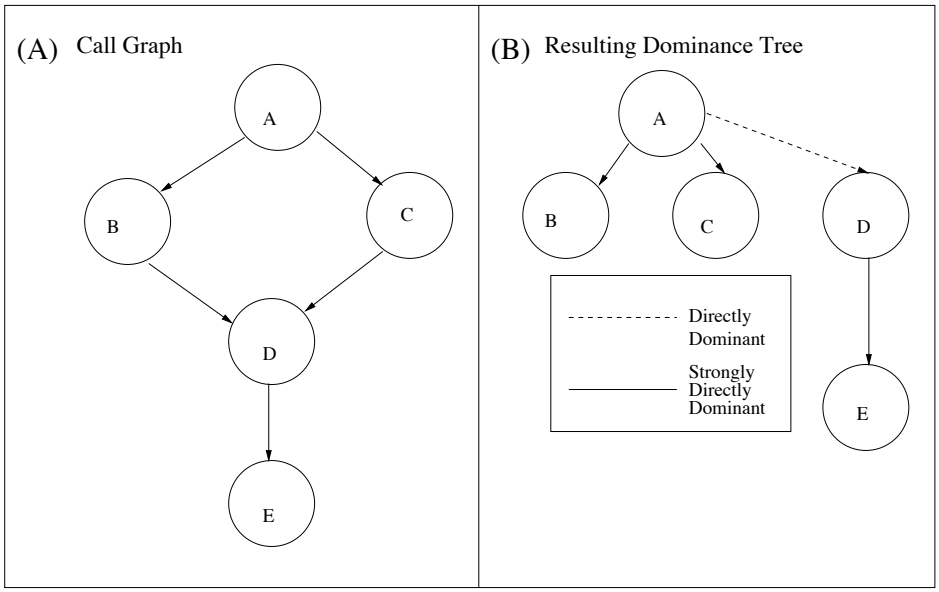

Figure 3.5: Example of dominance analysis.

Reflexion Modeling: Another cluster oriented reengineering method is Gail Murphy's reflexion modeling method (Murphy and Notkin 1997, Murphy et al. 2001, 1995, Kosche and Daniel 2003). This is discussed in further detail in section 3.3.

3.2.2 Control Flow Restructuring

In general, "goto" statements are viewed as bad programming practice (Dijkstra 1968). Control flow restructuring most often refers to the introduction of programming constructs such as "for" and "while" loops[5], replacing code where "goto" statements are present[6].

Urschler (Urschler 1975) provides us with a mathematical approach to control flow restructuring, which is provably correct and requires no regression testing. Structured formalisms can be both introduced or removed using the method. The process is reasonably easy to use, however a description of it's use is beyond the scope of this report.

3.2.3 Data Analysis

Data analysis is concerned with monitoring the states, relationships and integrity of data in software. Many software problems that require maintenance are due to the incorrect representations of data. Take the "millennium bug" problem as a prime example. This problem occurred due to inadequate representations of dates in legacy systems, which would have prevented post-1999 dates from being represented internally. Solving this problem was a priority for many organizations as they approached the millennium and resulted in massive amounts of data reengineering accompanied by the development of new methods, like HSML[7] (Cordy et al. 2001), to achieve the task. HSML is a high-level language used to identify important areas of executable code. This assigned

[5] Also described as structured code.
[6] Also referred to as unstructured code.
[7] Hot Spot Markup Language.

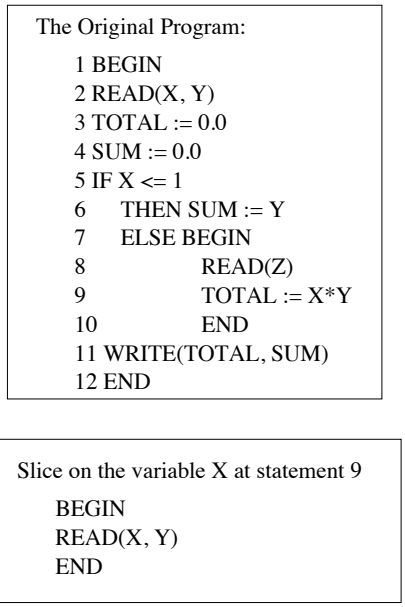

Figure 3.6: A Simple program and a slice on it (Weiser 1982)

importance is user defined and may be refined to yield a focused scope of data, highlighting specific areas to study when addessing this problem.

