THE SOFTWARE SERVICE BAY: A KNOWLEDGE BASED SOFTWARE MAINTENANCE METHODOLOGY

by

JONATHAN IRVINE MALETIC

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

1995

MAJOR: COMPUTER SCIENCE

Approved by:

____

Adviser Date

____

____

____

Acknowledgements.

I thank my advisor and mentor, Robert G. Reynolds. All my work is due to his help and direction. I would also like to thank my committee, William Grosky, Vaclav Rajlich, and Gregory Bachelis, for their time and help on this document. My parents must be acknowledged for their continued support, dedication, and contributions to my learning.

TABLE OF CONTENTS

Acknowledgements....ii

List of Figures....vi

List of Tables....viii

1. Introduction....1

    1.1. An Environment for Software Maintenance....1

    1.2. Maintenance Methodology Trends....3

    1.3. The Software Service Bay Methodology....4

        1.3.1. Pose Specific Maintenance Problem(s)....6

        1.3.2. Determine Service Rationale or General Maintenance Type....6

        1.3.3. Acquire Specific Domain Knowledge....7

        1.3.4. Develop Maintenance Strategies and Plans....7

        1.3.5. Operationalize Strategies and Plans....8

        1.3.6. Validate Solution....8

    1.4. Service Bay Agents....9

    1.5. Support of Adaptive Maintenance....9

    1.6. Organization....9

2. Relevant Literature on Maintenance Philosophies....11

    2.1. Standalone Software Maintenance Methodologies....12

    2.2. Methods and Technologies for Software Maintenance....14

        2.2.1. Process Oriented Software Maintenance....14

            2.2.1.1. Reuse as Process Oriented Software Maintenance....17

            2.2.1.2. Re-engineering as Process Oriented Software Maintenance....20

        2.2.2. Knowledge Oriented Software Maintenance....21

    2.3. Knowledge Based Maintenance....23

2.4. Summary. 24

3. Realization of a Maintenance Methodology. 25

3.1. A Knowledge Based Maintenance Assistant. 25

3.2. Maintenance as a Problem in Software Reuse. 29

3.3. Problem Solving and the Software Service Bay. 29

3.4. Summary. 31

4. Controlling the Maintenance Process in a Reuse Environment. 33

4.1. Problem Identification. 35

4.2. Determining Maintenance Type. 36

4.3. An Example System Specification 38

4.4. Example Maintenance Problems 41

4.4.1. An Adaptive Maintenance Problem. 41

4.4.2. A Perfective Maintenance Problem. 43

4.5. Suggesting a Problem Solving Approach. 45

5. A Reuse System in the Software Service Bay Framework. 47

5.1. The PM System. 47

5.1.2. PM Agents. 48

5.1.2. Blackboard Controller for the PM System 51

5.2. A Static Reusability Agent 52

5.2.1. Quality Model for Static Reuse 53

5.2.2. Learning in SRC. 54

5.2.3. SRC Results. 56

5.3. A Dynamic Reusability Agent. 58

5.3.1. Chunking in DPA 58

5.3.2. Quality Model for Dynamic Reuse. 61

5.3.3. Maintenance and DPA. 64

5.4. The Task Understanding Agent. 67

5.4.1. Implementing the Rule Learning. 69
5.4.2. Representing the Learning Problem....71
5.4.3. The Learning Loop. 73
5.4.4. A Maintenance Problem in TUNE....73
5.5. The Reuse Library Agent....74
5.5.1. Conceptual Clustering....74
5.5.2. Classification Model. 75
5.5.3. Maintenance and the Reuse Library Agent....77
6. Conclusion and Future Work. 78
Appendix A. System Specification for the Airline Reservation System. 80
Appendix B. Partial Implementation for the Airline Reservation System....91
Appendix C. Pascal Source Code for Simple Genetic Algorithm (SGA)....98
Bibliography. 104
Abstract....112
Autobiographical Statement....113

LIST OF FIGURES

1.1. Maintenance Types and Distribution 2
2.1. Current Maintenance Methods versus Service Bay Methodology 11
2.2. Structured vs. Unstructured Maintenance 12
2.3. Quick-Fix Process Model 15
2.4. Interactive-Enhancement Process Model 16
2.5. Reuse-Oriented Process Model 16
2.6. Re-engineering Process 20
2.7. Black Board Architecture 23
3.1. And/or description of the Software Service Bay. 26
3.2. Software Service Bay Specialists 28
4.1. Process Control of the Software Service Bay 33
4.2. Agents for Problem Solving Selection 34
4.3. Relationship between implementation and specification. 36
4.3. Initial Problem Statement of Example 38
4.4. Summary of Goals for System 39
4.5. Specification Skeleton for Proposed System 39
4.6. System Diagram for Airline System 40
4.7. Type Decomposition for Reservation System 41
4.8. Ada Code Segment for Airline Manager 43
4.9. New Type Decomposition for Reservation System 44
4.10. Additional of Goal for System 44
5.1. PM and the Software Service Bay 48
5.2. PM Reuse Agents 49
5.3. Knowledge contained in blackboard 51
5.4. Quality Model for Static Reusability 53

5.5. An example Decision Tree....................57

5.6. Operationalizing the chunking process'....................61

5.7. Pseudocode for DPA....................63

5.8. Template description of candidate code object....................63

5.9. Structure chart for SGA Program....................64

5.10. Components isolated by the DPA Agent....................66

5.11. Description of candidate code object produced by DPA....................67

5.12. Input file for TUNE....................69

5.13. Output of TUNE....................70

5.14. Version Space Controlled GA....................70

5.15. Representation of the belief space....................71

5.16. Program Segment....................72

5.17. COBWEB Algorithm....................76

LIST OF TABLES

2.1. Reuse Taxonomy 18

5.1. Selection of reusable objects..65

5.2. Facets and values 76

1. Introduction.

Software Maintenance costs are undeniably the major cost in the lifetime of any evolving software system. The cost of maintenance is dependent on the application domain, system age, hardware stability, and development environment [Pressman 92, Sommerville 89]. Normally maintenance costs are between two and four times the costs of development, but can be as high as 130 times development costs [Boehm 75]. Much of the work done to reduce the cost of maintenance is to invest in the production of software development methodologies used to construct large software systems. The key assumption here is if a system is constructed with maintenance in mind, then actual maintenance will be easier and therefore less costly. Even though this work is improving the quality of software and reducing the cost of maintenance, there still exist difficult and costly problems related to maintenance [Pressman 92]. Little work has focused on the actual maintenance phase and how to make it more efficient to do maintenance on, even, a well constructed software system [Lamb 88]. Much of the difficulty in doing maintenance on a software system is that the maintenance is being done in an environment intended for the development, and not for the maintenance of software.

1.1. An Environment for Software Maintenance.

What is Maintenance? Maintenance is a term used to refer to changes made to a software system after the system has been delivered to the user [Martin 83]. Maintenance is performed for a variety of reasons including: correcting errors, design improvements, platform changes, interface changes, database changes, system enhancements, etc. These involve exceedingly different types of problem solving activities. For example, correcting errors (bug fixes) and system enhancements certainly cannot be categorized as the same type of problem solving activity. Even so, both of these activities are viewed as

Software Maintenance. In general, software maintenance problem solving is commonly defined as one of four distinct types [Pressman 92]:

- Corrective
- Perfective
- Adaptive
- Preventive/ Preservation

Corrective maintenance is the correction of error or bugs found in the software system. Perfective Maintenance are those changes to the system designed to improve its current performance. Adaptive maintenance relates to program modifications done in response to changes in the runtime environment of the software system. The terms *perfective*, *adaptive*, and *corrective* were originally used by Swanson [Swanson 76]. *Preventive* maintenance is done to improve the future maintainability of the software system [Pressman 92]. This type of maintenance is also refereed to as *preservation* of the software system [Booch 94], where preservation involves using development resources to improve an aging system that often has a poor design and is therefore hard to maintain.

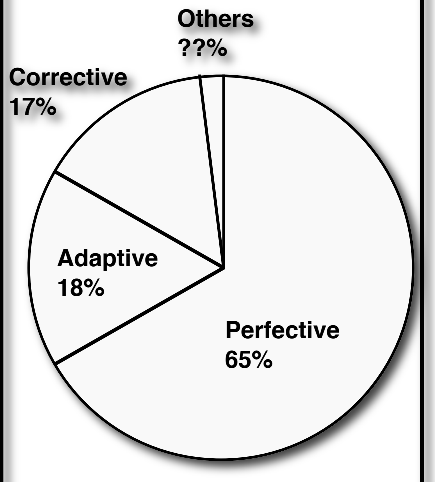


Figure 1.1. Maintenance Types and Distribution.

Of the four types of maintenance, perfective maintenance is cited as the most frequently occurring (see figure 1.1) type of maintenance [Pressman 92, Sommerville 89, Martin 83, Schach 93] and preventive maintenance is rarely done. However, it is assumed that a good software development methodology implicitly deals with this issue. Corrective and adaptive maintenance are done much less frequently than perfective maintenance. This is a particular problem since the maintenance activity is taking place in a software development environment. Therefore corrective maintenance can be done easily in a development environment due to the fact that debugging is a typical part of the development process. Generally many tools exist within a development environment to find and fix bugs. Tools intended to assist in the other types of maintenance are rarely (if ever) included in a development environment.

Each of the four types of maintenance problem solving require different goals, methods, and expertise employed to solve problems within their respective domains. Software Development environments do not reflect these differences in their support of the maintenance process. An environment focused on doing overall maintenance problem solving is needed to help in reducing the costs and problems encountered during maintenance of large software systems.

Before an environment focused on maintenance is described, a methodology must be defined that governs the realization of such an environment. To give perspective to such a methodology, a survey of current maintenance methodologies is presented in the following section.

1.2. Maintenance Methodology Trends.

Much of the literature pertaining to maintenance deals with the design of maintainable software, the cost of maintenance, and the organizational management of the maintenance task [Pressman 92, Sommerville 89, Martin 83, Schach 93, Lamb 88, Bendifalah 87, Littman 87, Veryard 85, Yau 88]. The concept of a comprehensive maintenance problem solving methodology is noticeably absent in the literature dealing with maintenance and software engineering even though the need is apparent [Osborne 90, Schneidewind 87]. The general principle used to deal with the huge maintenance problem is to build more maintainable software. Maintainable software supposedly has qualities that allow it to be modified, understood, and adapted more easily. That is, a software system built with the a priori knowledge that it will be enhanced, adapted, or debugged will be easier to maintain. This view is based on common sense, and this approach has made great strides in reducing the cost of the maintenance of such systems [McClure 76, Boehm 77, Lyons 81].

The current informal frameworks and methodologies for maintenance are described as modifications of widely used software development methodologies [Basili 90, 75] (i.e., waterfall model, spiral model, etc.) or guidelines for the maintenance process given varying amounts of system documentation [Pressman 92]. These frameworks and guidelines still view maintenance as one part of the development process and not as a separate methodology.

The following section introduces a formal methodology that focuses on maintenance as the central principle and not just as one of the development phases of the software life cycle. The methodology proposed in this paper is called the *Software Service Bay* [Maletic 94]. It is motivated by an analogy with the automotive service bay in which there is a different set of tools, problem solving knowledge, and problem solver, for each type of maintenance problem concerning a vehicle.

1.3. The Software Service Bay Methodology.

The Software Service Bay Methodology [Maletic 94] is akin to the concept of the automotive service bay at one's corner automotive service station. A car is taken in for service because of one or another reason, the engine is running poorly, a drive train problem, a brake problem, or routine/preventive maintenance. In the automotive service bay, with each of these maintenance problems goes a special set of tools and a specialized mechanic trained in the particular maintenance problem. The Software Service Bay Methodology works in much the same way. If a software component no longer meets the needs of the user (e.g., specification change, platform change, uncovered bug) then the component is sent in for "service". But, instead of the typical maintenance methodology, which is generally little more then a modified software development methodology, the Service Bay Methodology supports a host of special tools and experts in each of the maintenance types. Doing software maintenance within a software development methodology is analogous to doing automotive service on the shoulder of the highway instead of at the service station. The place where the vehicle broke down generally is not conducive to the support of any maintenance problem solving activities and in general is a harmful environment to do such activities.

The maintenance problem solving process within the Software Service Bay Methodology involves the following phases:

1) Pose specific maintenance problem(s).

2) Determine service rationale or general maintenance type.

3) Acquire specific domain knowledge.

4) Develop maintenance strategies and plans.

5) Implement strategies and plans.

6) Validate solution.

These phases are performed in the order given but may cycle back to a previous phase if necessary, such as when the solution is invalidated by the final phase. Each of these phases are now be described in more detail. These phases are similar to the integrated life cycle model for software maintenance presented by Yau [Yau 88].

1.3.1. Pose Specific Maintenance Problem(s).

The first phase encompasses defining the specific problems needed to be solved within the specific maintenance type. A description of the error, enhancement, platform change, or adaptation is needed for this phase to determine what type of maintenance problem solving is taking place. For example, the particular maintenance problem can possibly be transformed so that it can be solved by some current technology or a special purpose problem solving activity may have to be developed to tackle the problem. The concern in this phase is with the input and output of the software system rather than the internal design and structure of the system. The intent here is to do an analysis of the maintenance problem without dealing with the low level details of the system. The next phase, assessing the current state of the system deals with these design and structure issues relative to operationalizing a solution for the problem.

1.3.2. Determine Service Rationale or General Maintenance Type.

This phase of the Service Bay Methodology involves determining the problem solving rationale behind the service. Just as there is a definite reason why one takes a car in for service, e.g., it stalls at every stop light or the car's ride is very rough, there must be a rationale behind the need for maintenance. This rationale can be defined explicitly or implicitly in terms of the four types of maintenance: corrective, perfective, adaptive, and preventive. A maintenance type provides the focal point for the problem solving process to follow. Each phase is customizable, using domain knowledge, to reflect the type of maintenance to be done there. The service rationale is expressed relative to a maintenance schedule or set of constraints that is established with the generation of the software product. Within that schedule certain generic maintenance tasks are specified and provide temporal landmarks for the realization of other tasks. Identification of new problems can spawn auxiliary schedules for future use.

1.3.3. Acquire Specific Domain Knowledge.

In assessing the current state of the system and its environment, an understanding of the software system's design, structure, and data flow must be acquired relative to the current problem. The depth of knowledge required is dependent upon the rationale for the service and the problem at hand. For example, localizing a bug or determining where program changes for an enhancement will occur connote very different levels of knowledge about the software system. The corrective maintenance problem may require general knowledge of the entire system structure and only a very detailed understanding of maybe one or two routines. However, in a perfective or adaptive maintenance problem detailed knowledge of the entire system is often needed because this type of maintenance often requires many code changes throughout the system.

1.3.4. Develop Maintenance Strategies and Plans.

This phase of the maintenance methodology concerns the selection of a set of general problem solving strategies and plans that address the problems elucidated previously in terms of the current knowledge of the system acquired previously. The maintenance strategies developed here will then be operationalized later on in order to implement specific solutions to the particular maintenance problems at hand. For instance, problem solving strategies are used to suggest possible side effects resulting from the change and specify ways to deal with them. In the case of preventive maintenance, the actual problem to be solved is dictated by the strategies developed. That is, the strategies developed suggest ways to make a system more maintainable or point out areas in the system that are poorly constructed. Plans developed in this phase direct what fixes and modifications must be done to certain portions of the software system and the nature of the systems future maintenance schedule. These plans include "canned" fixes or routines known to fit the change specification. This type of planning knowledge and information are retrieved from a software reuse library.

1.3.5. Operationalize Strategies and Plans.

Once the maintenance problem solving strategies and corresponding plan are selected, they are implemented, and the changes and modification dictated by them are made to the software system. The plans may involve code changes, code rewrites, or the creation of new code however, they are not limited to modification of code. Changes also occur to the corresponding documentation in order to reflect the modification of the code. This step in the methodology is analogous to the auto mechanic doing the actual work on the vehicle (i.e., get their hands greasy). All of the canned fixes suggested by the previous phase is implemented in this phase. Therefore, a reuse system that allows easy access and indexing of such solutions is needed.

1.3.6. Validate Solution.

The last phase of this methodology involves the validation of the solution. Validation of these changes and modification are made with the knowledge gleaned from the previous phases and any testing history of the system and associated documentation being maintained. The testing of a modified system only entails testing part of the system versus the entire system. A modification may only effect a subset of the entire system. Knowledge of the modifications and where they occurred in the system are useful information in this phase. This type of knowledge is useful to the vehicle mechanic in the same way. For instance, if a new alternator (generator) is installed into a vehicle, the mechanic would check to see if the battery is being charged correctly. And the mechanic will not test the transmission in this case because the alternator has nothing to do with the transmission.

1.4. Service Bay Agents.

Each of the problem solving phases within the Service Bay Methodology is supported by one or more specialists and agents. These specialists and agents are experts in one particular area of the maintenance problem. The Service Bay specialists and agents are functionally independent but work in conjunction with one another in an opportunistic fashion. This topic will be discussed in more detail in section 3.

1.5. Support of Adaptive Maintenance.

To illustrate how the Software Service Bay Methodology can enhance the maintenance problem solving process, an adaptive maintenance problem will be expressed and solved within the Service Bay methodology. A model of how adaptive maintenance problems are handled in the Service Bay will be presented, and a particular real world adaptive maintenance problem will be expressed in this model.

1.6. Organization.

The first chapter presented a general methodology for software maintenance, i.e., The Software Service Bay. Chapter 2 reviews relevant literature in software maintenance methodologies and the tasks associated with the maintenance process. Chapter 3 describes a prototype knowledge based problem solver to deal with comprehensive maintenance activities. Chapter 4 describes the control structure of the Software Service Bay. An example software system is presented and maintenance problem are suggested to demonstrate the issues involved. Chapter 5 presents a system for software reuse using the Software Service Bay methodology. Examples of solving maintenance problems with the system are also presented in chapter 5. Conclusions and suggestions for future research on knowledge based maintenance methodologies are given in chapter 6.

2. Relevant Literature on Maintenance Philosophies.

Many of the current software development methodologies (i.e., waterfall model, spiral model, cleanroom engineering [Mills 93] etc.) depict maintenance as a single step at the end of a life cycle model [Schneidewind 87]. This view leaves little room to address the complexities associated with maintenance. In fact, maintenance is much more akin to starting over from the beginning and re-developing a system with different requirements. With this in mind, a methodology centered on maintenance and not on development is possible. The make up of such a methodology will now be discussed and literature relevant to developing such a methodology will be reviewed.

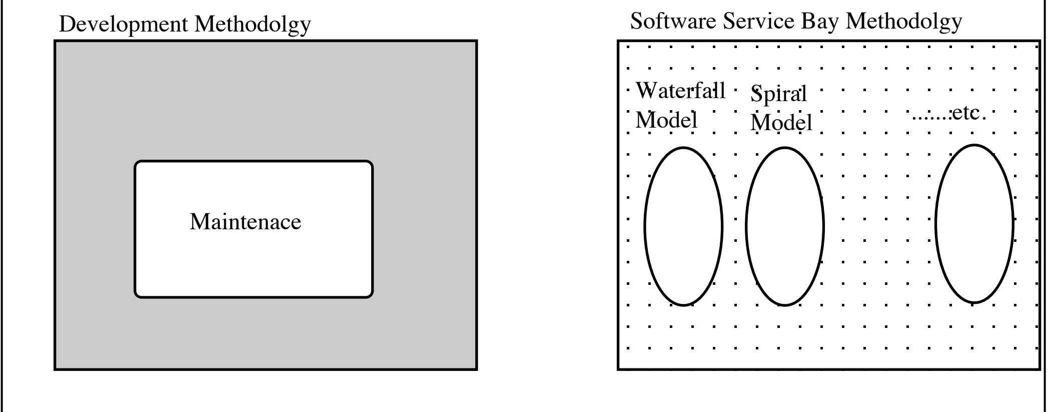


Figure 2.1. Current Maintenance Methods versus Service Bay Methodology

There are two basic approaches currently taken to software maintenance. First, it is viewed as a stand alone process independent of a particular development process. Second, it is viewed as a special case of another existing methodology. The methodologies are divided into two categories, process driven and knowledge driven. Each of the above approaches are discussed in the following sections.