Slicing

A common form of data flow analysis is *program slicing*. Weiser (1982) identified the concept of slices as a process used by software engineers during debugging as early as 1982. A program slice is an analysis on a given variable, *x,* that will result in the display of only statements involving x, up to a selected point, plus the remaining, necessary statements to form a valid program. The simple example from (Weiser 1982) illustrates slicing in figure 3.6. Here we see a simple program with various operations on variables. The corresponding slice shows only statements relevant to the variable X as far as statement 9 plus other statements to make the slice a valid program, i.e.- "BEGIN" and "END."

This is, of course, a highly trivial example. Algorithms grow increasingly complicated as we attempt to compute slices:

- inter-procedurally.
- on composite data types and pointers.
- in the presence of unstructured control flow.
- with object-oriented languages.
- with aspect-oriented languages.
- using a dynamic analysis approach.

A detailed discussion of solutions and the efficacy of the solutions to these problems can be found in (Tipp 1995). A more specific discussion of slicing for object-oriented and aspect-oriented software can be found in (Larsen and Harrold 1996) and (Zhao 2002) respectively. Slicing approaches have also been applied, with success, to facilitate regression testing in (Gupta 1992) while Gallagher and Lyle take the opposite approach and attempt to eliminate the need for regression testing using decomposition slicing in (Gallagher and Lyle 1991).

3.3 Re-engineering Towards Components

Another useful goal of the developer during reengineering could be the identification of reusable components for export. The focus upon software reuse began as early as the 1970's with the concept of *software parts*[8] being formed in Japan. The practicality of this concept was confirmed by the mid-1980's with reports of reuse in excess of 30% in some development teams (Mii and Takeshita 1993). This demonstrates serious potential for reuse.

[8] Analogous to the modern concept of components.

More recently the concept of component based development processes have emerged (Kruchten, 2000, Cheesman and Daniels, 2001, Allen and Frost, 1998) with the intention of capitalising on the vast potential for reuse as one of its agendas. However, little successful support exists among these processes for either bringing existing legacy systems into the development process or infering components from existing legacy systems to be reused as components within these processes. Many of the techniques discussed in section 3.2 highlight potentially reusable code within software. However, the task of exporting this code, packaging it as a component and integrating it with an existing component based development process is left largely to the intuition of the software engineer. Adapting these techniques to automate or semi-automate this currently manual step may be possible. The following are a few examples that may be particularly relevant.

Applying Dynamic Analysis to the Concept Assignment Problem and Component Identification

Despite the dominance of static analysis based techniques (section 2.1.1 on page 5), certain important advantages of dynamic analysis have emerged in recent times. According to Ritsch and Sneed (1993) the drawbacks of static analysis are that it,

> "...cannot discern between those which are really used and those which are never used."
>
> "...cannot provide any performance information."

and it

> "...cannot relate code to a particular transaction or business function..."

It is this final point that is of most interest. The possibility of inferring business functions from code using dynamic analysis could potentially be of benefit. It's use in procedure identification (Komondoor and Horwitz 2003), and locating program features

and business processes (Wilde and Scully 1995) is already known. This logical step from procedure and business process identification, towards a solution to the concept assignment problem, is not that large. Take, as an example, that the concept[9] "book a flight" in an airline booking system is not functioning to specification. By undertaking an execution trace while performing this task it would be possible to identify a large subset of related code segments for that given business process, therefore matching the concept to it's related code. The software reconnaissance technique employed by Wilde and Scully (1995) attempts to identify programs features by performing traces on executing code, however the agenda here was not component recovery. Related code could potentially be packaged in the form of a component that could be adapted to carry out the specified business process.

Clustering and Reflexion Modeling:

Varying degrees of automation are typically utilised during clustering. At one end of the scale complete automation may be implemented giving the engineer no control. However, software architecture is not an exact science and current approaches contain a human element during decision making (Kruchten 2000, Cheesman and Daniels 2001, Allen and Frost 1998).

Moving along this scale (from automation to semi-automation) we may introduce the engineer into the decision process allowing him to accept or reject candidate clusters, components, modules or packages. This returns control to the engineer and increases accuracy, but the candidature phase still remains in the hands of the technique.

Reflexion modeling is a powerful, semi-automated, technique for domain experts, that involves the engineer at both the candidature and decision stages of the process. Introduced by Murphy et al. (1995), the technique is primarily aimed towards aiding software understanding and architectural recovery, but the technique could equally be

[9]Which will also be analogous to a corresponding use case or business process in many instances.

used to identify components, subsystems and packages in software. The process in figure 3.7 executes as follows:

1. The user, who is a domain expert, defines a *high-level model* model of the system by reviewing available documentation, their knowledge and the source code.

2. A map is then defined matching source elements with entities in the high-level model[10].

3. An extraction tool is applied to the source code to form the *source model*.

4. The reflexion model is then computed by comparing the actual dependencies in the code (the source model) with the map defined by the user (the high-level model). A report is presented graphically to the user describing where his map corresponds to the existing dependencies and where it differs.

5. The hypothesis is that the user, uses this to target inconsistencies and refines his map accordingly. He/She then recomputes the reflexion model.

6. Step 5 iterates repeatedly until the source and high-level models converge.

This software understanding method is radically different to previously proposed approaches and is accompanied by promising results (Murphy and Notkin 1997, Murphy et al. 1995, Kosche and Daniel 2003). One experiment on Microsoft Excel (1 million KLOC+) allowed an engineer to state that he gained a level of understanding of the code within one month that would normally have taken two years (Murphy and Notkin 1997). Another experiment describes gaining a quality understanding of 100KLOC and 500KLOC compilers in 6 and 8 hours respectively (Kosche and Daniel 2003).

[10]This is essentially a form of human executed clustering.

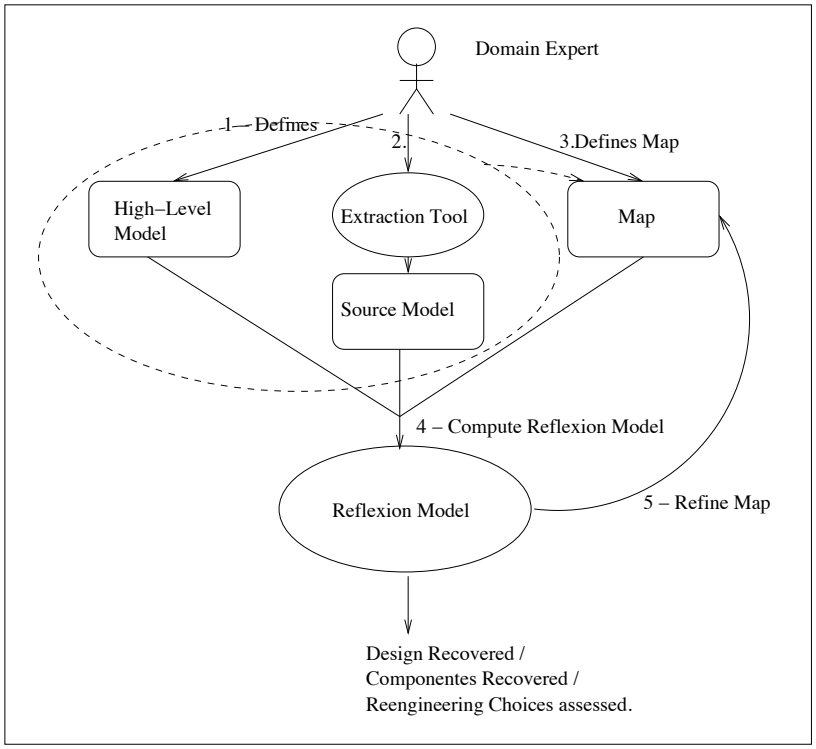

Figure 3.7: Reflexion Modeling

Introducing Design Patterns into Existing Source Code

The design of a software system should grow more flexible as it evolves (O'Cinneide 2001, page 2). Where initially a design pattern was not used, it may be worthwhile introducing a given pattern to code to facilitate software evolution. Mel O'Cinneide provides automatic support for pattern introduction in his DPT tool (O'Cinneide 2001, O'Cinneide and Nixon 2001, 2000, 1999). Automation is provided during the restructuring stage of the process and sucessfully introduces a selected design pattern to existing code when provided with:

- The selected design pattern to be applied.
- The existing components in the system that will be involved in implementing the pattern.