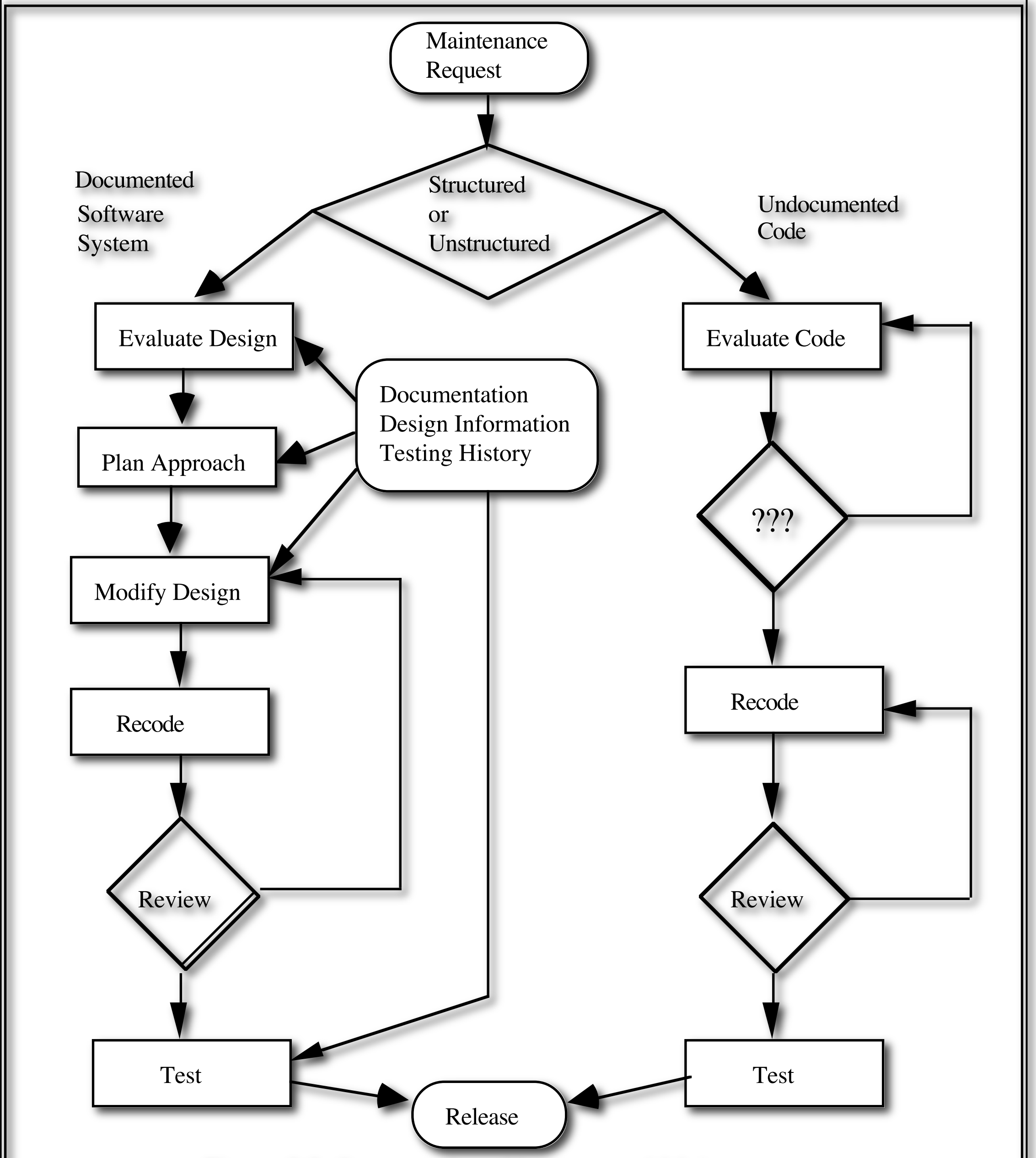


Figure 2.2. Structured vs. Unstructured Maintenance. [Pressman 92]

2.1. Standalone Software Maintenance Methodologies.

A maintenance methodology is defined by the types of maintenance that are done (corrective, perfective, adaptive, preventive) and the level of abstraction at which the maintenance problem is being posed (e.g., logical level, requirement specification level, implementation code level, etc.). How the maintenance methodology views the maintenance process is also a key issue e.g., is maintenance viewed as a final step in a development life cycle, or as a complete re-engineering of an existing system?

The Software Service Bay Maintenance methodology is unique in that it is intended to be independent of the life cycle, or development methodology that originally produced the system in the first place. Other maintenance philosophies, described in the literature, are tightly coupled with some development methodology (see figure 2.1). For example, maintenance is often viewed as the last phase of the Waterfall model or another cycle (or more) in a Spiral model. Because of the Service Bay's model is independent of any development methodology there must be explicit reference to the issues of maintenance i.e., the six phases of the Service Bay model.

One of the few general, if informal, frameworks for maintenance is described as *structured maintenance* [Pressman 92]. If a complete software configuration exists (i.e., documentation, specification, testing history, etc.) then the design documentation is first evaluated. Issues such as structure, performance, and interface characteristics of the system are determined. The impact of the modifications or corrections are assessed and an approach to solve the problem of implementing the modifications is planned. The design is then modified, to reflect the planned approach, and then reviewed. New source code is developed and then tested with respect to known testing specifications. This sequence of events constitutes the structured maintenance process and can occur when accurate and up to date documentation are available. If little or no system documentation is available to the maintainer then maintenance must take place in a more ad hoc fashion. Pressman refers to this as *unstructured maintenance*. Figure 2.2 gives a flowchart representation of the structured maintenance model and a possible scenario for unstructured maintenance.

Structured maintenance is predicated on the availability of appropriate system documentation. It is implicitly a knowledge based approach but the specific ways in which problem solving knowledge is used to tackle a problem in a given environment is not explicitly discussed. The absence of such documentation precludes the use of this approach to the maintenance of any large scale system.

2.2. Methods and Technologies for Software Maintenance.

The process of doing maintenance on a system results in a new version of the original system. This new version is a modification of the old system or an entirely new system that reuses much of the old system's components and structure [Basili 90]. Thus, some maintenance can be viewed as a problem in software reuse [Basili 90]. Likewise other specific maintenance activities are easily expressed as a type of software re-engineering [Green 91].

This section is concerned with identifying technologies (methods, philosophies, frameworks) that are used in solving special cases of the maintenance problem. That is, certain maintenance problems can be re-expressed as a problem in another paradigm for which there are existing technologies. These technologies include: software reuse, knowledge based software engineering, reverse engineering, and re-engineering of software systems. What specific maintenance activities these technologies support and how these technologies support the Software Service Bay methodology are now outlined.

2.2.1. Process Oriented Software Maintenance.

[Basili 90] describes three maintenance process models:

- the quick fix model,
- the iterative-enhancement model [Basili 75],

- the reuse-oriented model.

All three models express or transform specific maintenance situations in terms of other existing methodologies (see figures 2.3,2.4,2.5). All of these models assume that there exists a complete and consistent set of documents for the original system, and therefore are classified as examples of structured maintenance.

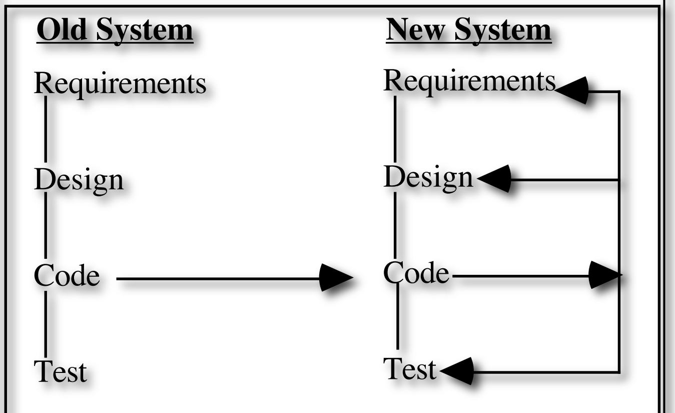


Figure 2.3. Quick-Fix process model [Basili 90]

In the quick-fix model (see figure 2.3), the original system, normally just the code, is modified to satisfy the maintenance objectives. The new system is then tested and the documentation is modified in order to reflect the changes made to the code. The quick-fix model is rooted in the waterfall development methodology. A change at any one level requires a reassessment of the other levels. This model is most appropriate for corrective and simple adaptive maintenance where an error correction or enhancement is needed in only one component or a simple structural change is required.

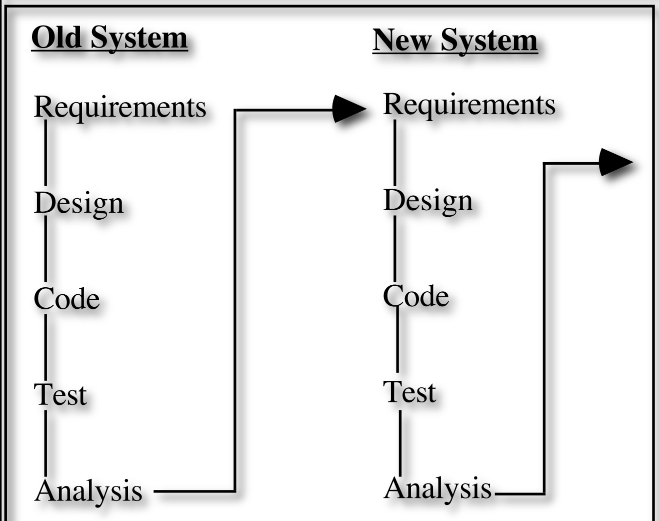


Figure 2.4. Interactive-enhancment process model [Basili 90]

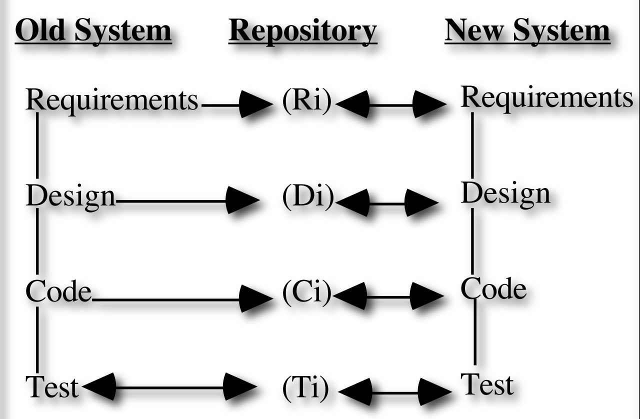


Figure 2.5. Reuse-Oriented process model [Basili 90]

The iterative enhancement model [Basili 75] was originally proposed as a development model. In the case of maintenance, the iterative enhancement model starts with the original system's requirements, design, code, testing information and documents. An analysis of the original system takes place in order to determine what type of modifications are needed. The system is then modified starting at the requirements level and then these changes are propagated down to the next level in the system development phase. Each step in the process allows the redesign of the system based on the analysis of the original system. This model is basically a restatement of a prototype development methodology. The original system is the prototype and the system resulting from the maintenance process is the new system. The iterative enhancement model is most appropriate for perfective maintenance, where the new system's requirements have not substantially changed in relation to the original system.

Maintenance can also be transformed into a type of software reuse process [Basili 90]. In this framework, the requirements for the new system guide the development process, reusing as much of the old system as possible (see figure 2.5). System documents and information for the new system are constructed from the old system and any other documents from the repository (reuse library). New system components and documents will be developed as needed from scratch. This model supports a wider variety of maintenance activities. Corrective, perfective, and adaptive maintenance are all supported by the reuse-oriented maintenance model.

2.2.1.1. Reuse as Process Oriented Software Maintenance.

Software reuse is the process of using existing software artifacts to build new systems rather than building them from scratch [Krueger 92]. Biggerstaff and Richter suggest that there are four fundamental subtasks associated with the overall task of operationalizing the reuse process [Biggerstaff 87]: finding reusable components, understanding these components, modifying these components, and composing components. Reuse normally involves these four steps though certain techniques may focus on certain steps and de-emphasize other steps.

Certain types of software maintenance can be viewed of as a form of reuse with replacement, in which new variants of an entire systems are created and then substituted for the original systems [Barnes 91]. Software reuse is at least as applicable to the maintenance process as it is to the development process. All maintenance types benefit from reuse at one or more levels of abstraction. Currently the relationship between reuse and maintenance is specified informally [Prieto-Diaz 93]. Current research in software reuse includes work on classification and retrieval of software components [Prieto-Diaz 85, Prieto-Diaz and Freeman 87, Horowitz 84, Lee 93], frameworks for reuse [Biggerstaff 87, Prieto-Diaz 87, Basili 88, Basili 91], and automatic acquisition of software engineering knowledge [Reynolds and Maletic 90].

Substance	**Scope**	**Mode**	**Technique**	**Intention**	**Product**
Ideas, concepts	Vertical	Planned, systematic	Compositional	Black-Box, as-is	Source code
					Design
Artifacts, components	Horizontal	Ad-hoc, oportunistic	Generative	White-box, modified	Specifications
Procedures, skills					Objects
					Text
					Architectures

Table 2.1. Taxonomy of Reuse

Table 2.1. is a taxonomy of software reuse approaches [Prieto-Diaz 93]. There are at least six characteristic perspectives for viewing software reuse: substance, scope, mode, technique, intention, and product. Substance relates to what is being reused, this is the first characteristic listed in table 2.1. Reuse of ideas, artifacts, or procedures determines the substance. The reuse of ideas deals with reusing general solutions to problems, such as generic algorithms. An example of Artifact reuse is Booch's Ada components [Booch

87]. Procedural reuse is focused on the formalization and encapsulation of software procedures and data. Collections of reusable modules can be connected to instantiate new and more complex systems.

The scope of the reuse is classified as either vertical or horizontal. Vertical reuse deals with reuse within the same domain or application area. Horizontal reuse is where generic parts of different application are used together to build systems. The mode of reuse refers to whether the reuse of a system is a built-in or planned-for practice versus an unplanned-for ad-hoc practice. Compositional and generative refer to the reuse technique or level of where the reuse is taking place. Compositional reuse uses existing components as building blocks for the new systems. Generative reuse is focused on reuse at the specification level. The intent of the reuse refers to how the components from the reuse library are used. Black-box reuse is the use of components from the reuse library without any modification to the reused component. White-box reuse allows for some modification of the reused components, such as the case with templates. The last characteristic in table 2.1 deals with the product to be reused. The product being reused may be source code components or it may be design architectures.

This taxonomy suggests the types of maintenance that the reuse process can help support. The product column relates to what type of knowledge is available to the maintainer. Each of the maintenance activities use the products of reuse to different degrees. For example, corrective maintenance uses only source code and design products, while perfective maintenance uses specifications, objects, design, and source code products. Software reuse technologies and specialists are an important part of the Service Bay methodology. The phases of the Service Bay methodology dealing with developing maintenance strategies and implementing these strategies assume the existence of a knowledge base of software engineering knowledge; much of this knowledge is embodied in a reuse library.

2.2.1.2. Re-engineering as Process Oriented Software Maintenance.

Re-engineering is a combination of forward engineering (software development) and reverse engineering (design recovery) [Alagappan 91]. The system specifications for a new system resulting from maintenance are constructed from integrating new system requirements with the desirable system specifications of the old system. Figure 2.6. describes a re-engineering process. The reverse engineering part of the re-engineering process relates to automatic program understanding, data-design recovery, and program analysis research.

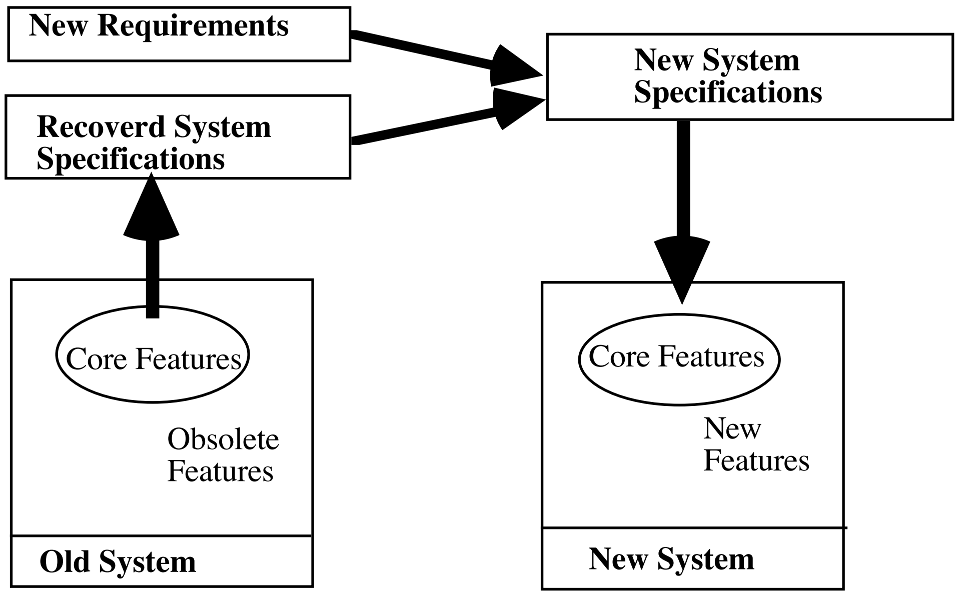


Figure 2.6. Re-engineering Process [Alagappan 91]

These concepts generally base the forward engineering process on a transformational approach [Barr 82, Johnson 91]. Transformations are a means of incrementally developing a specification. The transformations allow a developer to gradually refine, synthesize, and elaborate the specification until the required system behavior is achieved. This transformational approach was originally developed for forward engineering systems, though the technology is also applicable to reverse engineering applications [Letovsky 88, Johnson 91]. A model of the stepwise refinement process [Reynolds, Maletic 92] based on AI problem solving methods has been developed and relates to the transformations approach.

Re-engineering is design (specification) level reuse oriented model for maintenance. The types of maintenance activities directly supported by re-engineering are perfective and adaptive. This maintenance model does not support any type of corrective maintenance since it is rarely necessary to recover the system specifications for bug fixes.

2.2.2. Knowledge Oriented Software Maintenance.

The term Knowledge Based Software Engineering (KBSE) encompasses a wide range of Artificial Intelligence tools and techniques applied to the software development process. Some of these topics include automatic program design [Smith 90, Rich 88], automatic program understanding [Hartman 91, Harandi 90, Hausler 90, Abd-El-Hafiz 94], re-engineering [Chikofsky 90, Hall 92], and reverse engineering [Chikofsky 90, Rugaber 90]. The overall objective of KBSE is to provide AI based assistance for all parts of the software life cycle [Lowry 90, Rich 88]. More specifically KBSE research is focused on the following topics:

- To formalize the specification languages for precise and unambiguous specifications of the software systems intended to be built or maintained [Schonberg 81, Green 82, Bailes 93].

- To automatically record, organize and retrieve the knowledge behind the design decisions within a software system [Johnson 85, Hartman 91, Harandi 90, Hausler 90, Rist 92].
- To generate knowledge based software systems to assist in the synthesis, validation, and understanding of source code from formal specifications[Rich 90, Mana 86, Darlington 81, Barstow 79, Kant 83, Cheatham 84, Smith 93].
- To produce knowledge based software systems to develop and validate specifications [Good 84, Boyer 84, Hall 93, Zeil 93].

The above goals can be used to support maintenance activities in several ways. First, the formal specification of a system makes enhancement and adaptation of the system a well regulated process. Knowledge of the design information of the system facilitates maintenance by avoiding the current methods which reconstruct design decisions from the source code and written documentation. Knowledge based assistants for synthesizing and validating source code allow maintenance to be done at a specification level rather than a source code level. Therefore, KBSE technologies support, in some way, the corrective, perfective, and adaptive maintenance activities. Though, no one technology (automatic program design, automatic program understanding, etc.) supports all of these maintenance activities. Preventive maintenance is also supported to some degree with KBSE technologies. Via understanding and design recovery systems, a overview of a system could be attained. Given a set of design guidelines the systems architecture is then evaluated and suggestions for restructuring of the system made.

KBSE technologies fit well into the Software Service Bay methodology. The phases concerned with assessing the current state of the system, development of the maintenance strategies, implementation of these strategies, and validation of the solution are all addressed in some way by the KBSE approach. Much of the research in this field addresses the issues of maintenance in at least a limited way [Alagappan 91, Johnson 91, Smith 90], though no system is dedicated to the problems of maintenance.

2.3. Knowledge Based Maintenance.

The Software Service Bay Methodology supports a flexible Knowledge Based Maintenance environment that allows the addition of new automated (or semi-automated) service specialists as the need arises or technology changes. The Software Service Bay can be supported by a Blackboard Architecture.

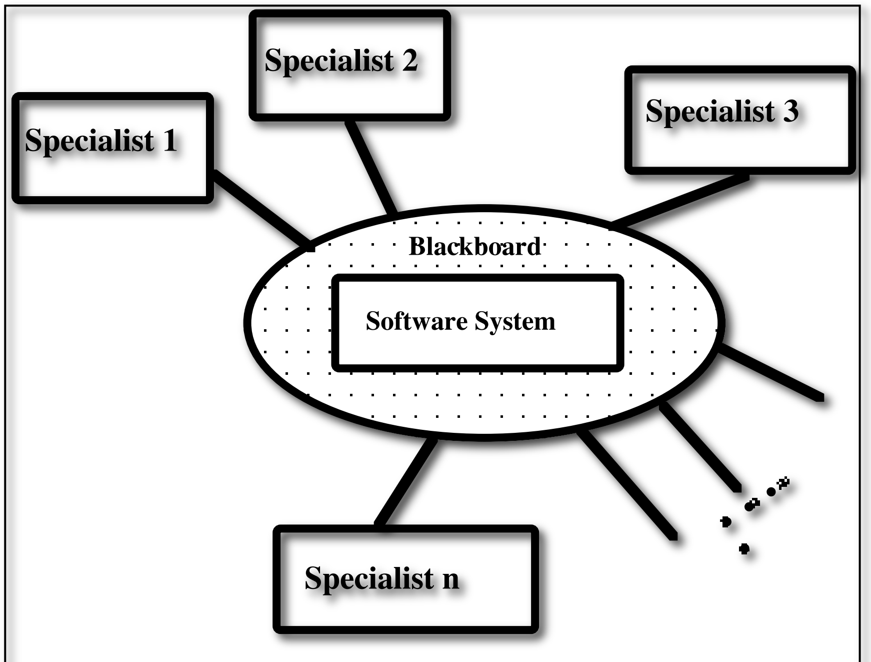


Figure 2.7. Blackboard Architecture of Software Service Bay Methodology

The *Blackboard Architecture* [Reddy 76, Luger 89, Firebaugh 88] is composed of a blackboard and a set of knowledge sources. The blackboard is a central global data base that acts as a clearing house of information for the independent knowledge sources. Each knowledge source (KS) acts as an autonomous production system or expert for a particular part of the problem at hand. The knowledge sources act anonymously on the information which it finds on the blackboard and may write to the blackboard accordingly. The KS's continually examine the information posted on the blackboard. Whenever information on the blackboard satisfies a condition within a KS, a production rule is fired which may write to the blackboard. Thus the blackboard architecture acts as an asynchronous set of demons which process information from a central global database into a meaningful process.