The scenario executes as follows (O'Cinneide 2001, page 3):

1. A software system needs to evolve to accomodate a new requirement.
2. Upon examination the programmer sees that implementing this requirment is difficult given the current design. Introducing a certain design pattern would aid the situation.
3. The programmer selects an appropriate design pattern and the existing components in the software where it should be applied.
4. This information is supplied to the DPT tool and it undertakes the required restructuring and introduction of the design pattern while still maintaining the programs external behaviour.

This work was successfully applied to a large number of common patterns from (Gamma et al. 1995), where considerable reuse of mini-patterns[11] and their associated mini-

[11] A design motif that regularly occurs, but a lower level construct than a pattern (O'Cinneide and Nixon 1999)

transformations[12] was achieved. Design patterns usually reduce coupling within a system (O'Cinneide 2001) hence increasing the likelyhood of cohesive regions of code existing within the software that could possibly be identified as components.

[12] A formal process of infering the behaviour of a minipattern (O'Cinneide and Nixon 1999)

Chapter 4

Tool Support for Software Reengineering

The processes and techniques described in section 2 are implemented by a broad variety of software tools (Sartipi 2001, Heuzeroth et al. 2003, Urschler 1975, O'Cinneide and Nixon 2000, Biggerstaff 1989, Komondoor and Horwitz 2003, Quilici and Yang 1996, Girard and Koschke 1997, Cordy et al. 2001, Schwanke 1991, Storey and Muller 1995, Murphy and Notkin 1997, Storey et al. 1997, Goldman 2000, Chiricota et al. 2003, Gallagher and Lyle 1991, Vestdam and Normark 2002, Wilde and Scully 1995, Linos et al. 1993, Cleary and Exton 2004, Buckley 1994). Some of these are prototype tools used only in experiment, while others are intended for commercial reengineering use. The following subsections describe aspects of functionality and design that are common to many existing tools.

4.1 Preferences and Requirements for Tool Adoption

Apart from being practical and correct in functionality, a tool must also be suited to its users. End user requirements for reengineering tools generally center around how

the reengineering methods provided are visualized in the software, and how applicable the tool is to the reengineering task (Bassil and Keller 2001, Tilley and Huang 2002, Cordy 2003). Bassil and Keller (2001) undertook a comprehensive survey of software visualization tools where they identified user preferences for both their functional and practical aspects. Here a number of tool functions were identified as "absolutely essential":

1. Ability to search for graphical or textual elements represented by the software tool.
2. Visualization of source code (textual views).
3. Hierarchical representations (subsystems, classes, packages, components, control structures).
4. The use of colours.
5. Source code browsing.
6. Navigation across the hierarchies described in 3.
7. Easy access from the symbol list (elements identified in the software), to the corresponding source code.

From a practicality point of view, the following six aspects were identified as being absolutely essential:

1. Tool reliability (absence of bugs).
2. Ease of use of tool (no cumbersome functionality).
3. Ease of visualizing large-scale software.
4. Quality of user interface.

5. Availability of on-line documentation.

6. Content and quality of documentation.

Tilley and Huang (2002) take a higher level approach and identify topics that are usually examined by a company when choosing to adopt a particular tool for a reengineering task. These are:

- Corporate-wide policies, including:
  - Cost of adopting the tool.
  - What hardware and platform is it built for.
  - How well will it integrate with existing software in the company.
- Project Specific Requirements, including:
  - Organization of source code.
  - How does the software and hardware interface.
  - How does the product behave at runtime.

The reengineering techniques discussed in section 2 are all valid and usable techniques. However, getting companies to adopt these processes is not completely straightforward. In Cordy's paper (Cordy 2003), on the barriers to industrial adoption of automated software maintenance techniques, he identifies *business risk* as the single largest factor of influence. For example, bad programming practices,as specified by academics and researchers, like cloning of code, are shown to be common place and preferable to use in industry. Options that reduce cloned sections of code into a single procedure are viewed as high risk since the potential to adversely affect the system is far higher. One of the major challenges of reengineering is to bring it's techniques to industries where business risk is addressed during maintenance.