In the case of the Software Service Bay the blackboard is the software system (source code, documentation, specifications, etc.) and the knowledge sources or experts are the Software Service Bay Specialists. As in the Blackboard Architecture, the Service Bay Specialists operate independently of each other. The software system acts as the global data base which is readable by each of the Specialists. Each Service Bay specialist can modify the system accordingly via a modification specialist. This Modification Specialist acts as the scheduler in the Blackboard System. Thus, the Modification Specialist handles the communication between any Specialists and supplies direction to the problem solving process. Figure 2.7. displays the software system as the blackboard and the Service Bay specialists as the independent knowledge sources.

2.4. Summary.

Current methods for attacking maintenance problems are little more then guidelines and idioms for specific maintenance situations. There does not exist a general purpose umbrella, methodology, or idiom, which covers all maintenance situations or maintenance types. There do exist special purpose tools and technologies (e.g., reverse engineering, software reuse, etc.) for solving specific maintenance situations but there is no methodology to use these tools in conjunction with one another. These approaches transform specific maintenance problems into other existing process oriented or knowledge oriented approaches. The Software Service Bay proposes a knowledge based methodology that acts as a general umbrella under which all maintenance situations are covered. The problem solving process is driven by collected knowledge of the problem, problem solving techniques, and current domain knowledge.

3. Realization of a Maintenance Methodology.

In section one, a methodology for maintenance problem solving, The Software Service Bay is presented. It is designed to support that performs in all types of maintenance activities from perfective to adaptive. This method for doing maintenance is different from current methods for maintenance in that the Service Bay is not embedded in a particular software development methodology whereas most current methods are. In order to implement the Service Bay approach, a number of tools and technologies must be integrated together to support some of the diverse problem solving activities done in maintenance. Section two describes how many maintenance situations can be reformulated into problems that are solved by other existing tools and technologies. The next step is to operationalize the Software Service Bay Methodology in terms of problem transformation and reformulation. The general problem solving mechanism is augmented by a special purpose problem solver. Each special purpose problem solver has a set of preconditions that must be satisfied, by a particular problem, for it to be activated. In this manner traditional problem solving approaches are integrated into the maintenance problem solving process.

3.1. A Knowledge Based Maintenance Assistant.

The maintenance problem solving process within the Software Service Bay Methodology involves the following phases:

1) Pose the specific maintenance problem(s).
2) Determine the maintenance type for the problem. Within that maintenance type can the problem be transformed into a problem that can be solved by an existing specialist. If so, identify the special purpose problem solver that is appropriate to the original problem

3) The selected problem solver acquires specific domain knowledge needed to solve the problem.
4) A maintenance problem solving strategy is developed by the system.
5) Implement problem solving strategies and plans.
6) Validate solution.

These phases are now operationalized (see figure 3.1) with reference to transforming particular maintenance problems into problems that can be solved opportunistically in terms of available tools and technologies associated with known problem solving situations.

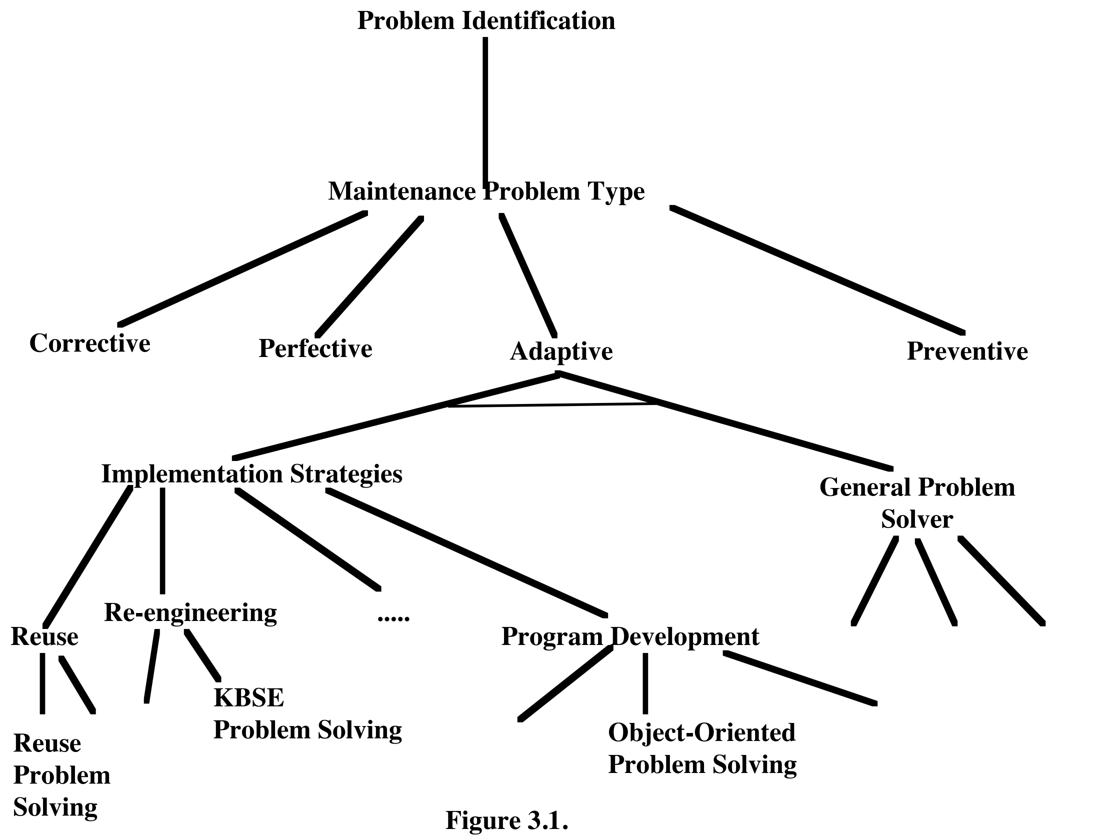


Figure 3.1.
And/or description of the Software Service Bay.

The overall problem solving procedure is described in a task oriented fashion using an and/or tree above. The first task in the operationalization scheme (figure 3.1) is Problem Identification. This relates to the first phase of the Service Bay methodology, namely, posing the specific maintenance problem. The next task is to determine the general maintenance type. Next, one must choose between the existing specialists for the given maintenance type. For example, the possible specialists for the adaptive maintenance category are reuse, re-engineering, and general purpose problem solvers. In order for the specialist to be selected, certain preconditions must be satisfied. If none of the available problem solvers for the maintenance category has its precondition satisfied, then a transformation specialist is activated. The task of the specialist is to attempt to transform aspects of the existing problem in such a way as to allow the transformed problem to satisfy the precondition for one of the problem solvers. The goal is to minimize the amount of effort performed in the transformation. Each time a problem transformation is applied, a reduction in the search space of possible solutions for the entire problem is made. The transformations are applied in an opportunistic way so that each time the conditions are satisfied for a special purpose problem solver to be used, it will be invoked upon that particular subproblem.

The special purpose problem solvers are represented in the Software Service Bay as *specialists* and generic tools are represented as *agents*. These specialists and agents are experts or tools in one particular area of the maintenance problem solving process and are functionally independent but work in conjunction with one another in an opportunistic fashion (see figure 3.2). For example, the problem reformulation specialist may determine that a particular subproblem meets the condition to be solved by the reuse specialist, the reuse specialist would then attempt to solve that problem. In another type of interaction between agents: the Static Structure agent, which computes and displays the static structure of a given software component, produces information used by the

Program Understanding agent, which needs the static structure information to construct a high level description of the component.

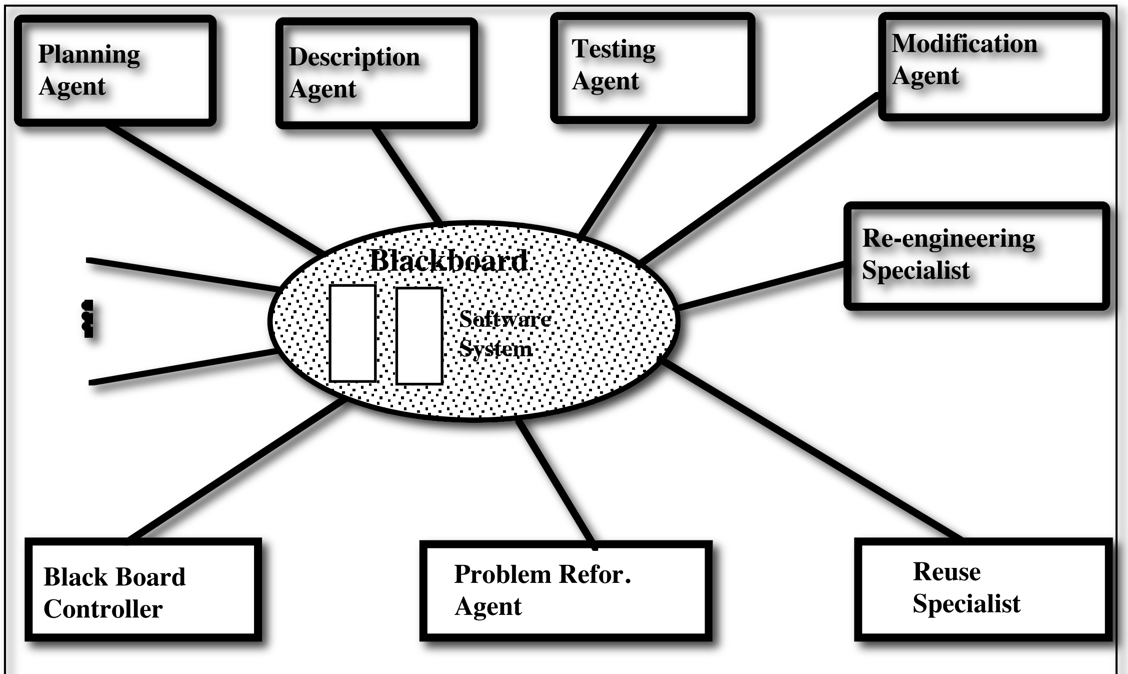


Figure 3.2. Software Service Bay Specialists and Agents.

The following is a suggested list of possible (Semi-) Automated Service Bay agents and specialists:

- Reuse specialist- Classifies, retrievals, and storage of software components.
- Re-engineering specialist- Applies re-engineering problem solving to maintenance problems
- Reformulation agent- Transforms problem for solution by specialists.
- Maintenance Strategy/Plan agent- Develops plans and strategies.
- Modification agent- Planner, organizer of modification to component.
- Static and Dynamic Structure agent- Compute and display static/dynamic structure.

- Description agent- Fact sheet, and viewing information about components.
- Understanding agent- At different levels.
- Testing agent- Testing of a component, history of testing.

The reuse specialist is a particularly important part of the Software Service Bay Methodology. By incorporating a reuse library and a reuse based problem solving paradigm into the maintenance problem solving process, a programmer is able to solve certain maintenance problems by reusing existing software. Going back to the analogy of the automotive service bay, the reuse library is much like the parts department at an automotive service bay. An inventory of new (and often times used) parts for a variety of vehicles are kept on hand so the mechanics may fix the malfunctions found.

3.2. Maintenance as a Problem in Software Reuse.

The problem solving process that takes the path, in figure 3.1, from *problem identification* along *perfective* down to reuse represents a solution of a software maintenance problem using a software reuse paradigm.

There are many cases in which maintenance problems can be solved using a reuse based paradigm. For example, in perfective maintenance a new specification for a component may match closely with an existing component in the reuse library. This existing component can then be inserted, with possible modification, as the solution to the specification.

3.3. Problem Solving and the Software Service Bay.

The maintenance problem solving process within the Software Service Bay Methodology involves the development of a task structure (knowledge based) for generating a plan to solve a given maintenance problem. This bears some similarity to

Chandrasekaran's task based design framework [Chandrasekaran 90]. Chandrasekaran describes design problem solving as a complex activity involving a number of subtasks with a number of alternative methods available for each subtask. The design problem is viewed as a search problem in a large space for objects that satisfy the constraints of the problem. Design involves a mapping from the space of design specifications to the space of components [Chandrasekaran 90, Simon 86] (e.g., in the case of software these components are implemented code).

For the process to be feasible, strategies that reduce the search space to a manageable size are needed. For the Software Service Bay to be feasible methodology a set of strategies must be realized which reduce the size of the maintenance problem solving space. The first of these strategies is to provide a mapping between the specification of a software system and the implementation of that system. With such a mapping the complexity of the maintenance problem solving process is greatly reduced. A mapping between formal specification of a software system and the implementation is supported during the software development process. The support a software design methodology gives to the construction of this type of mapping may be explicit or implicit. The Software Service Bay requires an explicit mapping between the formal specification and implementation. If the software system was not developed using a design methodology which explicitly supported the construction of such a mapping (such as [Smith 90, Rich 88]), then the mapping must be constructed using other methods [Abd-El-Hafiz 94, Harandi 90, Hartman 91, Alagappan 91].

Solving a maintenance problem within the Service Bay involves selecting a method for the problem solving task (i.e., maintenance task). A method can be described by the set of operators it uses, the objects being operated on, and the domain knowledge needed to apply the operators to the objects to satisfy the task [Chandrasekaran 90]. There are a number of different classes of methods which can be applied to the task, [Chandrasekaran 90] including knowledge based systems and direct algorithmic solutions. The Service

Bay is able to support a variety of problem solving methods each acting as an independent agent. Determining which of the problem solving methods is most appropriate for the particular task at hand is an important problem for the Software Service Bay.

The choice of problem solving method needs to be based on the properties of the solution (or goal), the solution process, and the availability of the knowledge required by the method [Chandrasekaran 90]. Each of the Service Bay's agents must have a procedure for determining when a particular problem is of interest. That is, if an agent finds a problem very interesting it will most likely be able to apply its problem solving methods to the task at hand. Since the knowledge representation of a software system is hierarchical in natural, heuristics which operate on such structures can be utilized. A heuristic measure of *interestingness* has been proposed [Hamilton 95] which indicates areas in a knowledge base that may be of interest to a method. The heuristic is based on the estimated complexity of a node in a hierarchical structure. Similar metrics, refinement metrics [Maletic 89, Reynolds 90], where proposed to estimate the syntactic complexity of program stubs in pseudocode. These types of heuristics measure the potential for information to be appropriate for a particular problem solving method. A measure of problem interestingness will need to be constructed or learned for each of the Service Bay agents.

3.4. Summary.

To develop a prototype system of the Software Service Bay, based on the concept of reuse as a specific problem solving method for maintenance problems, a knowledge based system for software reuse, The PM System, [Reynolds, Maletic 90], will be examined. This system will be embedded into the Service Bay framework such that the components of the PM System will act as the agents and specialists for parts of the reuse aspects of the Service Bay methodology. This will be discussed in detail in Chapter 5. Before the specific agents are introduced a more formal description of the control structure for the Service Bay will be described.

4. Controlling the Maintenance Process in a Reuse Environment.

In viewing maintenance as a software reuse problem solving process, Basili [Basili 90] identifies a variety of models to represent particular maintenance problems. One type of maintenance problem solving model may be more readily applied to a particular maintenance problem then another model [Basili 90]. This section will describe how the Software Service Bay can support maintenance in a reuse environment. Given a software system developed in a reuse environment, many questions can be answered about what types of maintenance problem could be encountered and about the problem solving strategies applied to such problems.

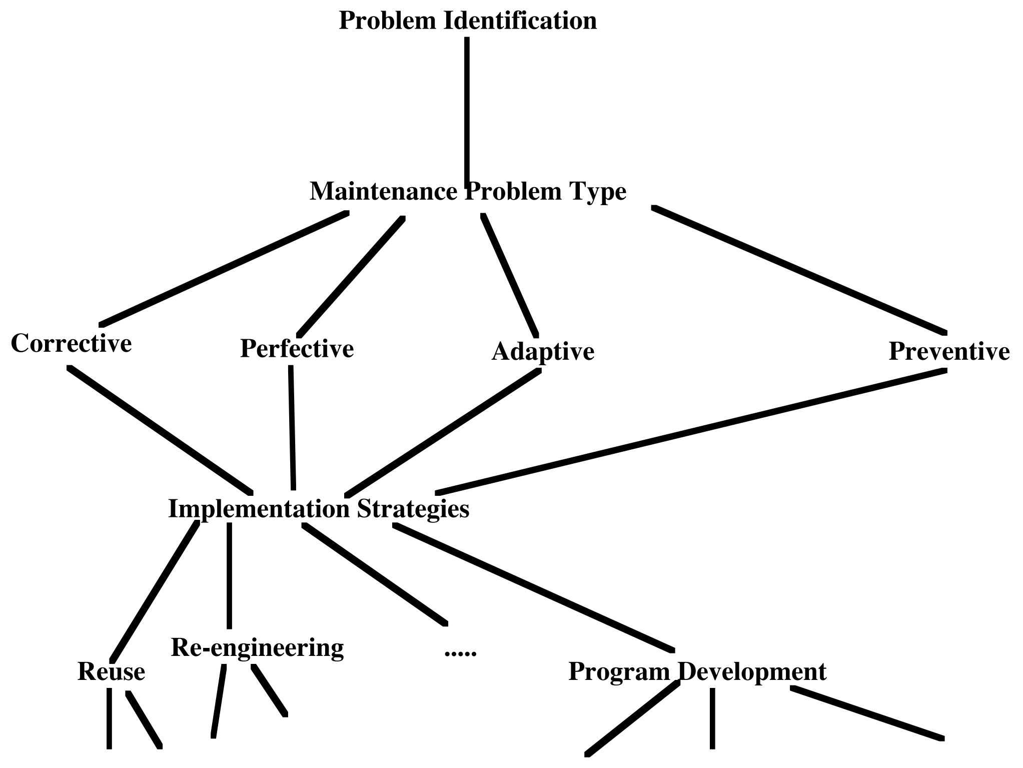


Figure 4.1.
Process Control of the Software Service Bay.

Within a reuse environment certain types of maintenance problems will be encountered less often. Corrective maintenance will occur less frequently since many components are taken from a reuse library. Components take from the reuse library are tested prior to insertion to the library. Preventive maintenance will also be less of an issue for the same reasons. Components taken from a reuse library are by default well written, and maintainable. Therefore, doing maintenance in a reuse environment to increase a systems maintainability is moot in general.

The general control structure for the Software Service Bay is given in figure 4.1. Each of the process nodes corresponds with one or more the phases of the Service Bay. Each of these phases is supported by a Service Bay agent which specializes in solving that particular type of control problem (figure 4.2).

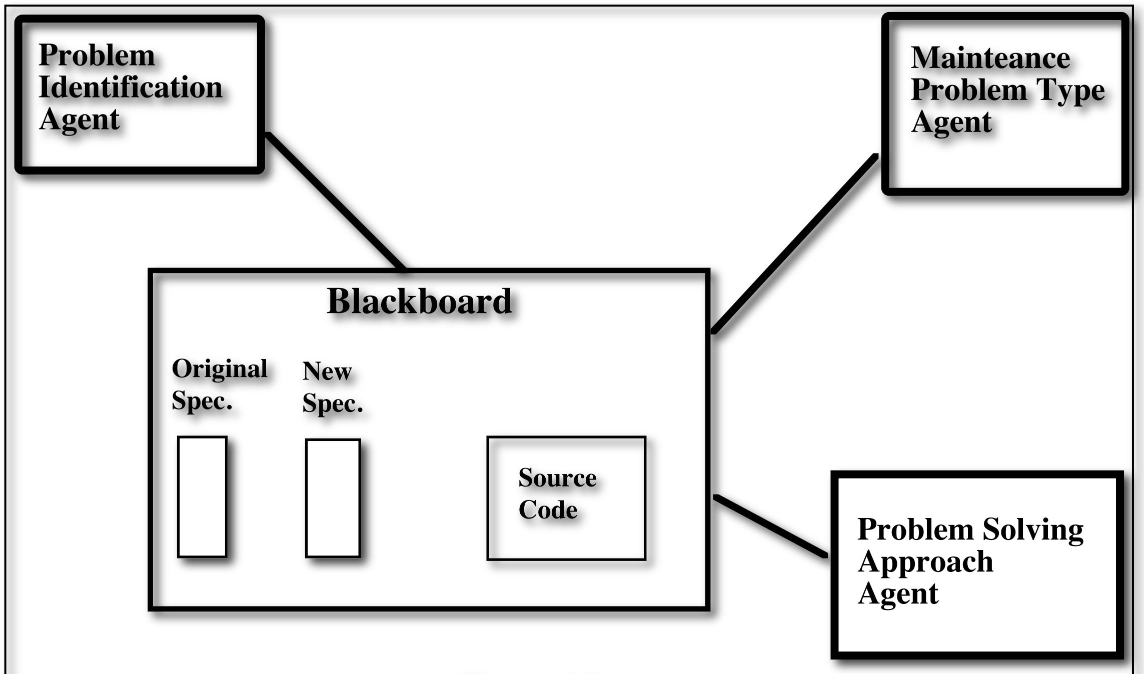

Problem Identification Agent
Mainteance Problem Type Agent
Blackboard
Original Spec.
New Spec.
Source Code
Problem Solving Approach Agent

Figure 4.2.
Software Service Bay Agents for determining Problem Solving Approaches.

Problem Identification works to solve the problem of posing the maintenance problem. A specification language which supports the maintenance problem solving process will be needed for this process (section 4.1). Given the specification of the current system, S, and the specification of the proposed new system, S', an identification of what type of maintenance problem can be made (section 4.2). The identification of the type of maintenance problem will direct the Service Bay problem solving mechanism and supply valuable knowledge about what the types of maintenance problems that could be enchanted. An example system and maintenance problem will be presented to demonstrate these issues (section 4.3) The next step in the Service Bay framework will be to identify what type of problem solving approach is needed to solve the problem at hand (section 4.4).

4.1. Problem Identification.

A maintenance problem is posed by presenting a formal specification, S', of the new intended software system. This new specification is a derived from the specification, S, which describes the current software system. The choice of the specification language and how the specifications are developed must be tailored for the maintenance problem solving process. What properties should a specification language poses in order to support the maintenance problem solving process? How could the specifications be developed such that they are conducive to solving a maintenance problem?

Criteria for judging specifications and the implications those criteria have on specification languages have been suggested [Balzer 86]. Maintenance is addressed in much the same way as it is in program (code) development, that is, make sure the specification is written in such a way that it will be maintainable. What is needed in the case of the Software Service Bay is a set of guide lines for the development of specifications. With certain guide lines in place, a number of assumptions can be made which will further problem solving by other Service Bay agents.

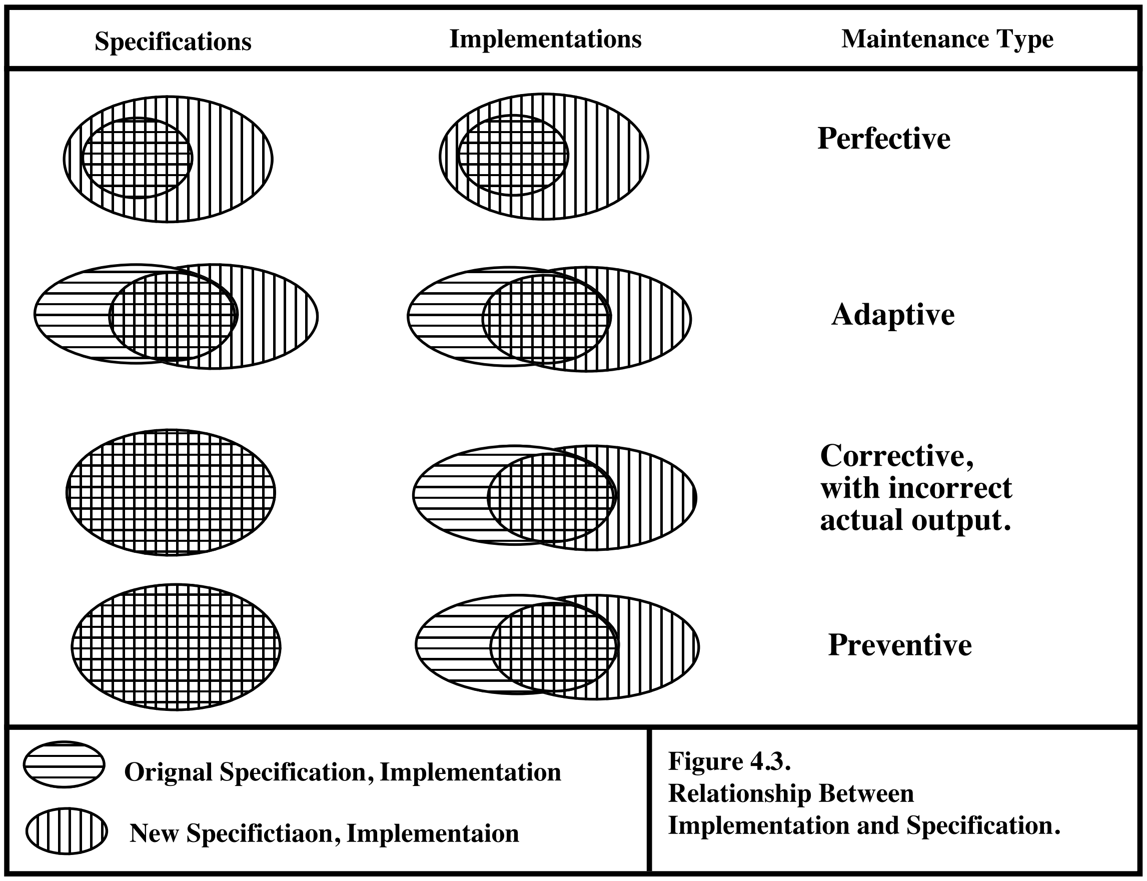


Figure 4.3.
Relationship Between
Implementation and Specification.

4.3. Relationship between implementation and specification.';

A specification language must support modularity, decomposability, and relationships between specification and implementation. Because the system is already written and working, the specifications should reflect the implementation. Therefore, a change in the specification can be directly mapped to parts of the implementation.

4.2. Determining Maintenance Type.

The formal specification of an existing software system consists of:

S = <Function, Input, Output, Machine env, Operating env, Implementation>.

Where the Function is a formal functional specification of the system written in a formal specification language, Input is a specification of the input to the system, Output is a specification of the Output of the system given the Input, Machine environment is the type of hardware the system is running, Operating environment is the operating systems, data base systems, user interface environment that the system utilizes, and the Implementation is the source code of the system that corresponds to formal specification. There exists a mapping between the functional specification and the implementation of the system.

Figure 4.3 displays the relationship between a system's specification and its implementation. For perfective maintenance a change in the specification is needed due to the fact that new or additional functionality is proposed. Therefore the new specification for the system will contain the original specification. Similarly, the implementation for the new system will consist of the old system plus an amount of new source code. Adaptive maintenance is characterized as a change in the specification of the system, normally a change in the machine or operating environment, and a change in the implementation of the system. These changes represent the removal of some of the original system and its replacement with new material, be it new specifications or source code. Corrective maintenance, or an implementation error, is represented by no change in the specification of the original system and with the removal and replacement of part of the implementation. The removal and replacement of part of the source code reflects the fixing of a "bug". Corrective maintenance is tied in closely with the actual output of the system, i.e., an incorrect output value, according to the original specification, must be detected and a request by the customer for its correction must be made. Preventive maintenance is very in appearance with corrective maintenance. The only difference is that the system is producing correct output. The removal and replacement of source code reflects the modification of code to prevent future problem with maintenance or to anticipate future perfective maintenance problems.

4.3. An Example System Specification.

An example system specification from the literature, an airline reservation system developed in Spec [Berzins 91], will be used to illustrate the criteria needed by the Service Bay to support maintenance problem identification. Spec [Berzins 91], the formal specification language used for this example is designed for large scale software applications. The Spec language uses a second order temporal logic which includes first order logic as a subset. A complete specification of Spec can be found in [Berzins 91]. There are a number of formal specification languages [Gehani 86, Claben 93, Jones 91, Alexander 90] which could be used with in the Software Service Bay framework, Spec and the airline reservation system provide a robust example for demonstration with in this framework.

The specifications for the airline reservation system example [Berzins 91] (see Appendix A) have been run through a Spec syntax checker and the system has been coded in the Ada programming language (see Appendix B). The code was found to run in accordance with the specification on all test cases tried by the authors. The system was not verified to be correct mathematically.

The purpose of the airline reservation system is to help the travel agent sell tickets to passengers on commercial airlines. The travel agents must be able to find all of the flights meeting a passenger's needs and make reservations based on the passenger's perferences.

Figure 4.3.
Initial problem statement of the airline reservation system [Benzins 91]

G1: The purpose of the system is to help the travel agent sell tickets.

G1.1: The system must help the travel agent find all of the flights meeting a passengers needs.

G1.1.1: The system must be able to find all flights with a given origin and destination.

G1.1.2: The system must display the arrival and departure times, the price and the flight number for each flight it finds.

G1.1.3: The passenger will choose a flight based on the displayed information.

G1.2: The system must let the travel agent make reservations.

G1.3: The system must allow the travel agent to cancel reservations.

G1.4: The system must be able to find all reservations on a given flight after a given date.

G2: The system must provide a means for the airline manager to manage flights.

G2.1: The system must allow the airline manager to schedule new flights.

G2.2: The system must allow the airline manager to drop flights from the schedule.

G2.3: The system must allow the airline manager to change the fare for a flight.

Figure 4.4.Summary of goals for airline reservation system [Berzins 91]

The initial problem statement for the airline reservation system is given in figure 4.2. The high level goals of the system are to help the travel agent sell tickets, help the travel agent find all flights meeting a passengers needs, and let the travel agent make reservations. Some of the high level constraints of the system are that the system be implemented in Ada, that the performance of the system be fast enough for real time use (less than a second response time), and the system be developed by three people in ten weeks. A complete set of the refined goals of the airline reservation system are given in figure 4.3.

```
MACHINE airline_reservation_system
  INHERIT travel_agent_interface
  INHERIT airline_manager_interface
END

MACHINE travel_agent{id: nat}
END

INSTANCE travel_agent_instance{i: nat}
    --Declares 300 instances of the travel agent machine.
  WHERE travel_agent_instance{i} = travel_agent{i}
  FOREACH(i: nat SUCH THAT 1 <= I <= 300)
END

MACHINE airline_manager
END
```

Figure 4.5.
Skeleton Specification for proposed and external systems [Berzins 91].

From the requirements analysis stage, a formal specification is developed for the system. Figure 4.4 gives the skeleton specification for the proposed and external systems. The complete formal specification for the airline reservation system can be found in Appendix A. A diagram of the airline reservation system is given in figure 4.5, and a type decomposition is given in figure 4.6.

The system is organized around interfaces, there is one external interface for each external system, i.e., `travel_agent_interface` and `airline_manager_interface`. There is a Spec module for each external system and proposed subsystem.

To demonstrate the issues of maintenance a couple of maintenance problems will now be proposed for the airline reservation system just described. How the specification for the system is affected and how the Service Bay will use the changed specification will be discussed.

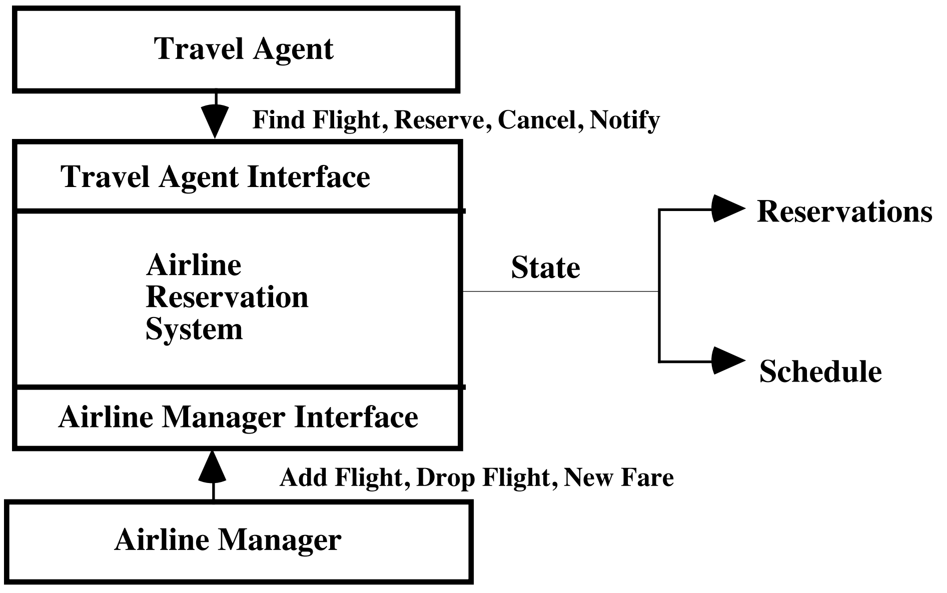

Travel Agent
Find Flight, Reserve, Cancel, Notify
Travel Agent Interface
Airline Reservation System
State
Reservations
Schedule
Airline Manager Interface
Add Flight, Drop Flight, New Fare
Airline Manager

Figure 4.6.
System Diagram for Airline Reservaton System [Berzin 91]

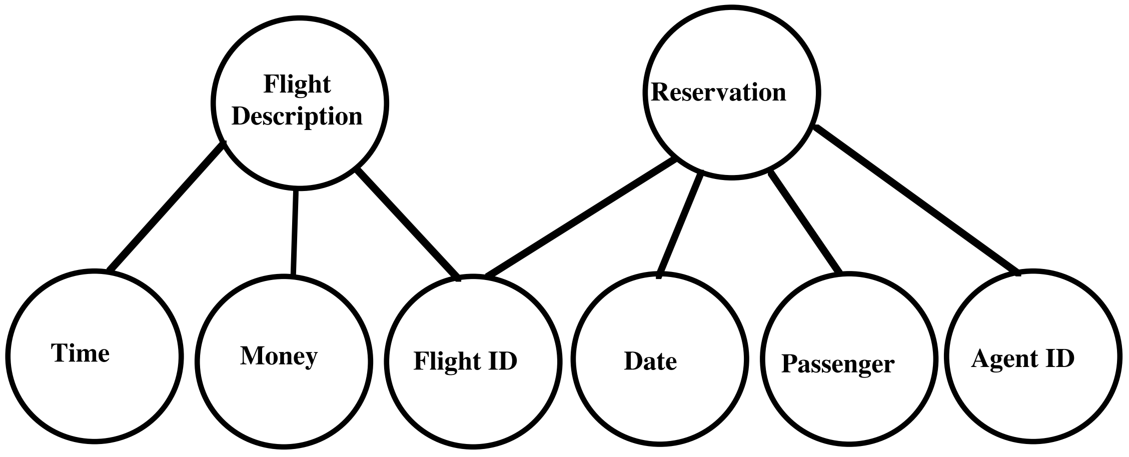


Figure 4.7.
Type Decomposition for the Airline Reservation System
[Berzin 91]

4.4. Example Maintenance Problems.

To demonstrate the issues of the maintenance problem solving process two example maintenance problems, for the airline reservation system, will be presented. The first example is an adaptive maintenance problem. The problem is to upgrade the user interface for the airline manager subsystem from a command line interface to a menu driven type system. The second example will be a perfective maintenance problem. The problem here is to add capabilities for the travel agent to make seat assignments. The impact of the maintenance solutions for these problem will be discussed. The amount of the specification needing modification, projected amount of re-coding, and relative location of changes will be specifically addressed.

4.4.1. An Adaptive Maintenance Problem.

A common adaptive maintenance problem is to modify the user interface of a system. In this case, the system uses a simple command line interface and the customer would like the user interface for the airline manager subsystem to be changed to a menu driven type interface. The airline managers would like to have a command menu and prompts for the arguments associated with each command. The problem does not involve a change in the functionality of the of the system, but instead concerns a change in the output environment of the system there fore the problem is classified as an adaptive maintenance problem.

The first two phases of the Service Bay Methodology have been addressed above. Namely, posing the maintenance problem and determining the maintenance type. The next phase in the Service Bay Methodology is to determine what types of knowledge are needed to solve this type of maintenance problem. In this case knowledge of the type of interface requested by the customer will be required. Also, the parts of the specification and implementation affected by such a system modification need to be identified. This maintenance problem is localized to only two parts of the system specification, namely the command interface for the airline manger, (see appendix A, A.8) and the airline manager command formats (see appendix A, A.9). These two parts of the specification deal with the interface between the system and the user, in this case the airline manager. Since an adaptive maintenance problem does not effect the formal specification, because there is no change in the functionality of the system, the only changes to the system is in the implementation and the requirement of the operating environment. Figure 4.8 is a gives a segment of Ada source code which represents part of the implementation for the airline manager interface (also see appendix B, B.2). This is the segment of code which would require modifications to reflect the changes requested by the customer.

```
task body airline_manager_type is
  input, output:   file_type;
  last:            natural;
  buffer, command: text;
begin
  open(input, in_file, input_name);
  open(output, out_file, output_name);
  loop  --Forever.
    get_line(input,buffer);
    buffer := edit_command(buffer);
    next_item(buffer, command);
    interpret_command(ada_string(command), buffer, output);
end loop;

-- rest of code, exceptions...

end airline_manger_type;
```

Figure 4.8.
Ada Code Segment for Airline Manager [Berzins 91].

Notice that this problem will do little to impact the entire system, rather it is a localized problem. The actual source code modifications are localized to a relatively small part of the system with no impact to the rest of the system.

At this point in the maintenance process a problem solving strategy can be selected which is conducive to solving the problem at hand. Given the existence of a reuse library, a similar type of menu driven interface can be located. This component from the reuse library can then be integrated in the airline manager subsystem to solve the maintenance problem.

4.4.2. A Perfective Maintenance Problem.

The maintenance problem presented here is typical perfective maintenance problem. The customer asks for a follow on to the system which will add the capabilities for the travel agent to make seat assignments. This maintenance problem is much more complex than the previous example. Here a modification to the data representation of the reservation is needed to implement the requested changes along with added functionality to travel agent subsystems and to the airline reservation state model.

A reservation cannot be made on a full flight, so the total bookings available and the total number of reservations for a flight must be checked. The travel agent will make the seat assignments based on first come first serve from the front of the plane back. One new goal for the system is added to the original set of goals (see figure 4.4.) and is presented in figure 4.10. Figure 4.9 describes the addition of the seat assignment information to the type decomposition for the system.

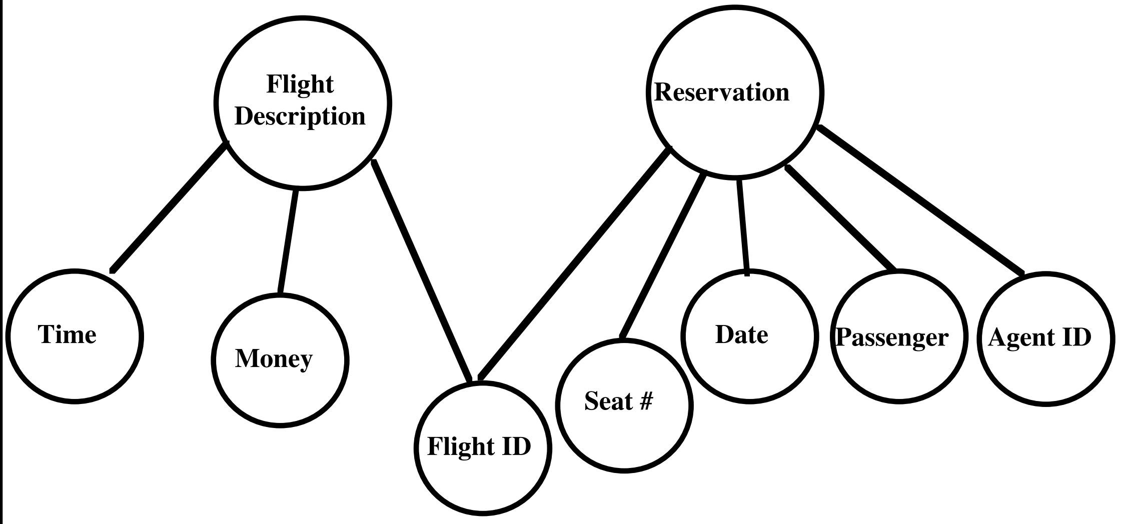


Figure 4.9.
New Type Decomposition for the Airline Reservation System [Berzin 91]

G1.5: The system must be able to allow the travel agent to make seat assignments.

Figure 4.10.Additional goal for airline reservation system

The affected parts of the formal specification of the system will include the travel agent interface (appendix A, A.1), the state model concepts (appendix A, A.2), the command interface for the travel agent (appendix A, A4), the travel agent command formats (appendix A, A.5), the travel agent output formats (appendix A, A.6), the type definitions (appendix A, A.7), and the travel agent view (appendix A, A.13). The segment of source code affected by the maintenance process will include the packages: travel_agent_pkg (appendix B, B.1) and airline_reservation_system_pkg (appendix B, B.3). Also, the type declarations will be affected by the maintenance process.

4.5. Suggesting a Problem Solving Approach.

There is an explicit need for a mapping between the formal specification and the implementation. Given this mapping, a modification to the high level specification can be mapped to the affected code implementation. In the airline example, the mapping between the formal specification and the Ada code is implicit but apparent due to the relatively size of the system. Because this mapping is identifiable, for this example, changes to the source code can be regulated and the global effects identified using methods such as program slicing [Korel 90, Weiser 84, Horwitz 90] and code browsing [Rajlich 90].

A heuristic to suggest a proper problem solving method can be based on the scope of the affected source code. A metric(s) to predict how localized the effect the maintenance problem solving process will have on the source code would be useful. This type of metric along with the knowledge of the maintenance type can suggest a problem solving approach. In the case of using software reuse for the problem solving method, a metric value the reflects a set of localized changes is most appropriate. If the changes were very distributed throughout the implementation of the system reuse will not be feasible.

For both example maintenance problems, the interface upgrade and the seat assignment problem, the effects to the implementation were relatively localized. Both examples would therefore be candidates for applying software reuse to solve the maintenance tasks. It will be necessary to have a reuse library with knowledge of the required components to solve the maintenance problems.

5. A Reuse System in the Software Service Bay Framework.

This chapter focuses on the specific agents needed for problem solving via reuse in the Software Service Bay. Biggerstaff and Richter suggest that there are four fundamental subtasks associated with the overall task of operationalizing the reuse process [Biggerstaff 87]: finding reusable components, understanding these components, modifying these components, and composing components. The Partial Metrics (PM) System's [Reynolds, Maletic 90, Reynolds 92], focuses on operationalizing the first two of these subtasks, namely: finding and understanding reusable components. As such, it is a good example of how a particular path in the search tree can be embedded in the blackboard framework. The overall Service Bay organization will support a number of these paths (see figure 5.1).