4.2 Tool Architectures

Software reengineering tool designs can vary. Three major architectural paradigms for software reengineering tools include the *parser-viewer, repository* and *programmable reverse engineering* architectures. The parser-viewer architecture has been implicitly covered in section 2.1. The repository and programmable architectures are discussed here.

Repository Architectures

This architecture was employed by the REDO project (Zuylen 1993, Bowen et al. 1993) in the early 1990's and later commercialized by Piercom Ltd.[1] in their reengineering toolsets. The design centers around storing any parsed information in a common repository of information, which again can be used by any number of view composers. Figure 4.1 illustrates the concept of a repository based architecture. Here we can see several apparant benefits:

- Language independence.
- Views may be generated independent of the subject systems language.
- Parsers may be written independent of previously programmed view composers.
- The modular architecture facilitates extensibility.

A Programmable Reverse Engineering Architecture

Although based upon the repository architecture, the Rigi reengineering tool has evolved over time towards a programmable reverse engineering environment (Tilley et al. 1994). This architecture specifically provides for ease of extension and evolution

[1] Plassy Technological Park, Limerick, Ireland.

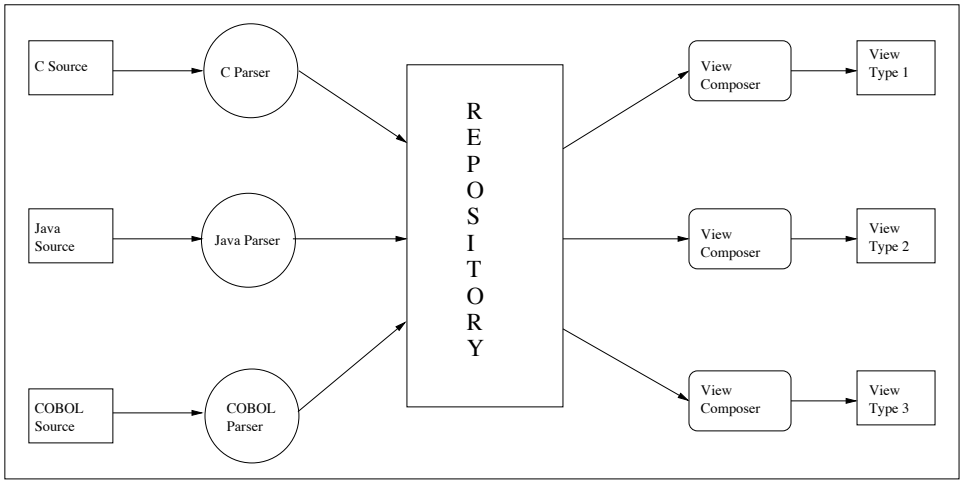

Figure 4.1: The Repository Architecture

of the tool. Figure 4.2 demonstrates the programmable reverse engineering architecture used by Rigi.

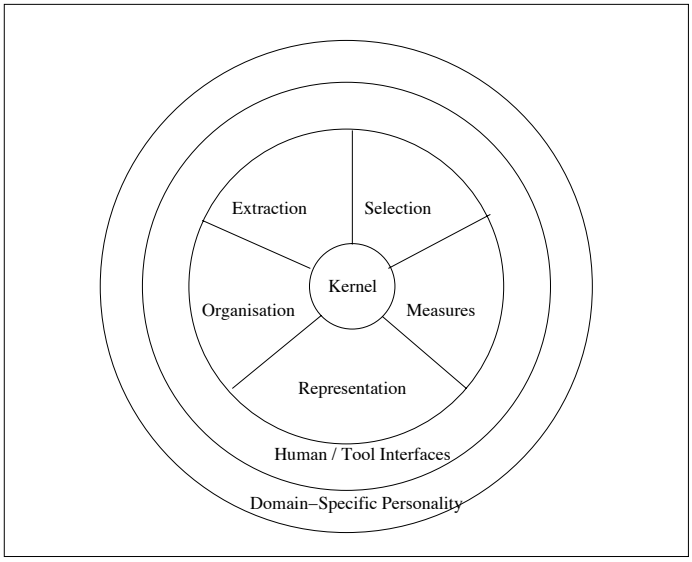

Figure 4.2: The Programmable Architecture of Rigi

Chapter 5

Summary

Software maintenance is now recognized as an unavoidable, large step in the software development process (Bowen et al. 1993). Software reengineering is useful in addressing many of these concerns. This report has comprehensively reviewed the topic of reengineering and its associated techniques. In particular, focus in the review was directed towards the potenial to reengineer towards components in legacy software. Based upon this, three possible research routes were identified:

1. Reflexion modelling (Murphy et al. 1995, 2001, Murphy and Notkin 1997) - a program understanding technique whose basis is grounded in modern software comprehension theory.

2. Applying dynamic analysis to component identification - A method for identifying components in software based upon software reconaissance (Wilde and Scully, 1995).

3. Automatic introduction of design patterns (O'Cinneide and Nixon, 2000, O'Cinneide, 2001, O'Cinneide and Nixon, 2001, 1999).

Based upon discussions with our industrial partner a single method or combination of these methods will be chosen, implemented and empirically evaluated.

Bibliography

Peter Aiken, Alice Muntz, and Russ Richards. A framework for reverse engineering dod legacy systems. In *Working Conference on Reverse Engineering*, pages 180–191, 1993.

Paul Allen and Stuart Frost. *Component-based Development for Enterprise Systems: Applying the SELECT Perspective*. Managing Object Technology Series. Cambridge University Press, February 1998.

Robert S. Arnold. *Software Reengineering*. IEEE Computer Society Press, 1993.

Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs*. Addison-Wesley, 1990.

L. Bannon and S. Bodker. *Designing Interaction: Psychology at the Human-Computer Interface*.

Sarita Bassil and Rudolf K. Keller. Software visualization tools: Survey and analysis. In *International Conference on Program Comprehension*, volume 9, pages 7–17, 2001.

T. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of Working Conference on Reverse Engineering*, pages 27–43, 1993.

Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7): 36–49, 1989.

Jonathan P. Bowen, Peter T. Breuer, and Kevin C. Lano. A compendium of formal techniques for software maintenance. *Software Engineering Journal*, 8(5):253–262, 1993.

Jim Buckley. Dbsum : diagram based software understanding and maintenance. Master's thesis, University of Limerick, 1994.

Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Gunter Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution Research and Practice*, page (to appear), 2003.

John Cheesman and John Daniels. *UML Components : A simple Process for Specifying Component-Based Software*. Component Software Series. Addison Wesley, 2001.

Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, 1990.

Yves Chiricota, Fabien Jourdan, and Guy Melancon. Software components capture using graph clustering. In *International Conference on Program Comprehension*, pages 217–227, 2003.

A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems Software*, 28(2):117–127, 1995.

Brendan Cleary and Chris Exton. Chive - a program source visualisation tool. (to appear), January 2004.

James R. Cordy. Comprehending reality - practical barriers to industrial adoption of

software maintenance automation. In *International Conference on Program Comprehension*, pages 196–205, 2003.

James R. Cordy, Kevin A. Schneider, Thomas Dean, and Andrew J. Malton. Hsml: Design directed source code hot spots. In *International Conference on Program Comprehension*, pages 145–154, 2001.

Guide Int'l Corp. Application reengineering. In *Guide Pub. GPP-208*, Chicago, 1989.

Thomas Dean and Yuling Chen. Design recovery of a two level system. In *International Conference on Program Comprehension*, volume Design Recovery Architecture, pages 23–32, 2003.

E. W. Dijkstra. Go-to statement considered harmful. *Communications of the ACM*, 11: 147–148, 1968.

Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

Erlich Gamma, Richard Helm, Ralph Johnson, and John Vlisides. *Design Patterns: Elements of Reusable Objecto-Oriented Software*. Professional Computing Series. Addison-Wesley, Massachusettes, 1 edition, 1995.

Jean-Francois Girard and Ranier Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *International Conference on Software Maintenance*, pages 58–65, Bari, Italy, 1997.

Glass and Noiseux. *Software Maintenance Guidebook*. Prentice Hall, 1981.

Neil M. Goldman. Smiley - an interactive tool for monitoring inter-module function calls. In *International Conference on Program Comprehension*, pages 109–119, 2000.