5.1. The PM System.

The PM System works in two phases: the learning phase and the production phase. In the learning phase PM extracts component reuse knowledge from software systems. This phase builds a knowledge base for reuse in the software domain currently at hand. The knowledge acquired may be at different levels of granularity, system, procedural, or code level, depending on the domain or needs of the users. Once the knowledge is acquired the PM system can then be used in a production mode to assess, store, and retrieve reusable software components.

The problem of software reuse requires the coordination of a number of different types of knowledge. Within the Software Service Bay each type of knowledge is utilized and acquired by a specialized agent. Each of these agents act independently accessing global data from a centralized blackboard. The PM System (see figure 5.1) is composed of four agents: The Structural Reusability Classification Agent, The Dynamic

Performance Analysis Agent, The Task Understanding Agent, and The Reuse Library Agent.

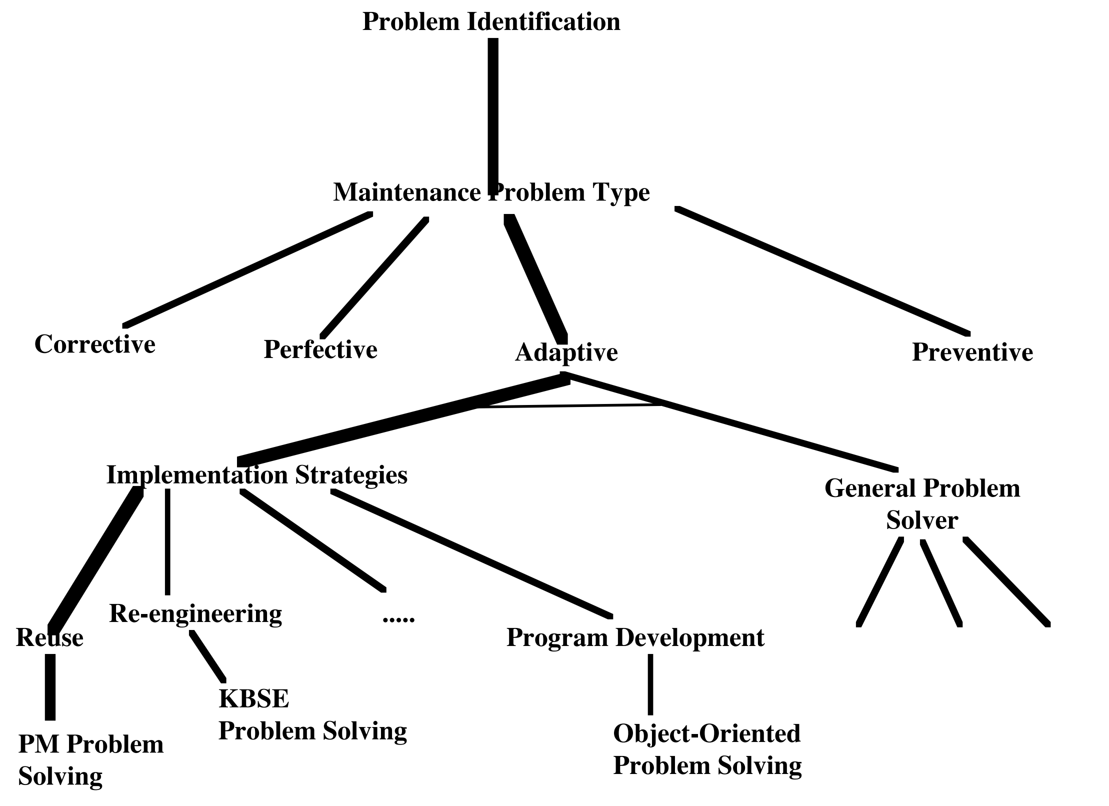


Figure 5.1.
PM and the Software Service Bay.

5.1.2. PM Agents.

The Structural Reusability Classification (SRC) Agent assesses the static structure of the candidate object and the Dynamic Performance Analysis (DPA) Agent is used to isolate reusable components or chunks in the candidate system based upon their behavioral performance. The criteria used for the assessment is a function of the granularity at which the programming knowledge is extracted. These two agents support the first phase of Biggerstaff's reuse process.

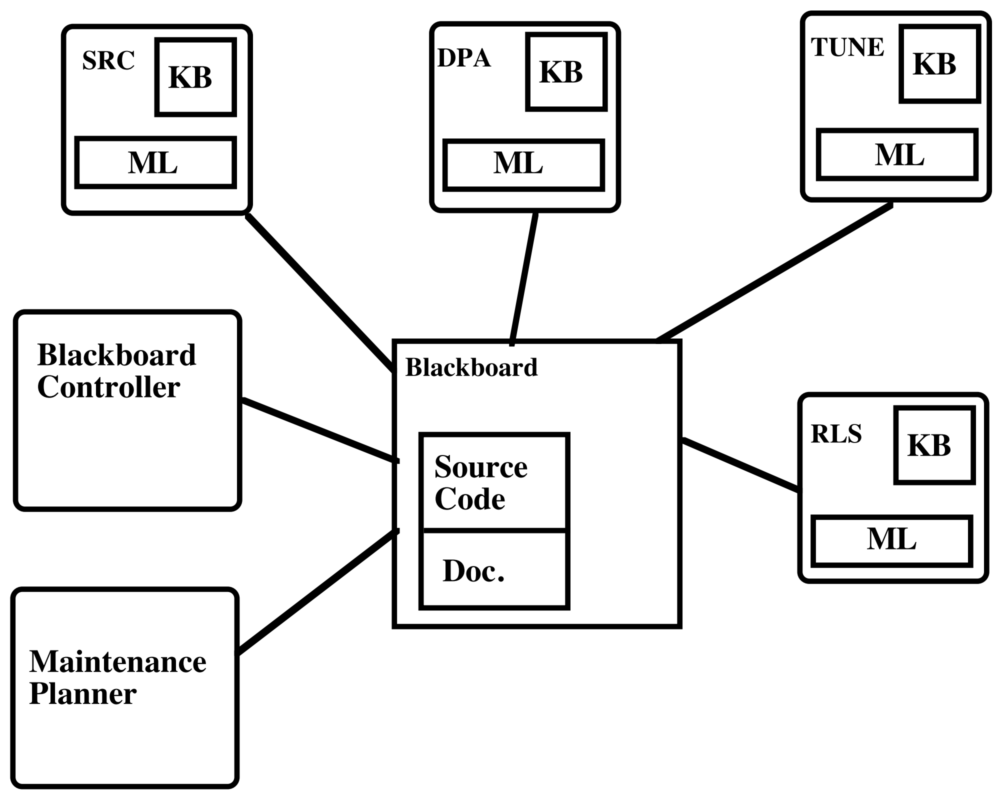


Figure 5.2.
Specific Reuse Agents in the Software Service Bay taken from the PM System.
KB: Knowledge Base
ML: Machine Learning Subsystem

The Task Understanding (TUNE) [Reynolds 94, Posner 94] Agent supports Biggerstaff's second phase of the reusability process, understanding the component to be reused in terms of a faceted classification scheme. The goal here is to produce a generalized description of the context in which the candidate can be meaningfully utilized. In the current version, a faceted classification scheme is used for this purpose. The classification structure for the storage of the modules follows the approach developed by Prieto-Diaz [Prieto-Diaz 85]. Each module is analyzed by the TUNE agent and a list of facet-value pairs that are supported by the code is produced. This agent is developed as a knowledge base with the rules determining the values for the facets depending the presence of certain information in the code of the module. Currently, the description has six major facets: Object, Medium, Function, Functional Area, System, and Settings. An example facet-value pair might be (Object, array). The code is analyzed and appropriate keywords are extracted for each facet in order to reasonably describe the module.

The Reuse Library Agent (RLS) [Wickey 94] stores the code components that are found to be reusable. RLS employs an incremental conceptual clustering algorithm presented by Fisher [Fisher 90] to build the library index. The stored objects are represented by facet vectors.

Each of the agents described above is supported by a knowledge-base that is used to generate the reuse decisions performed by that agent. This knowledge is acquired automatically during a learning/configuration phase of the PM system by each agent. In the configuration process, the agents are provided with examples of acceptable and unacceptable decisions. This approach is predicated on two basic assumptions [Esteva 91]:

1. It is easier to show an example of what is or is not reusable code than to precisely describe the criteria used in the judgment.
2. The criteria used to make a reuse decision in a given application are relatively straightforward but they can change dramatically from one application to another.

The reuse criteria learned for each agent is stored in its associated knowledge base. That knowledge base is accessed by the corresponding subsystem during the processing of a candidate at a given level of granularity. After the knowledge base is constructed then the agents are able to solve particular problems or assess components within their domain. The control structure for that portion of the Software Service Bay that supports the generic reuse process of the PM system will now be discussed.

5.1.2. Blackboard Controller for the PM System.

The blackboard controller acts upon the two specific goals of the each of the PM system agents. Each agent has a goal of learning a knowledge base to assist in solving a particular problem in the reuse framework and a goal of applying the learned knowledge base to particular problem instances. The controller manages and determines the specific goals of the agents, whether learning or problem solving. If the goal is to learn reuse knowledge, the blackboard contains the set of examples needed by the agents.

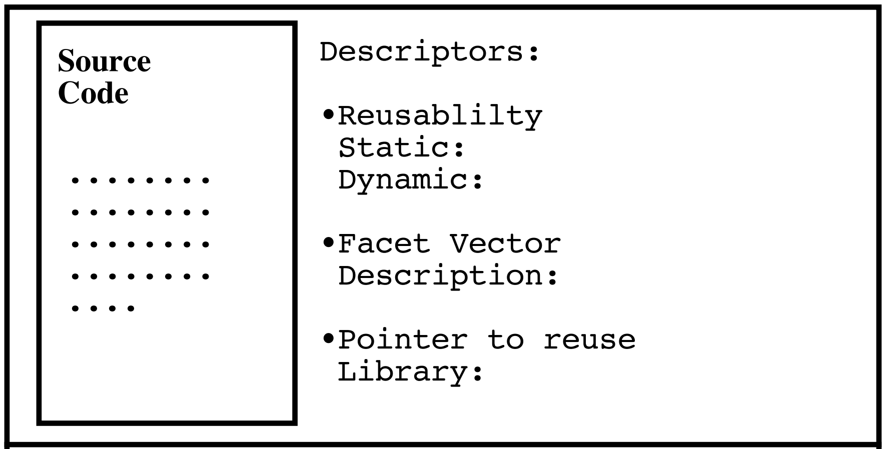

Source Code
Descriptors:
•Reusablilty Static: Dynamic:
•Facet Vector Description:
•Pointer to reuse Library:

Figure 5.3.
Knowledge contained in Blackboard.

When the PM system is utilized to solve a maintenance problem, the blackboard contains the knowledge about the current state of the problem solving process. If, for example, the problem is to determine if a source code component is reusable and how to store a reusable component in a reuse library, the knowledge needed by the agents of the PM system is the source code, facet vector description, and the components candidacy for reusability. Figure 5.2 gives a description of the knowledge represented in the blackboard for the PM system. Each of the agents (figure 5.1) assess the knowledge in the blackboard and determines if its problem solving methods are needed to progress the current problem state into a state closer to the goal state, i.e. the component stored in a reuse library.

Each of the agents work independently, therefore the system should not be viewed as lower level processes producing data for higher level processes. For instance, an agent may request that another agent re-evaluate the blackboard to make another hypothesis about the data. An agent may find a set of examples particularly useful for learning and may hand these to higher or lower processes for analysis. Also, a component may be put on the blackboard with some user supplied descriptor values, the agents would act upon this information versus information computed by other agents.

Next, each of the Service Bay reuse agent's activation conditions and learning phases will be examined in more detail.

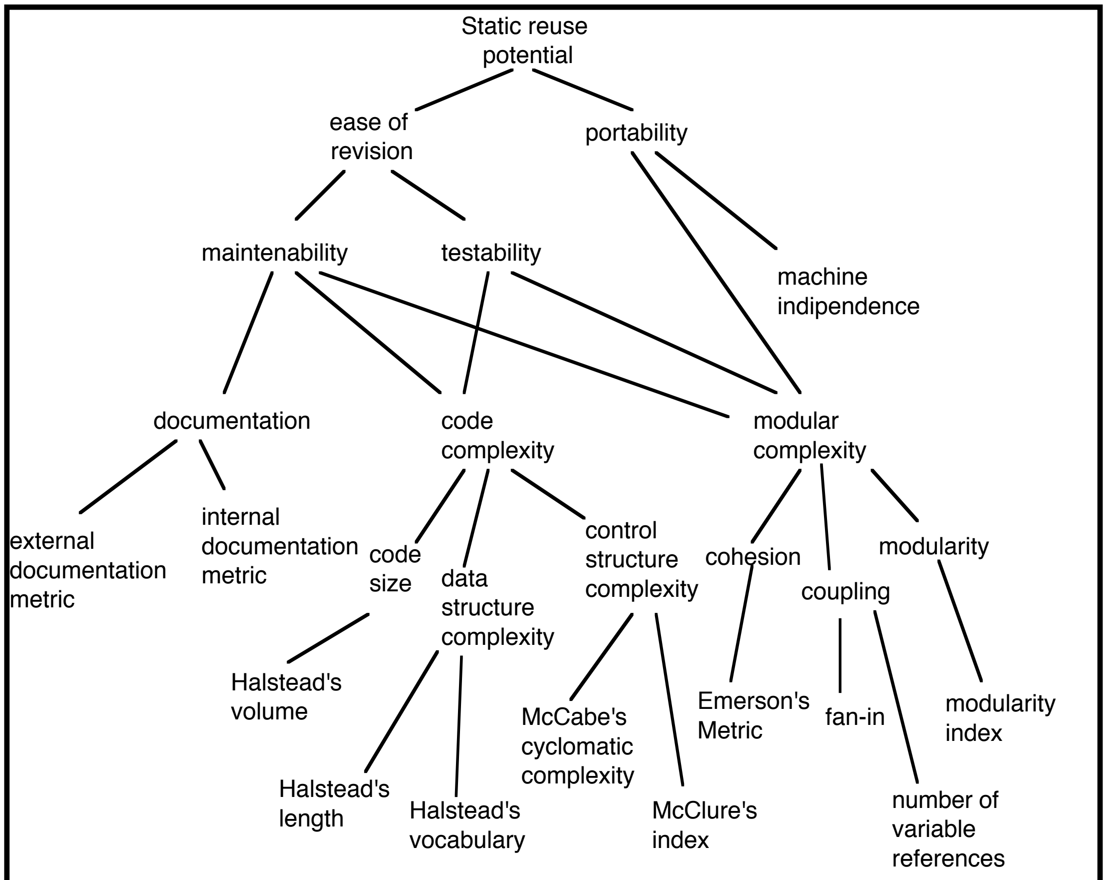
5.2. A Static Reusability Agent.

Assessment of a component's static structure requires little knowledge of the component. The Static Reusability Classification [Esteva 91] Agent (SRC) uses only the candidate's source code to perform an assessment of static structure. The SRC agent will therefore be activated any time the static reusability of a component is unknown, and the source code of the component is available on the blackboard.

An assessment of the static reusability of a components will identify ill-structured programs that can complicate behavioral assessment or the maintenance process. If the current maintenance problem is a preventive one, then the ill-structured components will be candidates for improvement. The SRC is also useful in assessing any new code components or component modifications added to a system because of a maintenance task. The new code added to a system then can be built to a learned quality measure. The quality measure can be derived in part from the existing system.

5.2.1. Quality Model for Static Reuse.

The criteria used to perform the static assessment process will vary with the level of granularity of a component. At the system level the criteria will relate to the complexity of its interface to the environment. At the procedural level, the criteria relate to procedural structure and content. At the code level, the criteria relate to syntactic structures used in the implementation. Here the focus will be on the procedural structure of components.



Figaure 5.4.
Quality Model used to assess static reusability

While the criteria for reuse are provided implicitly by those configuring the system in terms of positive and negative examples, the SRC system utilizes an explicit domain model to aid in explicitly operationalizing the criteria. The domain model currently in use at the procedural level is given in Figure 5.2. This is an example of a software quality model, similar to those of Boehm [Boehm 78], and Fenton [Fenton 91]. Here, the concept of reuse is decomposed into successively more specific factors, ultimately into metrics that assess reuse in terms of a systems static structure. Details concerning the metrics used can be found in [Esteva 91].

While the model given above specifies what factors can be important in determining the reusability of software, the acceptable values of and relative importance for each will vary from one application to another. The goal of the learning element associated with the SRC agent is to acquire this information for a specific application. The source of this information is a training set of examples of reusable and not reusable code. The system attempts to generate a decision procedure, expressed in terms of values for selected metrics, that is capable of discriminating between the positive and negative examples. This is an example of an inductive concept learning problem in which the concept to be learned is static reuse.

5.2.2. Learning in SRC.

Quinlan developed an approach to generating decision trees in a top down fashion based upon information theoretic principles [Quinlan 86]. The approach was implemented in a system designated as ID3. The ID3 procedure was used in SRC in a slightly modified form as a basis for generating decision trees. One assumption made in this approach was that the attributes used in forming the decision tree be categorical in nature. Therefore, for each attribute a limited number of observational categories were established, based upon standard interpretations of these metrics taken from the software engineering literature.

The procedure utilized in the SRC agent is described below:

```
Let M be the set of software modules to be classified by a subtree rooted at the
current node. Initially M contains modules from which the tree will be constructed.
1. If M is empty then Quit.
2. If all the modules of M are of the same class (positive examples or negative
   examples) or if M is empty then
     A. Designate the current node as a leaf node and label it according to the class of
        its objects.
     B. Quit
```

3. Apply an evaluation function to select an attribute A_i having possible values $A_{i1}, A_{i2},..., A_{ij}$ to partition M into j subsets. Select the attribute with the greatest discriminatory power, i.e. the degree in which an attribute partitions the objects into subsets. If no attribute is able to partition the set of objects, label it with both classes and quit.
4. Create a child node in the tree for each of the j classes.
5. For each child node recursively call this procedure

The evaluation function used to select the most appropriate attribute at each step was based on each variable's ability to sort the remaining M modules into homogeneous classes of either positive or negative examples according to their observational category for a given attribute. The extent to which an attribute was able to achieve this goal is termed its information gain. The information gain for each attribute, A_i, is calculated as:

$$\text{gain}(A_i)=H(p, n)-E(A_i).$$

The information gained by adding a decision node to the tree based upon attribute A_i is a function of the number of positive (p) and negative (n) examples currently associated with the parent node (H(p,n)) and the degree of homogeneity that will be expected ($E(A_i)$) within each category of A_i. H(p,n) is the entropy for the current distribution of positive and negative examples and represents the average information about the distribution received when an arbitrary example is selected. H(p,n) is calculated based upon expectation for the positive and negative examples where:

$$H(p, n)= - (\text{prob}(p) \log_2 \text{prob}(p)) - (\text{prob}(n) \log_2 \text{prob}(n))$$

Prob(p) is the probability of occurrence for positive examples while prob(n) is the probability for a negative example.

The expected information for an attribute A_i is the average of the proportion of positive and negative examples associated with each of the r categories for A_i weighted by the entropy for the distribution of positive and negative examples found in that category. Thus,

$$E(A_i) = \sum_{j=1}^{r} \frac{p_j+n_j}{p+n} H(p_j,n_j).$$

5.2.3. SRC Results.

Over two hundred Pascal programs from diverse software libraries, both academic and commercial, were used to generate example decision trees. A representative decision tree produced from these examples is given in figure 5.3. Decision nodes that represent the partitioning based upon values for a given attribute are displayed as diamonds. Leaf nodes are represented as circles and correspond to either acceptance (+) or rejection (-) of a module as the result of following the branches dictated by the metric values for the system. Note that this decision tree requires information about each of the major subtrees in the software quality model described earlier; documentation, code complexity, and modularity. Also, based upon fan-in from the higher level concepts one would expect modularity concerns to be most important of the three, followed by code complexity issues, and then documentation. This is exactly what occurs in the example.

In fact, experiments with the system have led to some general observations about the structure of the generated trees. Firstly, for any given application environment, the generated tree is relatively shallow and broad. The average number of decisions needed to classify a system is 3.1, with an average number of 7.4 internal nodes. Secondly, the decision procedure exhibited performance equal to or better than human experiments in making such discriminations, with accuracy's ranging from 75% to 98% depending upon the number of training examples used. In general, a training set of 25 or more examples, evenly divided between positive and negative instances, guaranteed accuracy's in excess of 90%. Thirdly, attributes associated with module complexity are most frequently found at or near the root. This reinforces the validity of the quality model used since in that model, modular structure influences three higher level factors in the underlying quality model: testability, maintainability, and portability. Lastly, although the decision trees for any given application are small, they can change dramatically in terms of the variables used from one application to the next. Once a decision tree is configured for an application, it is stored in the knowledge base associated with the SRC agent.

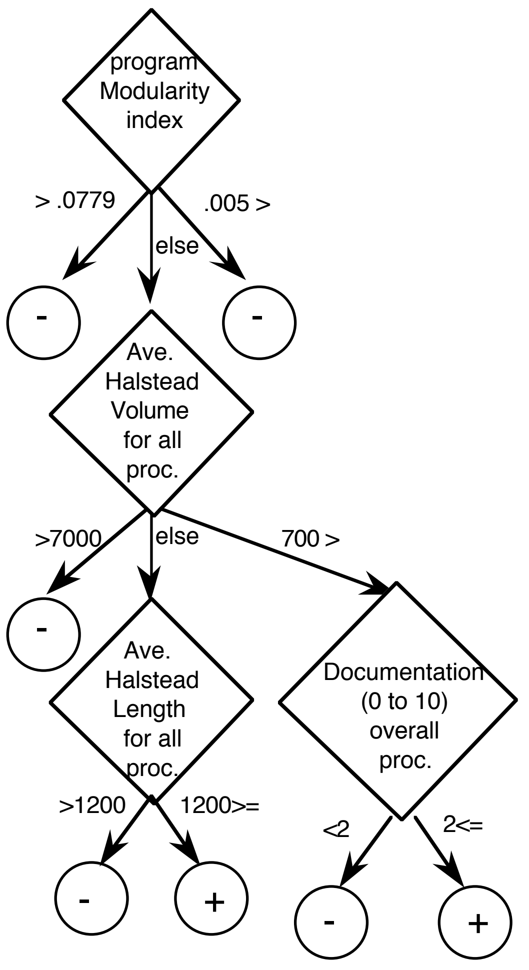

program Modularity index
> .0779
else
.005 >
-
-
Ave. Halstead Volume for all proc.
>7000
else
700 >
-
Ave. Halstead Length for all proc.
Documentation (0 to 10) overall proc.
>1200
1200>=
<2
2<=
-
+
-
+

Figure 5.5.
An Example Decision Tree

The modularity index is the first metric used to evaluate the program. This metric is the ratio of the number of procedures to the number of lines of executable code as proposed by Prieto-Diaz [Prieto-Diaz 85]. The value here is .07 which falls within the acceptable range. Thus, the next decision is based upon the average Halstead volume for all procedures. The value (~840) is also greater than 700 but much less than 7000. Halstead's volume is defined as

$$(\text{Length}) \log_2(\text{vocabulary}) .$$

Length is the total count of operators and operands in the procedure and vocabulary is the total number of unique operators and operands [Halstead 77]. The next decision is based upon the average length, as defined above, of each procedure. The average length here is ~175 which is again within acceptable limits (1200>=). As a result, the SRC agent returns the class associated with the terminal node in the path which is '+'.

5.3. A Dynamic Reusability Agent.

The goal of the Dynamic Performance Analysis (DPA) agent is to isolate reusable candidate structures at a given level of code granularity. The agent can be activated when knowledge of the components static reusability is present on the blackboard along with the source code for the component.

5.3.1. Chunking in DPA.

The general model for the process supported by the DPA agent is that of chunking. Chunking involves the encoding of detailed knowledge into a more abstract form and its decoding, when necessary, to unveil the detailed information. The chunking process is motivated by the need for humans to associate specific input with more general or abstract patterns. This approach was suggested by the work of Miller [Miller 56] on the size of a human's short-term memory. His results suggested that an average person has a limit of seven plus or minus two items. This limitation can be overcome when each of the objects in short-term memory is allowed to represent an encoding of a collection of more detailed objects. For example, few people would be able to remember the string of bits 010110101100. However, if the string is linearly decomposed into three 4-bit subsequences, (0101, 1010, and 1100), then each 4-bit sequence can be encoded in its hexadecimal equivalent. The resultant set of objects, 5AC, is easy to remember. This result can then be decoded into the original, more detailed, representation if necessary.

The above example suggests that chunking consists of several distinct phases. The first phase identifies groups of domain objects sharing structural or behavioral properties, i.e. showing meaningful patterns. Each group can be viewed as an aggregate (or chunk). These aggregates should be relatively independent in order to simplify the encoding process. The decomposition of the binary sequence above can be expressed hierarchically as

```
  010110101100
   /    |    \
0101  1010  1100
```

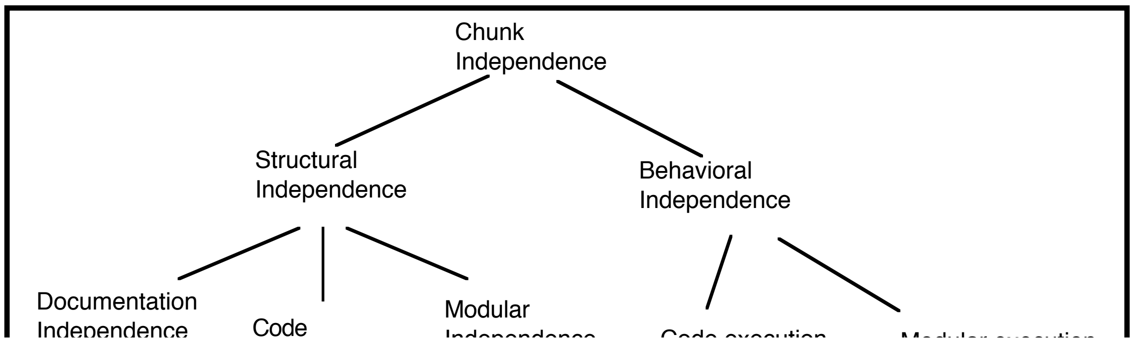
Simon observed [Simon 67] that many complex systems can be decomposed into a collection of hierarchically structured subsystems, such that the intra-system interactions are relatively independent of inter-system ones. If there is no dependence between the inter- and intra-subsystems, then the system is said to be *completely decomposable*, as is the case for the example above. However, in term of software systems, such complete independence is rarely found because of the degree of coupling between modules (shared variables, side effects, etc.). If the independence constraint is relaxed slightly, *nearly decomposable systems* are obtained, exhibiting a few interactions, but still allowing efficient encoding to take place.

The second phase of the chunking process is encoding or abstraction. If the aggregates are independent then the encoding process is particularly straightforward since the abstraction mechanism can be applied independently to each. The presence of some interaction between components may cause the propagation of symbolic constraints produced by the chunking of one aggregate to others. In a nearly decomposable system this propagation should be relatively local in nature. The encoding of the aggregates above can be expressed as

```
0101  1010  1100
 |     |     |
 5     A     C
```

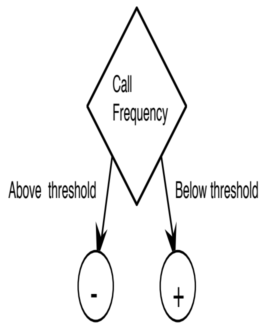
The third phase involves the storage of the chunked aggregates in a knowledge structure that enables its future use in a problem solving activity. It is assumed that the collection of chunked aggregates can be described in a hierarchical manner.

In PM, the first phase of the chunking process, the decomposition process, is supported by the DPA system. The second phase, the abstraction process, is supported by the TUNE system. The third phase, storage and retrieval, is performed by the Reuse Library System. What is actually being chunked is a function of the granularity at which the process is currently working. At the highest level of granularity the goal is to isolate independent systems. At the procedural level the goal is to identify collections of independent procedures within a larger system. At the code level the goal is to extract aggregates of code found within a procedure. The detailed procedure for extracting code aggregates has been implemented and is described in [Reynolds, Maletic 90]. The remainder of this section will focus on how the process is currently implemented at the procedural level.



5.3.2. Quality Model for Dynamic Reuse.

The identification of chunks is also supported by a quality model. The current model is given in figure 5.5. As in the SRC subsystem, the learning component will consist of building a decision tree inductively based upon assessing the structure and behavior of positive and negative examples provided by the users. This decision tree will operationalize the user's notion of independence. For our purposes let us use the simple tree given below:



The call frequency metric for a hypothesized collection (chunk) of procedures, x, is the ratio of the total number of calls made to the procedures that are members of the hypothesized chunk, divided by the total number of calls made to x (intended as calls to the root of the subgraph corresponding to the chunk). This assesses the cohesiveness of action within the code. If no calls are made to components of a chunk (other than the root) from outside the chunk, the value for the call frequency is 1, and the collection exhibits high cohesiveness.

For a given software system the set of possible chunks correspond to each of the possible connected subgraphs present in the call graph. The null hypothesis here is that only the whole system itself can be chunked according to the user supplied criteria. This corresponds to chunking the whole system with the main module as the root. It is considered the null hypothesis since the system had already been accepted for reuse in a static manner. The goal here is to identify collections of component procedures that satisfy the user's criteria. In this case the criterion is that acceptable collections must exhibit a value for the call frequency metric that is below a certain threshold, in our example it is fixed to be two.

The current version of the DPA subsystem is able to automatically generate the collection of procedures that satisfy a given constraint in the decision tree. Additional information can be found in [Wang 91]. The pseudocode for the DPA main module is given in figure 5.7.

```
Procedure DPA();
repeat
    1.READ PASCAL SOURCE PROGRAM;
    2.MODIFY SOURCE PROGRAM (INSERT "WRITE STATEMENTS") IN ORDER TO TRACE
      ROUTINE EXECUTION PATH;
    3.COMPILE THE MODIFIED PROGRAM
    4.EXECUTE THE MODIFIED PROGRAM (EXECUTION HISTORY IS GENERATED);
    5.GENERATE A DESCRIPTION OF THE CALL GRAPH FOR THE SYSTEM. LET EACH
      PROCEDURE IN THE GRAPH REPRESENT A HYPOTHESIS THAT IT AND ALL
      PROCEDURES CALLED BY IT SHOULD FORM A CHUNK;
    6.STORE EXECUTION HISTORY DATA WITH EACH HYPOTHESIZED CHUNK;
    7.DERIVE APPROPRIATE STATISTICAL METRICS FOR EACH HYPOTHESIS BASED
      UPON
      EXECUTION HISTORY DATA;
    8.PERFORM A PREORDER TRAVERSAL OF THIS TREE OF HYPOTHESES TO EXTRACT
      ACCEPTABLE CHUNKS;
    9.FOR EACH SELECTED AGGREGATE, STORE APPROPRIATE INFORMATION IN A
      FRAME.STORE THESE FRAMES AND ASSOCIATED CODE IN A FILE FOR BY THE
      TUNE SYSTEM.UNTIL (ALL SUBMITTED SYSTEMS TESTED);
end
```

Figure 5.7. Pseudocode for DPA.

```
Object : object name
**** Object Global Variables ****
Global variable1: Type
Global variable2: Type
......................
******* Object Interfaces *******
Routine-Type-Qualifier: function/proc.
Routine-Name: function/procedure name
Routine-Result-Type:  Type
Formal-Parameter-List:
    input variable:  Type  . . .
    output variable: Type  . . .
******** Object Contents ********
...........................
```

Figure 5.8.
The Template Description of a Candidate Code Object Produced by the DPA System

Once an acceptable aggregate is identified, the system prepares a frame that describes its interfaces in term of name, type, parameter list, as well as any external data or functions that it requires. The template used to generate each frame is given in figure 5.8.

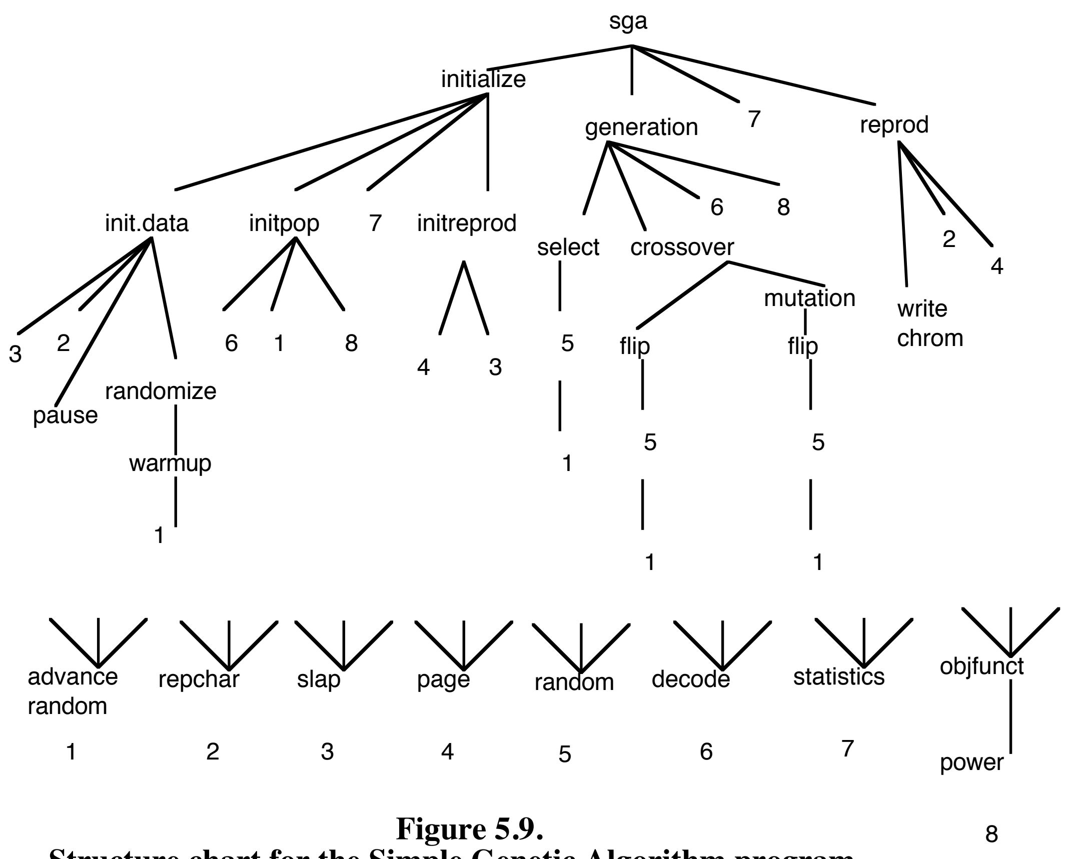


Figure 5.9.
Structure chart for the Simple Genetic Algorithm program

5.3.3. Maintenance and DPA.

The DPA agent is useful for many different maintenance situations. In the case of a corrective maintenance problem DPA can be used to assess the impact of a code modification. For example if a code modification is made to a procedure it is important to know what procedures are related to the modified one. This type of problem is demonstrated by an example using a small program. The procedural structure for an example Pascal program, Simple Genetic Algorithm (SGA) system taken from [Goldberg 89], is given in figure 5.9. In order to simplify the diagram, procedures that are called by several others are indexed by integers in the diagram, and the names of the indexed procedures are given. The Pascal source code for the SGA system is given in appendix C.

Based upon the input the system calculates the appropriate metrics required by the currently active decision tree. In this case it is just call frequency. The collected statistics averaged over all of the runs, for the SGA example, are given in table 5.1.

objects	called together	all calls	call frequency
initialize	1	21689	21689.00
initdata	1	274	274.00
randomize	1	192	192.00
warmup	1	191	191.00
initpop	1	21581	21581.00
flip	**10050**	**10541**	**1.05**
random	**10350**	**10353**	**1.0003**
objfunc	**330**	**330**	**1.00**
initreport	1	14	14.00
generation	10	31031	3103.10
select	300	10541	35.14
crossover	150	29591	197.27
mutation	9000	20591	2.29
report	10	686	68.60

Table 5.1.
Selection of reusable objects for the Simple Genetic Algorithm Program

Each hypothesized chunk is labeled by its root node. The average number of calls made to the root of the chunk per run is then given followed by the total number of calls per run made to all the procedures of the chunk by any procedure of the system. The call frequency, as previously defined, is shown next. In this table the statistics for the leaf nodes are not included, since those correspond to isolate chunks of single elements. The selected hypotheses are those with a call frequency value lower than 2. The positioning of these hypothesized collections in the call graph for the SGA is given in figure 5.10. Note that these aggregates are all positioned at low levels in the call graph. This is due to the particular nature of the analyzed system, since many procedures are not used in a single context, but are called from different modules. Several of the possible chunks do not present a single entry point.

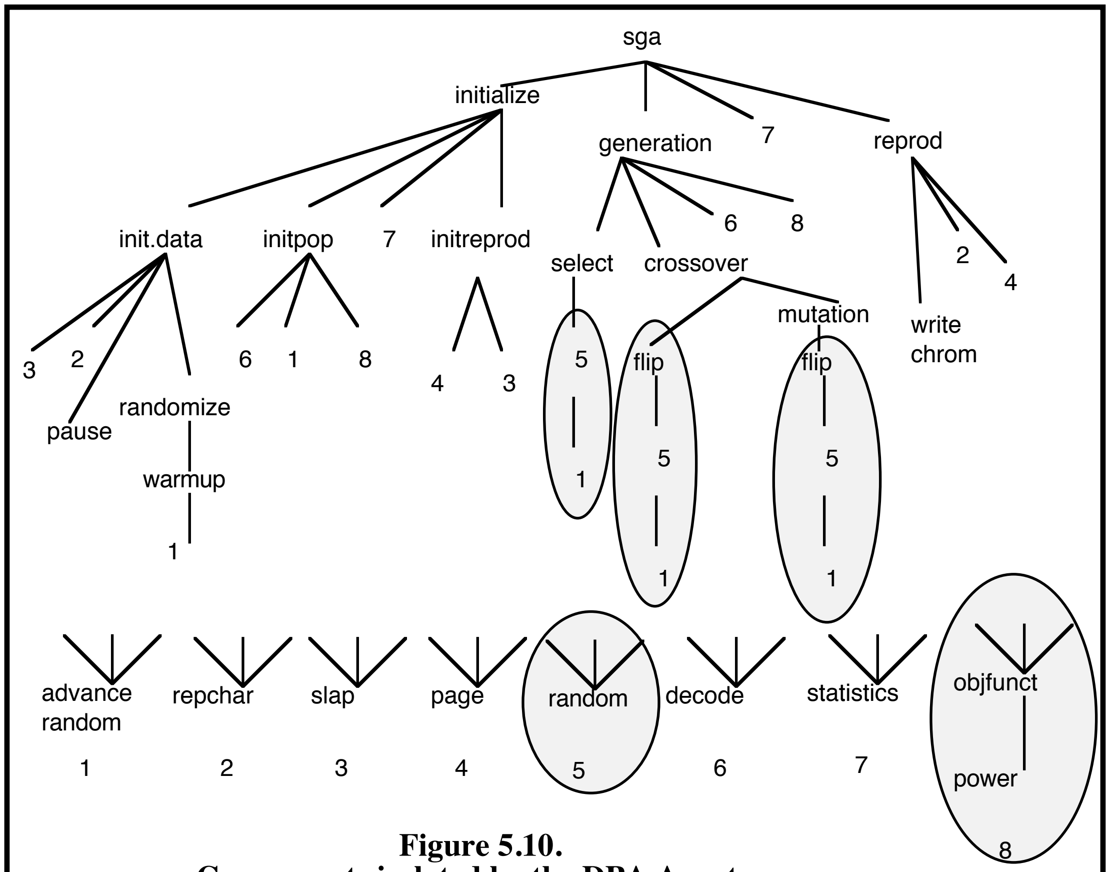


Figure 5.10.
Components isolated by the DPA Agent.

A description of the frame produced by the system for the FLIP module is given in figure 5.11. This frame is automatically placed in a file for future reference by the TUNE system. A modification to any of these procedures would therefor require an impact analysis of all of the procedure included in the chunk.

The DPA can be used to isolate components or groups of components that need to be modified in a adaptive or perfect maintenance situation. In an adaptive situation there are many time a set of interface routines which need to be maintained to account for the change in the environment.

```
======= flip =======
global  jrand           : integer
global  oldrand   :array[1..55] of real
procedure_type_qualifier      : function
procedure_name          : flip
procedure-result-type   : boolean

function flip(probability:real):boolean;
begin
  if probability = 1.0  then  flip := true
  else flip := (random <= probability);
end;
function random : real;
begin
  jrand := jrand + 1;
  if (jrand > 55) then begin
    jrand := 1;
    advance_random;
  end;
  random := oldrand[jrand];
end;
procedure advance_random;
var   j1         : integer;
      new_random : real;
begin
  for j1 := 1 to 24 do begin
    new_random := oldrand[j1]- oldrand[j1+31];
    if (new_random <0.0) then
      new_random := new_random + 1.0;
    oldrand[j1] := new_random;
  end;
  for j1 := 25 to 25 do begin
    new_random:=oldrand[j1] - oldrand[j1-24];
    if (new_random <0.0) then
      new_random := new_random + 1.0;
    oldrand[j1] := new_random;
  end;
end;
```

Figure 5.11.
The description of a candidate code object produced by the DPA subsystem

5.4. The Task Understanding Agent.

The TUNE agent produces a generalized description of a given component. The agent is activated when the blackboard contains the information describing the static and dynamic reusability of the component.

The TUNE agent automatically learns classification rules for the reuse library given a set of facet values and an example component. The rules are used to classify candidate modules in terms of a faceted classification scheme. The faceted classification scheme involves identifying objects in the domain to be classified. These objects have attributes which constitute the terms of the facets [Ranganathan 67]. Facets are parameters of the classification scheme. They represent the indexing keys for a given classification. The values that are associated with the facets differentiate the objects that are being classified. The classification of a module is an abstract representation in the form of facet-value pairs. A facet value pair consists of a facet and a list of attributes possessed by the module for that facet. Each facet expresses some fundamental functional characteristic of any module, with respect to a reuse perspective. For each facet there is a predetermined set of possible terms representing possible attributes of the object to be classified.

The inputs to the system are the facet values to be concluded and an example program, or programs. For example, Facets may be: sort_array or push_stack. The example program contains the software modules that support the facet being defined to the system. The goal of the learning activity here is to identify those parts of the program that are relevant to the concept to be learned. Commercially available program modules were used to test the system to create a real-world scenario. These program modules are object classes that are available with Borland's Turbo C++ version 1.00. Because of the object-oriented nature of the examples, it was logical for a part of the example (program module) to refer to facet value pairs. Also, as in its future use, different versions of the same facet value pairs were introduced.