Judith Good and Paul Brna. Towards authentic measures of program comprehension. In *Psychology of Programming Interest Group*, pages 29–49, 2003.

Rajiv Gupta. An approach to regression testing using slicing. In *Conference on Software Maintenance*, pages 299–308, 1992.

Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, and Welf Lowe. Automatic design pattern detection. In *International Conference on Program Comprehension*, pages 94–103, 2003.

Raghavan Komondoor and Susan Horwitz. Effective, automatic procedure extraction. In *International Conference on Program Comprehension*, pages 33–42, 2003.

Ranier Kosche and Simon Daniel. Hierarchial reflexion models. In *Working Conference on Reverse Engineering*, November 2003.

Phillipe Kruchten. *The Rational Unified Process: A Introduction*. Addison Wesley, April 2000.

Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *International Conference on Software Engineering*, pages 495–505, 1996.

B.P. Leintz and E.B. Swanson. *Software Maintenance Management, a Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980.

Stanley Letovsky. Cognitive processes in program comprehension. *Empirical Studies of Programmers*, pages 58–79, 1986.

B. P. Lientz and E. B. Swanson. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, June 1978.

P. Linos, P. Aubet, L. Dumas, Y. Helleboid, P. Lejeune, and P. Tulula. Care: an environment for understanding and re-engineering c programs. *Proceedigs of Conference on Software Maintenance*, pages 130–139, September 1993.

David C. Littman, Jeannine Pinto, Letovsky Stanley, and Elliot Soloway. Mental models and software maintenance. In *Empirical Studies of Programmers*, pages 80–98. Ablex Publishing Corpoation, 1986.

Andrew J. Malton and Kevin A. Schneider. Processing software source text in automated design recovery. In *International Conference on Program Comprehension*, pages 127–134, 2001.

A. Von Mayrhauser and A. M. Vans. Program understanding: Models and experiments. *Advances in Computers*, 40:1–36, 1995.

A. Mendelzon and J. Sametinger. Reverse engineering by visualizing and querying. *Software - Concepts and Tools*, 16(4):170–182, 1995.

Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current research and future trends. *Electronic Notes in Theoretical Computer Science*, 82(3):1–17, 2003.

N. Mii and T. Takeshita. Software re-engineering and reuse from a japanese point of view. *Information and Software Technology*, 35(1):45–53, 1993.

Hausi A. Muller and Karl Klashinsky. Rigi - a system for programming in the large. In *International Conference on Software Engineering*, pages 81–86, Singapore, 1988.

Hausi A. Muller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

Gail C. Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 17(2):29–36, 1997.

Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Symposium on the Foundations of Software Engineering*, pages 18–28, Washington D.C., 1995. ACM SIGSOFT.

Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, April 2001.

Michael P. O'Brien and Jim Buckley. Inference-based and expectation-based processing in program comprehension. In *International Conference on Program Comprehension*, volume Program Understanding Comprehension, pages 71–78, 2001.

Mel O'Cinneide. *Automated Application of Design Patterns: A Refactoring Approach*. Phd., University of Dublin, Trinity College, 2001.

Mel O'Cinneide and Paddy Nixon. A methodology for the automated introduction of design patterns. *Proceedings of the International Conference on Software Maintenance*, 1999.

Mel O'Cinneide and Paddy Nixon. Composite refactorings for java programs. *Workshop for formal Techniques for Java Programs, European Conference on Objected-Oriented Programming*, 2000.

Mel O'Cinneide and Paddy Nixon. Automated software evolution towards design patterns. *Proceedings of the International Workshop on the Principles of Software Evolution*, 2001.

John O'Gorman. *Operating Systems with Linux*. Cornerstones of Computing. Palgrave

Publishers Ltd., Houndsmills, Basingstoke, Hampshire, RG21 6SX and 175 Fifth Avenue, New York, NY 10010, 2001.

Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *ACM SIGPLAN/SIGSOFT Symposium on Practical Programming Development Environments*, pages 177–184, Pittsburgh, Pennsylvania, 1984. ACM SIGPLAN /SIGSOFT.

Nancy Pennington. Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop*, pages 100–112. Ablex Publishing Corporation, 1987a.

Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987b.