As an example, assume that the input to the system is the list program in figure 5.13, and the facet to be learned is 'put_list', the output of the system would be the program segment in figure 5.14. This segment or segments contains those textual elements that pertain to the specific inference of the goal concept. At this point, a post processing operation would condense this information for the rule. The conclusion will be the concept to be learned, push-stack.

```
// list.cpp:Implementation of the List Class
#include <iostream.h>
#include "list.h"

int List::put_elem(int elem, int pos)
{
   if (0 <= pos && pos < nmax)
   {
      list[pos] = elem;    // Put an element into the list
      return 0;
   }
   else
      return -1;           // Non-zero means error
}

int List::get_elem(int elem, int pos)
{
   if (0 <= pos && pos < nmax)
   {
      elem = list[pos];    // Retrieve a list element
      return 0;
   }
   else
      return -1;           // non-zero means error
}

void List::print()
{
   for (int i = 0; i < nelem; ++i)
      cout << list[i] << "\n";
}
```

Figure 5.12.
An example input file for the TUNE Agent.

5.4.1. Implementing the Rule Learning.

A Cultural Algorithm [Reynolds 93] is a dual-inheritance system that is useful for exploring large search spaces. There is macro evolutionary process that occurs through changes in a Belief space and micro evolutionary process that occurs through changes in trait sequences. Trait sequences represent individual examples that help drive the system to conclusive beliefs, while the belief system manages the domain of current beliefs, and directs the generation of examples.

```
int List::put_elem(int elem, int pos)
{
   if (0 <= pos && pos < nmax)
   {
      list[pos] = elem;    // Put an element into the list
      return 0;
   }
   else
      return -1;           // Non-zero means error
}
```

Figure 5.13.
The output corresponding to input of figure 5.12 and facet: function = put_list

```
Procedure VGA
begin
  t=0;
  Read in Number.of.Factors;
  Initialize POP(t);
  Initialize Boundary.set(t);
  For i=1 to Number.of.Factors do
   Initialize Graph(Factor(i));
  Evaluate Structures in POP(t);
  For i=1 to Number.of.Factors do
   VIP(POP(t), Graph(Factor(i)));
   While Vote.change.any.factor do
   begin
     t=t+1;
     Select POP(t) from POP(t-1);
     Recombine structures in POP(t);
     Evaluate structures in POP(t);
     For i=1 to Number.of.Factors do
     VIP(POP(t), Graph(factor(i)))
     Update.Boundary
end.
```

Figure 5.14.
The Version Space Controlled Genetic Algorithm

The specific implementation of the Cultural Algorithm in this project is called the Version Space Guided Genetic Algorithm (VGA). The algorithm for the VGA is outlined in figure 5.15. Genetic algorithms are used to represent the population space, and Version spaces are used to represent the Belief Space. The formal properties of VGA's are discussed elsewhere [Reynolds, Maletic 93, Reynolds 94].

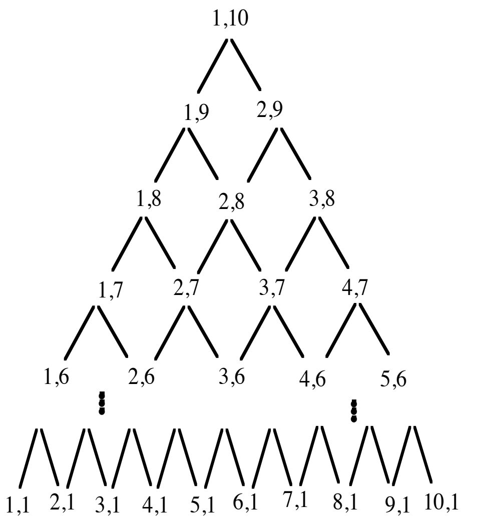


Figure 5.15.
The representation of the belief space.

5.4.2. Representing the Learning Problem.

The rule learning task is divided into three phases. In the first phase, the learning system attempts to identify portions of the code example that support the concept to be learned. In the second phase, a post processor extracts a basic set of syntactic elements from the selected portion of the code. These terms are then conjoined to form the preconditions for a rule that has the concept as a conclusion. Here we focus on the first phase which is the most complex of the three.

The belief or hypothesis space can be viewed as a lattice of line ranges in the program. Each specific hypothesis can be viewed of as a tuple <s, c>. s represents the starting line number for the segment, and c represents the number of subsequent lines contained in the code segment. The lattice for the set of possible sequences in a 10 line program is given in figure 5.17. The corresponding program is given in figure 5.16. The goal of the search is to find those subsequences that support the concept to be learned. It is expected that there will be several disjunctive sequences that support a given concept. The system needs not find them all, in order to produce sufficient information for a rule. In fact, the goal will be to find the largest of the several disjuncts that might be present in the code. If the code is well structured this is reasonable, since code that functions together should be grouped together within the program body.

```
1.   int Stack::push(int elem)
2.   {
3.     int m = getmax();
4.      if (top < m)
5.      {
6.          put_elem(elem,top++);
7.           return 0;
8.       }
9.       else
10.          return -1;
11.  }
```

Figure 5.16.

Program Segment with line numbers

The trait or population space can be represented as a collection of chromosomes or binary vectors, each of length equal to the number of lines in the code. A '1' in the i^{th} position of a chromosome signifies that there is evidence for the concept in line i. A '0' means otherwise. For the program given in figure 5.16 a possible chromosome would be:

000001111100

Note that if we only specify consecutive sequences, the we can represent a chromosome as a tuple <s,c> as done in the Belief space.

The performance function for the chromosome is defined over a binary set. A '1' is returned if there is a complete or partial match between the code and the decision of an oracle or expert, '0' otherwise. This approach uses the domain knowledge of the expert to guide the process. The assessment does not require a detailed decision, and there is a limit to the number of assessments that an expert can make for a run.

5.4.3. The Learning Loop.

The basis for the learning activity is the Vote-Inherit-Promote cycle. Individuals in the population are evaluated according to the performance function. These individuals are then associated with hypotheses in the current Version space that they represent. Mitchell's candidate elimination algorithm is used to keep track of the current version space. The current space is described in terms of an S set and a G set. The G set consists of those hypotheses that are maximally general, that is they are not contradicted by any negative examples presented so far. At the beginning of the run for the example space in figure 5.15, the G set is the root hypothesis, since there are no negative examples that rule out specific lines as evidence. The S set consists of generalizations about the positive population examples. New examples can move the G set down if they are negative, and the S set up if they are positive.

The population of examples is first used to update (vote) the current S and G sets. Then each hypothesis in the S and G sets are given a weighted score based upon the examples that have been generated so far to support them. This score is then inherited by the population elements associated with them. Details of the scoring scheme can be found in Posner's work [Posner 94]. These scores are then used to produce a new population via the probabilistic application of the standard genetic operators of mutation and crossover to copies of successful individuals.

5.4.4. A Maintenance Problem in TUNE.

The TUNE agent identifies code objects via a set of rules. In solving a maintenance problem, program understanding is a major part of the problem solving process. In chapter 4 a perfective maintenance problem is proposed in which new capabilities are needed for the Airline system. Location of the procedures that address the issues of the maintenance problem can be located using the TUNE system. One such procedure that is located via TUNE address the make_reservations procedure.

5.5. The Reuse Library Agent.

The purpose of the Reuse Library Agent [Wickey 94] is to store code components that are deemed reusable, by the blackboard, under a given criterion. The agent also needs a description of the component which is also supplied on the blackboard.

The code components are stored such that they are easily identified and retrieved in a reuse situation. The indexing structure used to store and retrieve components stores similar code components near one another. This allows the quires, based on the index, to retrieve a set of components that address related problems. The Reuse Library Agent employs an incremental conceptual clustering learning algorithm [Fisher 90] to build the index. The index is based on a faceted [Prieto-Diaz 87] representation of the code components.

In the learning phase of this agent, a set of classified components are presented to the agent. Conceptual clustering is performed on the classified components and an index is developed. This section presents a brief description of conceptual clustering and some results produced by the Reuse Library Agent.

5.5.1. Conceptual Clustering.

Conceptual clustering is concerned with the concepts being represented by the clusters rather than the relationships between individual objects [Michalski 83, Fisher 90, Michalski 90]. The goal of a conceptual clustering system is to generate a classification based on the input objects where all the objects in a group represent an example of a particular concept. The grouping represent intra-group similarity and inter-group dissimilarity. The Reuse Library Agent employs a specific incremental conceptual clustering algorithm, COBWEB [Fisher 90], that performs a hill-climbing search through a space of hierarchies as each new example is added. The quality criterion that is employed is category utility from library science.

Category utility comes from the study of the quality of cognitive abilities of humans and is a statistical measure originally used as a means of predicting the basic level in human classification hierarchies [Gluck 85]. Basic level categories are those that are most readily retrieved by humans because the are not too general nor too specific. That is, high inter-group similarity and high inter-class dissimilarity. Category utility is computed on partitions whose objects are represented as vectors of attribute values. Measures of inter-group and inter-group similarity are represented as probabilities over the examples [Fisher 90]

COBWEB performs conceptual clustering by building a hierarchical classification tree based on the input examples. Each internal node of the tree represents a group of objects that share some similarities. Children of a node represent a partitioning of a group into a number of subgroups. Leaf nodes represent individual examples (objects). The root node represents all objects being classified. The tree is build incrementally, that is, for each new example the tree is adjusted to include that object. Figure 5.17 is the COBWEB algorithm.

5.5.2. Classification Model.

The input to the Reuse Library Agent is provided as facet [Prieto-Diaz 87] vectors that describe the code components. Prieto-Diaz proposes using facet vectors containing six facets. Six independent views of software objects will provide adequate information for reuse [Prieto-Diaz 85].

```
Function COBWEB(object, root)
1) Update counts of the root.
2) IF root is a leaf
   Then return the expanded leaf to accommodate the new object.
   Else find the child of root that best hosts object and perform
      one of the following.
      a) Consider creating a new class and do so if appropriate.
      b) Consider node merging and do so if appropriate and call
         COBWEB(object, merged-node).
      c) Consider node splitting and do so if appropriate and call
         COBWEB(object, root)
      d) if none of the above were performed call COBWEB(object,
         best-child-of-root).
```

Figure 5.17.
COBWEB Algorithm [Fisher 90]

Faceted classification is a ground up, rather than top down (enumerative classification) approach. A facet is an attribute or elemental of a set of objects. Facets may also be considered as perspectives, viewpoints, or dimensions of a particular domain [Prieto-Diaz 87]. In faceted classification, a group of facets is defined that adequately describes the objects to be classified. Individual objects can be identified by assigning specific values to each of the facets. These facet values are then combined into a facet vector that identifies the objects.

Function	**Object**	**Medium**	**System Type**	**Functional Area**	**Setting**
add	arguments	array	assembler	accounts pay	advertising
append	arrays	buffer	code gen.	accounts rec.	appliances
close	backspaces	cards	code optim.	analy. struct.	sales

compare	blanks	disk	compiler	auditing	association
complement	buffers	file	DB manag.	job control	auto repair
compress	characters	keyboard	file handler	billing	barbershop
create	descriptors	line	file handler	bookkeeping	cable station

Table 5.2.
Facets and values used in Library Agent.

The facets used by the Reuse Library agent are: function, object, medium, system type, functional area, and setting (see table 5.2). This is the vector format recommended by Prieto-Diaz and Freeman [Prieto-Diaz 87]. Though each facet in the agent does not take into account the ordering of the facets, rather it weights each of the facets equally when performing storage and retrieval.

5.5.3. Maintenance and the Reuse Library Agent.

RLS constructs the reuse library in a manner which is conducive to software maintenance. Retrieval of a set of components which address similar problems directly supports many maintenance problem solving situations. In the case of a corrective maintenance problem, suggestions for fixes can be made by retrieving components from the library which address the particular or related problem. In an adaptive situation, canned fixes can be developed and inserted into the reuse library.

6. Conclusion and Future Work.

There is a large number of methodologies for the development of software systems. These methodologies focus on improving the quality of the product and the cost effectiveness and efficient of development. Though the quality of the product directly relates to maintainability of that software system, these development methodologies are not directed to maintenance problems. Since the most costly venture in the software process occurs in the maintenance of existing software system, a methodology which focuses primarily on the maintenance problem is necessary.

Because of the diverse nature of the maintenance problem solving process, a variety of problem solving paradigms must utilized. Maintenance problems include corrective, perfective, adaptive, and preventive type problems. Each of these maintenance types requires different problem solving methods. Trying to apply the same type of problem solving paradigm to the diverse set of maintenance types and problems is not cost effective. A methodology which utilizes a variety of problem solving techniques is needed in the case of software maintenance.

The Software Service Bay is a knowledge based methodology for solving software maintenance problems. The Service Bay's black board architecture allows for a variety of problems solving systems to be used in conjunction opportunistically. By identifying what type of maintenance problem is at hand specialized knowledge sources can be accessed to attack the specific type. The analogy between the Software Service Bay and the automotive service bay serves to demonstrates the variety of problems and solution types for maintenance. There are many different auto mechanic experts, each with specific knowledge on different maintenance problem, be they transmission, brakes, engine, etc. Each mechanic solves problem with their domain using different knowledge, tools, and methods. Software maintenance is much the same way. The variety of problem types (perfect, corrective, etc.) lends to a specific set knowledge, specific set of tools, and specific problem solving methods for each type.

The Service Bay is an ideal framework to incorporate process improvement automation. As maintenance problems are encountered, the solution methods applied and the results of those applications can be monitored and later analyzed. The analysis of the solution process will suggest possible better methods or necessary knowledge to improve the process. The Service Bay framework acts as a laboratory to conduct experiments on problem solving. As new specialists are added to the Service Bay their utility to the overall maintenance process must be assessed. Improvement to the software maintenance process is an ongoing cycle which requires experimentation, assessment, and modification of the existing problem solving methods and knowledge sources.

Future research on the Software Service Bay will include applying and testing this framework to large scale real world problems. The Service Bay should be conducive to a group type working environment and therefor be readily usable in an commercial setting.

Extending this type of framework to an integrated development and maintenance environment is also of interest. Development methodologies can be used to solve particular maintenance problems. Therefore, will a framework which is intended for development and maintenance prove more practical. A development and maintenance environment which explicitly maps the formal concepts of development to the implementation is necessary to facilitate cost effective maintenance.

Appendix A. System Specification for the Airline Reservation System [Berzins 91].

```
MACHINE final_travel_agent_interface
  INHERIT final_travel_agent_view
  INHERIT airline_reservation_state_model

STATE(reservations: set{reservation},
      schedule: map{flight_id, flight})
INVARIANT existing_flights(reservations),
          no_overbooking(reservations),
          single_reservations(reservation)
INITIALLY reservations = {}, domain(schedule) = {}
   --Initially ther reservation set and the schedule are empty.

MESSAGE find_flights( origin destination: airport)
   --G1.1.1, G1.1.2
REPLY flights(s: sequencee{flight_description))
   --Flights from the origin to dest.
  WHERE ALL(f: flight :: description(f) IN s <=>
            f In range(schedule) & origin(f) = origin &
            destination(f) = destination )

MESSAGE reserve(a:agent_id, i: flight_id, d:date,
                p: passenger)   --G1.2.
  WHEN i IN scchedule &
            bookings(i,d,reservations) < capacity(schedule[i]) &
            ~holds(p,i,d,*reservations)  --Seat available.
    CHOOSE (r:reservations SUCH THAT
            passenger(r) = p & id(r) = i & date(r) = d &
            agent(r) = a )
    REPLY done TRANSITION reservations = *reservations U {r}
       --Add reservation
  WHEN holds(p,i,d,*reservations)
    REPLY EXCEPTION reservation_exists
  WHEN ~(i IN schedule)  --Unknown flight.
    REPLY EXCEPTION no_such_flight
  OTHERWISE REPLY EXCEPTION no_seat
MESSAGE cancel(i: flight_id, d: date, p: passenger) --G1.3
  WHEN i IN schedule & holds(p,i,d,*reservation) --Res. found
    CHOOSE(r: reservation SUCH THAT
           r IN * reservations & passenger(r) = p &
           id(r) = i & date(r) = d)
    REPLY done TRANSITION reservations = *reservations - {r}
       --Remove reservation
  WHEN ~(i IN schedule) REPLY EXCEPTION no_such_flight
  OTHERWISE REPLY EXCEPTION no_reservation

MESSAGE notify(a: agent, i: flight, d: date) --G1.4.
  WHEN i IN schedule  --Respond to flight cancellation.
    REPLY reservations(s: sequence{reservation})
      WHERE ALL (r: reservation :: r IN s <=>
                 r IN reservations & id(r) = i & agent(r) = a &
                 date(r) >= d)
  OTHERWISE REPLY EXCEPTION no_such_flight
```

```
CONCEPT holds(p: passenger, i: flight_id, d: date,
              rs: set{reservation})
  VALUE(b: boolean)
  WHERE SOME(r: reservation SUCH THAT r IN rs ::
            passenger(r) = p & id(r) = i & date(r) = d)

CONCEPT description(f:flight) VALUE(fd: flight_description)
  WHERE fd= [id:: id(f), dep:: departure(f),
            arr:: arrival(f), price:: price(f) ]

CONCEPT flight_description: type
  WHERE flight_description =
        tuple{id:: flight_id, dep arr:: time, price:: money}
END
```

Figure A.1
Travel Agent Interface [Berzins 91].

```
DEFINITION airline_reservation_state_model

CONCEPT existing_flights(s: set{reservation})
  VALUE(b:boolean)
  WHERE ALL(r: reservation SUCH THAT r IN s ::
            id(r) IN schedule)

CONCEPT no_overbooking(s: set{reservation})
  VALUE(b:boolean)
  WHERE ALL(i: flight_id, d:date SUCH THAT I IN SCHEDULE ::
            bookings(i, d, reservations) <=
            capacity(schedule[i]))

CONCEPT bookings(i: flight_id, d;date, rs:set[reservation})
  VALUE(n: nat)
  WHERE n = NUMBER(r: reservation SUCH THAT
                   r IN rs & id(r) = i & datae(r) = d :: r)

CONCEPT single_reservation(s: set{reservation})
  VALUE(b: boolean)
  WHERE ALL(r1 r2 : reservation SUCH THAT r1 IN s & r2 IN s
            :: id(r1) = id(r2) & date(r1) = date(r2) &
            passenger(r1) = passenger(r2) => r1=r2 )
END
```

Figure A.2
State Model Concepts [Berzins 91].

```
MACHINE final_airline_manager_interface
  INHERIT final_airline_manager_view
  INHERIT airline_reservation_state_model

STATE(reservations: set{reseration},
      schedule: map{flight_id, flight} )
INVARIANT existing_flights(reserations),
          no_overbooking(reservations),
          single_reservation(reservations)
INITIALLY reservations = {}, domain(schedule) = {}

MESSAGE add_flight(i: flight_id, price : money,
                   origin destination : airport,
                   departure arrival: time, capacity : nat )
  WHEN ~(i IN schedule)   --New flight
    CHOOSE(f: flight SUCH THAT id(f) = i & price(f) = price &
           origin(f) = origin & destination(f) = destination &
           departure(f) = departure & arrival(f) = arrival &
           capacity(f) = capacity )
    REPLY done TRANSITION schedule = bind(i,f,*schedule)
       --Add flight
  OTHERWISE REPLY EXCEPTION flight_exists

MESSAGE drop_flight(i: flight_id)
  WHEN i IN schedule     --Flight exists
    REPLY done
    TRANSITION schedule = remove(i,*schedule)
      ALL(r: reservation :: r IN reservations <=>
          r IN * reservations & id(r) ~= i )
  OTHERWISE REPLY EXCEPTION no_such_flight

MESSAGE new_fare(i:flight_id, price: money)
  WHEN i IN schedule
    CHOOSE(f: flight SUCH THAT id(f) = i & price(f) = price &
           origin(f) = origin(*schedule[i]) &
           destination(f) = destination(*schedule[i]) &
           departure(f) = departure(*schedule[i]) &
           arrival(f) = arrival(*schedule[i]) &
           capacity(f) = capacity(*schedule[i]) &
    REPLY done TRANSITION schedule = bind(i,f,*schedule)
  OTHERWISE REPLY EXCEPTION no_such_flight

END
```

Figure A.3
Airline manager Interface [Berzins 91].

```
MACHINE travel_agent(id: nat)
  INHERIT final_travel_agent_view HIDE time money
  INHERIT travel_agent_command_formats
  INHERIT travel_agent_output_formats
  IMPORT flight_description FROM travel_agent_interface

MESSAGE interpret_command(command : string)
   --Command from travel agent's keyboard
  WHEN is_find_flights(edit(command), origin, destination)
    SEND find_flights(origin destination: airport)
      TO airline_reservation_system
  WHEN is_reserve(edit(command),a,i,d,p)
    SEND reserve(a:agent_id, i:flight_id, d:date, p:passenger)
      TO airline_reservation_system
  WHEN is_cancel(edit(command),i,d,p)
    SEND cancel(i:flight_id, d:date, p:passenger)
      TO airline_reservation_system
  WHEN is_notify(edit(command),a,i,d)
    SEND notify(a:agent_id, i:flight_id, d:date)
      TO airline_reservation_system
OTHERWISE REPLY(s:string) WHERE s= "Command not recognized"

--Normal responses.

MESSAGE flights(sf: speuence{flight_description})
  WHEN sf ~=[]
    CHOOSE(rows: sequence{row} SUCH THAT
           length(rows) = length(sf) &
           ALL(i: nat SUCH THAT i IN domain(sf) ::
               rows[i] = flight_row(sf[i]) ))
    SEND(lines: sequence{string}) TO display
      WHERE lines = table(rows, [6,8,8,10])
  OTHERWISE SEND(s:string TO display
    WHERE s="No flights found"

MESSAGE done
  SEND(s:string) TO display WHERE s="Done"

MESSAGE reservations(sr: sequence{reservation})
  WHEN sr~= []
    CHOOSE(rows: sequence{row} SUCH THAT
           length(rows) = length(sr) &
           ALL(i: nat SUCH THAT i IN domain(sr) ::
               rows[i] = reservation_row(sR[i]) ))
    SEND(lines: sequence{string}) TO display
      WHERE lines = table(rows, [8,20])
  OTHERWISE SEND(s:string TO display
    WHERE s="No reservations found"

CONCEPT edit(s:string)
  VALUE(es:string)

END
```