Helen C. Purchase, Jo-Anne Allder, and David Carrington. Graph layout aesthetics in uml diagrams: User preferences. *Journal of Graph Algorithms and Applications*, 6 (3):255–279, 2002.

Alex Quilici. A hybrid approach to recognizing programming plans. In *IEEE Workshop on Program Comprehension*, pages 96–103, Capri, Italy, 1993.

Alex Quilici, Steven Woods, and Yongjun Zhang. New experiments with a constraint-based approach to program matching. In *Working Conference on Reverse Engineering*, pages 114–123, 1997.

Alex Quilici and Qiang Yang. Applying plan recognition algorithms to program understanding. In *Knowledge-Based Software Engineering Conference*, pages 96–103, Syracuse, NY, USA, 1996.

Vaclav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *International Conference on Program Comprehension*, pages 271–278, 2002.

Filippo Ricca and Paolo Tonella. Using clustering to support migration from static to dynamic web pages. In *International Conference on Program Comprehension*, pages 207–216, 2003.

Charles Rich. Artificial intelligence and software engineering: The programmer's apprentice project. *ACM Annual Conference*, 1984.

H. Ritsch and H. Sneed. Reverse engineering programs via dynamic analysis. In *Proceedings of Working Conference on Reverse Engineering*, pages 192–201, 1993.

Jack B. Rochester and David P. Douglass. Re-engineering existing systems. *I/S Analyser*, 29(10):1–12, October 1991.

Kamran Sartipi. Alborz: A query-based tool for software architecture recovery. In *International Conference on Program Comprehension*, pages 115–116, 2001.

Kamran Sartipi, Kostas Kontogiannis, and Fahrad Mavaddat. A pattern matching framework for software architecture recovery and restructuring. In *International Conference on Program Comprehension*, pages 37–47, 2000.

Robert W. Schwanke. An intelligent tool for re-engineering software modularity. In *International Conference on Software Engineering*, pages 83–92, 1991.

Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984.

Christoph Stoemer, Liam O'Brien, and Chris Verhoef. Moving towards quality attribute driven software architecture reconstruction. In *Working Conference on Reverse Engineering*, 2003.

Margaret-Anne D. Storey, Casey Best, and Jeff Michaud. Shrimp views: An interac-

tive environment for exploring java programs. *International Conference on Program Comprehension*, 9:111–112, 2001.

Margaret-Anne D. Storey and Hausi A. Muller. Manipulating and documenting software structures using shrimp views. In *International Conference in Software Maintenance*, pages 275–285, Nice, France, 1995. IEEE.

Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Muller. Rigi: A visualization environment for reverse engineering. In *International Conference on Software Engineering*, pages 606–607, Berlin - Heidelberg - New York, 1997.

Scott R. Tilley and Shihong Huang. On selecting software visualization tools for program understanding in an industrial context. In *International Conference on Program Comprehension*, pages 285–288, 2002.

Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Muller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

Frank Tipp. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

John B. Tran, Michael W. Godfrey, Eric H.S. Lee, and Richard C. Holt. Architectural repair of open source software. In *International Conference on Program Comprehension*, pages 48–59, 2000.

G. Urschler. Automatic structuring of programs. *IBM Journal of Research and Development*, 19:181–194, 1975.

Thomas Vestdam and Kurt Normark. Aspects of internal program documentation - an elucidative perspective. In *International Conference on Program Comprehension*, pages 43–52, 2002.

Andrew Walenstein. Theory-based analysis of cognitive support in software comprehension tools. *International Workshop on Program Comprehension*, 10:75–84, 2002.

Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

Norman Wilde and Michael C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1): 49–62, 1995.

Steven Woods and Alex Quilici. Some experiments toward understanding how program plan recognition algotithms scale. In *Working Conference on Reverse Engineering*, pages 21–30, Monterey, CA, USA, 1996.

Ed Yourdon. Re-3 part 1. *American Programmer*, 2(4):3–10, April 1989.

Jianjun Zhao. Slicing aspect-oriented software. In *International Conference on Program Comprehension*, pages 251–260, 2002.

H. J. Van Zuylen. *The REDO compendium*. John Wiley and Sons Ltd., Baffins Lane, Chichesterm West Sussex PO19 1UD, England, 1993.

Index