Figure A.4
Command Interface for the Travel Agent [Berzins 91].

```
DEFINITION travel_agent_command_foramts
  INHERIT airline_reservation_type_formats
  IMPORT is_list FROM format  --Standard formatting concepts.

  CONCEPT is_find_flights(command: string
                          origin destination : airport)
    VALUE(B:BOOLEAN)
    WHERE b <=> is_list(command, "f", orignin, distination)

  CONCEPT is_reserve(command: string, a:agent_id, i: flight_id,
                     d: date, p: passenger)
    VALUE(b: boolean)
    WHERE b<=> SOME(ds: string ::
                    is_list(command, "r", a,i,ds,p) &
                    date(ds) = d)

  CONCEPT  is_cancel(command: string, i:flight_id, d: data,
                     p: passenger)
    VAULE(b: boolean)
    WHERE b <=> SOME(ds: string ::
                     is_list(command, "c", a,i,ds,p) &
                     date(ds) = d)

  CONCEPT  is_notify(command: string, i:flight_id, d: data,
                     p: passenger)
    VAULE(b: boolean)
    WHERE b <=> SOME(ds: string ::
                     is_list(command, "n", a,i,ds,p) &
                     date(ds) = d)

END
```

Figure A.5
Travel Agent Command Formats [Berzins 91].

```
DEFINITION travel_agent_output_formats
  INHERIT final_travel_agent_view
  INHERIT airline_reservation_type_formats
  IMPORT flight_description FROM travel_agent_interface
  IMPORT row FROM format  --Standard formatting concepts

  CONCEPT flight_row(f: flight_description)
    VALUE(r: row)
    WHERE r = [f.id, f.dep, f.arr, f.price]

  CONCEPT reservation_row(r :reservation)
    VALUE(r1: row)
    WHERE SOME(ds: string ::
               r1 = [ds, passenger(r)] & data(r) = data(ds)

END
```

Figure A.6
Travel Agent Output Formats [Berzins 91].

```
DEFINITION airline_reservation_type_formats
  INHERIT character_properties
  IMPORT Subtype FROM type

  CONCEPT date(s: string)
    VALUE(d1:date)
    WHERE SOME(m d y: string :: s = [$m, '/', $d, '/', $y] &
               digits(m) & digits(d) & digits(y) &
               1<=nat(m)<=12 & 1<=nat(d)<=31 &
               length(y) = 2 &
               d1 = create@date(nat(d), nat(m), nat(y)) )

  --Format definitions for the types in the travel agent
interface

  CONCEPT flight_id: type
    WHERE subtype(flight_id, string)
      ALL(f: flight_id :: SOME(a n: string :: f= a || n &
          letters(a) & length(a)=2 & digits(n) & length(n) <= 4))

  CONCEPT agent_id: type
    WHERE subtype(agent_id, string)
      ALL(a: agent_id :: letters(a) & length(a)=3)

  CONCEPT airport: type
    WHERE subtype(airport, string)
      ALL(a: airport :: letters(a) & length(a)=3)

  CONCEPT money: type
    WHERE subtype(money, string)
```

```
      ALL(m: money :: SOME(d c: string :: m = ['$', $d, '.', $c]
&
          digits(d) & digits(c) & length(c) = 2) )

  CONCEPT time: type
    WHERE subtype(time, string)
      ALL(t: time :: SOME(h m: string :: t = [$h, 'm', $s] &
          digits(h) & digits(m) & 0 <= nat(h) <= 23 &
          0<=nat(m)<=59 & length(h) >= 1 & length(c) = 2) )

  CONCEPT passenger: type
    WHERE Subtype(passenger, string)
      ALL(p: passenger, c: char SUCH THAT c IN p ::
          letter(c) | c = space),
      ALL(p: passenger :: length(p) > 1 & p[1] ~= space &
                          p[length(p)] ~= space)
    --This format implies the passenger mus be the last
    --argument because passenger names can contain spaces.

END
```

Figure A.7
Concrete Type Definitions [Berzins 91].

```
MACHINE airline_manager
  INHERIT final_airline_manager_view HIDE time money
  INHERIT airline_manager_command_formats

  MESSAGE interpret_command(command: string)
    --Command from airline manager's keyboard
    WHEN is_add_flight(edit(command),i,price,origin,
                       destination, departure,arrival,capacity)
      SEND add_flight(i:flight_id, price:money,
                      origin destination: airport,
                      departure arrival: time, capacity: nat)
      TO airline_reservation_system
    WHEN is_drop_flight(edit(command), i)
      SEND drop_flight(i: flight_id)
      TO airline_reservation_system
    WHEN is_new_fare(edit(command), i, price)
      SEND new_fare(i: flight_id, price: money)
      TO airline)reservation_system
    OTHERWISE REPLY(s: string) WHERE s = "Command not recgonized"

  MESSAGE done
    SEND(s:string) TO display WHERE s = "Done."

  CONCEPT edit(s:string)
    VALUES(es: string)

END
```

Figure A.8
Command Interface for Airline Manager [Berzins 91].

```
DEFINITION airline_manager_command_formats
  INHERIT format

  CONCEPT is_add_flight(command:string, i:flight_id, p: money,
                      o d: airport, dep arr:time, cn: nat)
    VALUE(b:boolean)
    WHERE b <=> SOME(ds cap: string SUCH THAT
                   d=date(ds) & cn=nat(cap) ::
                   is_list(command,"add_flight",i,p,o,ds,
                           dep,arr,cap))
  CONCEPT is_drop_flight(command: string, i:flight_id)
    VALUE(b:boolean)
    WHERE b <=> is_list(command,"drop_flight",i)

  CONCEPT is_new_fare(command: string, i:flight_id, p:money)
    VALUE(b:boolean)
    WHERE b <=> is_list(command,"new_fare",i,p)


END
```

Figure A.9
Airline Manager Command Formats [Berzins 91].

```
MACHINE final_travel_agent
  INHERIT travel_agent(id)
  INHERIT basic_edit

  MESSAGE EXCEPTION no_seat
    SEND(s:string) TO display WHERE s="No seat available."

  MESSAGE EXCEPTION reservation_exists
    SEND(s:string) TO display
    WHERE s="The passenger already has a reservation."

  MESSAGE EXCEPTION no_reservation
    SEND(s:string) TO display
    WHERE s="The passenger does not hold a reservation."

  MESSAGE EXCEPTION no_such_flight
    SEND(s:string) TO display WHERE s="Unknown flight id."

END
```

Figure A.10
Interface for Travel Agent [Berzins 91].

```
MACHINE final_airline_manager
  INHERIT airline_manager HIDE is_add_flight is_drop_flight
                               is_new_fare
  INHERIT final_airline_manager_command_formats
  INHERIT basic_edit

  MESSAGE EXCEPTION flight_exists
    SEND(s:string) TO display WHERE s =
        "The flight is already scheduled." || [newline] ||
        "to chage it, use drop_flight and the add_flight."

  MESSAGE EXCEPTION no_such_flight
    SEND(s:string) TO display WHERE
     s="No such flight was scheduled, check flight id."

END
```

Figure A.11
Interface for Airline Manager [Berzins 91].

```
DEFINITION final_airline_environment
  INHERIT final_travel_agent_view
  INHERIT final_airline_manager_view
  INHERIT system
  INHERIT user

  CONCEPT airline_reservation_system: software_system
    WHERE proposed(airline_reservation_system),
          Uses(travel_agent, airline_reservation_system),
          Controls(airline_reservation_system, reservation)
          Uses(airline_manager, airline_reservation_system)
          Controls(airline_reservation_system, flight)

END
```

Figure A.12
Final Environment Model: Proposed System [Berzins 91].

```
DEFINITION final_travel_agent_view
  INHERIT final_flight_view
  INHERIT user       --defines user_class, uses
  INHERIT business   --defines vendor, customer, sells
  IMPORT Subtype Immutable_instances Indestructible FROM type

  CONCEPT travel_agent: User_class
    WHERE ALL(ta: travel_agent ::
      Subtype(travel:agent, vendor),
      ALL(t:ticket:: SOME(ta: travel_agent :: sells(ta, t))),
      ALL(r: reservation ::
          SOME(ta: travel_agent :: supplies(ta, t)) )

  CONCEPT reservation: type
    WHERE ALL(t: trip ::
              SOME(r: reservation :: Needed_for(r, t)) ),
    Immutable_instance(reservation)

CONCEPT id(r:reservation) VALUE(id: flight_id)
CONCEPT date(r:reservation) VALUE(d:date)
CONCEPT passenger(r:reservation) VALUE(p:passenger)
CONCEPT agent(r:reservation) VALUE(a:agent_id)

CONCEPT agent_id: type

CONCEPT ticket: type
  WHERE Subtype(ticket, product),
    ALL(t:trip :: Needed_for(ticket, t)),
    Immutable_instance(ticket),
    indestructible(ticket)

CONCEPT trip:type WHERE Subtype(trip, activity)
CONCEPT origin(t: trip) VALUE(a: airport)
CONCEPT destination(t:trip) VALUE(a:airport)

CONCEPT passenger: type
  WHERE Subtype(passenger, customer),
    ALL(p:passenger :: SOME(t:trip :: wants(p,t))),
    ALL(p:passenger :: SOME(t:trip :: buys(p,t))),
    ALL(p:passenger :: Needed_for(flight, t)

END
```

Figure A.13
Final Environment Model: Travel Agent View [Berzins 91].

```
DEFINITION final_airline_environment
  INHERIT final_flight_view
  INHERIT user

  CONCEPT airline_manager: User_class
    WHERE ALL(am: airline_manager ::
                uses(am, airline_reservation_system) ),
    Maintains(airline_manager, flights)

END
```

Figure A.14
Final Environment Model: Airline Manager View [Berzins 91].

```
DEFINITION final_flight_environment
  INHERIT time
  INHERIT location
  INHERIT period
  IMPORT Subtype FROM type
  IMPORT One_to_one from function(flight, flight_id)

  CONCEPT flight: type
    WHERE Subtype(flight, activity)
      ALL(f:flight ::periodic(f) & period9f) =(1 day))
  CONCEPT origin(f:flight) VALUE(a: airport)
  CONCEPT destination(f:flight) VALUE(a: airport)
  CONCEPT departure(f:flight) VALUE(t: time_of_day)
  CONCEPT arrival(f:flight) VALUE(t: time_of_day)
  CONCEPT price(f:flight) VALUE(m: money)
  CONCEPT id(f:flight) VALUE(i: flight_id)
    WHERE One_to_one(id)
  CONCEPT capacity(f:flight) VALUE(n: nat)

  CONCEPT airport: type WHERE Subtype(airport, location)
  CONCEPT flight_id: type

  CONCEPT airline: type
    WHERE Subtype(airline, supplier),
      ALL(f:flight :: SOME(a: airline :: supplies(a,f)))


END
```

Figure A.15
Final Environment Model: Flight Properties [Berzins 91].

Appendix B. Partial Implementation for the Airline Reservation System [Berzins 91].

```
package body travel_agent_pkg is
  use text_pkg_instance;
  use flight_sequence_pkg;
  use reservation_sequence_pkg;

  --Local Declarations go here.

task body travel_agent_type is
  input, output: file_type;
  last: natural;
  buffer, command: text;
begin
  open(input, in_file, input_name);
  open(output, out_file, output_name);
  loop --forever.
    get_line(input, buffer);
    buffer := edit_command(buffer);
    next_item(buffer, command);
    if length(command) = 1
    then interpret_command(char(command, 1), buffer, output);
    else interpret_command(' ',buffer, output); --Illegal
command.
    end if;
  end loop;
exception
  when end_error => put_line(output, "input terminated");
  --All other exceptions here.
  when others => put_line(output, "Exception");
end travel_agent_type;


procedure interpret_command(cmd:character; args: in text;
                              f: in file_type) is

  procedure display_reservation(res: in reservation) is
    put_line(f,res.d, res.p);
  end display_reservation;

  procedure display_reservations is
     new resrevation_sequence_pkg.scan(display_reservation);

  procedure display_flight(fd: in flight_description) is
  begin
    put_line(f,fd.id,fd.dep,fd.arr.fd.price);
  end display_flight;

  procedure display_flights is
     new flight_sequence_pkg.scan(display_flight);
  o,d: airport;  i: flight_id; da: date;
  p: passenger; a: agent_id;  fs:flight_sequence;
  rs: reservation_sequence;  ok: boolean;

begin -- Body of inpterpret command.
  case cmd is
    when 'f' =>
      parse_find_flights(args, o, d, ok);
      if ok then
       airline_reservation_system.find_flights(o, d, fs);
       if length(fs) > 0 then display_flights(fs);
       else put_line(f, "no flights found"); end if;
      else put_line(f, "command not recognized"); end if;
    when 'r' =>
      parse_reserve(args, a,i,da,p, ok);
      if ok then
       airline_reservation_system.reserve(a,i,da,p);
       put_line(f, "done");
      else put_line(f, "command not recognized"); end if;
    when 'c' =>
      parse_cancel(args, i,da,p, ok);
      if ok then
       airline_reservation_system.cancel(i,da,p);
       put_line(f, "done");
      else put_line(f, "command not recognized"); end if;
    when 'n' =>
      parse_notify(args, a,i,da, ok);
      if ok then
       airline_reservation_system.notify(a,i,da,rs);
       if length(rs) > 0 then display_reservations(rs);
       else put_line(f, "no reservations found"); end if;
      else put_line(f, "command not recognized"); end if;
exception
  when no_seat => put_line(f, "NO seat available");
  --Rest of exceptions
  --  ....

end interpret_command;


procedure parse_find_flights(s: in text; o, d:out airport;
                               result: out boolean) is
  line : text:= s;
  item : text;
begin
  next_item(line, item);
  if is_airport(item) then o := item;
    else result :=false; return
  end if;
  next_item(line, item);
  if is_airport(item) then d := item;
    else result :=false; return
  end if;
  next_item(line, item);
  result := length(item) = 0;
end parse_find_flights;


procedure parse_reserve(s: in text; a : out agent_id;
                           i: out flight_id;
                           d: out date; p: out passenger;
                           result: out boolean) is

  --Finds values for a, i, d
  --Does error checking, returns result as false if error.

end parse_reserve;

procedure parse_cancel(s: in text; i: out flight_id;
                       d: out date; p: out passenger;
                       result: out boolean) is

  --Finds values for i, d, p
  --Does error checking, returns result as false if error.

end parse_cancel;

procedure parse_notify(s: in text; a: out agent_id;
                       i: out flight_id; d: out date;
                       result: out boolean) is

  --Finds values for a, i, d
  --Does error checking, returns result as false if error.

end parse_notify;

end travel_agent_pkg;
```

Figure B.1
Partial Implementation of Travel Agent in Ada [Berzins 91].

```
package body airline_manager_pkg is

--Local declarations go here.

task body airline_manager_type is
  input, output: file_type;
  last: natural;
  buffer, command: text;
begin
  open(input, in_file, input_name);
  open(output, out_file, output_name);
  loop --forever.
    get_line(input, buffer);
    buffer := edit_command(buffer);
    next_item(buffer, command);
    interpret_command(ada_string(command), buffer, output);
    end if;
  end loop;
exception
  when end_error => put_line(output, "input terminated");
  --All other exceptions here.
end airline_manager_type;
```

```
procedure interpret_command(cmd: string; args: in text;
                            f: in file_type) is

  i: flight_id; p:money; o,d:airport;
  dep, arr: flight_time; c: natural; ok:boolean;
begin
  if cmd = "add_flight" then
    parse_add_flight(args,i,p,o,d,dep,arr,c,ok);
    if ok then airline_reservation_system.add_flight(i,p,o,

d,dep,arr,c);
       put_line(f,"done");
    else put_line(f, "command not recognized"); end if;
  elsif cmd = "drop_flight" then
    parse_drop_flight(args,i,ok);
    if ok then airline_reservation_system.drop_flight(i);
       put_line(f,"done");
    else put_line(f, "command not recognized"); end if;
  elsif cmd = "new_fare" then
    parse_new_fare(args,i,p,ok);
    if ok then airline_reservation_system.new_fare(i,p);
       put_line(f,"done");
    else put_line(f, "command not recognized"); end if;
  elsif cmd = "retire" then
    airline_reservation_system.retire;
  else put_line(f, "command not recognized"); end if;
exception
  when flight_exists =>
    put_line(f, "The flight is already scheduled.");
    --Rest of exceptions ...
end interpret_command;


procedure parse_add_flight(s: in text; i: out flight_id;
                           p: out money; o, d: out airport;
                           dep, arr: out flight_time;
                           result: out boolean)  is
  --Returns p,o,d,dep,arr
  --Checks for errors, returns result true if ok.

end parse_add_flight;


procedure parse_drop_flight(s: in text; i: out flight_id;
                           result: out boolean)  is
  --Returns p
  --Checks for errors, returns result true if ok.

end parse_drop_flight;


procedure parse_new_fare(s: in text; i: out flight_id;
                           p: out money; result: out boolean)  is
  --Return i,p
  --Checks for errors, returns result true if ok.

end parse_new_fare;
end airline_manager_pkg;
```

Figure B.2
Partial Implementation of AirlineManager in Ada [Berzins 91].

```
package body airline_reservation_system_pkg is

--Local declaration go here.


task body airline_reservation_system_type is
  schedule: flight_schedule;
  reservations: reservation_set;
  has, exists: boolean;
  b,c: natural;
begin
  open(text_string("s"), schedule);
  open(text_string("r"),reservations);

  loop --Forever.
    begin --Exception handler frame.
      select
        accept find_flights(origin, destination: in airport;
                            fs: out flight_sequence ) do
          make_flights(schedule, origin, destination, fs);
        end find_flights;
      or
        accept reserve(a: in agent_id; i; in flight_id;
                       d: in date; p: in passenger) do
          holds(p,i,d,reservations,has);
          if has then raise reservations_exists; end if;
          begin
            bookings(i,d,reservations,b);
            capacity(i,schedule,c);
            if b=c then raise no_seat; end if;
            add(a,i,d,p,reservation);
          exception
            when flight_schedule_pkg.no_such_flight =>
               raise airline_reservation_system_pkg.
                                         no_such_flight;
          end;
        end reserve;
      or
        accept notify(a: in agent_id; i: in flight_id;
                      d: in date; rs: out reservation_sequence)
do
          member(i, schedule, exists);
          if no exists then
            raise airline_reservation_system_pkg.no_such_flight;
          end if;
          make_reservations(reservations,i,d,a,rs);
        end notify;
      or
        accept add_flight(i: in flight_id; price: in money;
                          origin, destination: in airport;
                          departure, arrival: in flight_time;
                          capacity: in natural) do
          member(i,schedule,exists);
          if exists then raise flight_exists; end if;
          add(i,price,origin,destination,departure,
              arrival,capacity,schedule);
        end add_flight;
      or
        accept drop_flight(i: in flight_id) do
          member(i,schedule,exists);
          if exists then remove(i, schedule);
          else raise airline_reservation_system_pkg.
                                          no_such_flight;
          end if;
        end drop_flight;
      or
        accept new_fare(i: in flight_id; price: in money) do
          begin
            new_fare(i,price,schedule);
          exception
            when flight_schedule_pkg.no_such_flight =>
              raise airline_reservation_system_pkg.
                                         no_such_flight;
          end;
        end new_fare;
      or
        accept retire do
          retire_reservation(reservations);
        end retire;
      or
      end select;
    exception when others => null;
    end; --Exception handler frame.
  end loop;
end airline-reservation_system_type;


procedure make_flights(s: in out flight_schedule;
                       o,d: airport;
                       fs: out flight_sequence) is

--add flight to schedule.

end make_flights;


procedure make_reservations(r: in out resrvation_set;
                            i: in flight_id; d: date;
                            a: in agent_id;
                            rs: out reservation_sequence) is

--add reservation to sequence.

end make_reservations;


procedure retire_reservations(rs: in out reservation_set) is
  now: time := clock;
  d: date := create(natural(day(now)),
                    natural(month(now)),
                    natural(year(now)) mod 100);
begin
  remove(d,rs);
end retire_reservations;


task body retire_demon is

  --driver for the temporal event "retire".

end retire_demon;


end airline_reservation_system;
```

Figure B.3
Partial Implementation of Airline Reservation System in Ada [Berzins 91].

Appendix C. Pascal Source Code for Simple Genetic Algorithm (SGA) [Goldberg 89].

```
program sga;

const maxpop    = 100;
      maxstring = 30;
type  allele    = boolean;
      chromosome= array[1..maxstring] of allele;
      individual= record
                   chrom: chromosome;
                   x:real;
                   fitness:real;
                   parent1, parent2, xsite: integer;
                  end;
      population= array[1..maxpop] of individual;
var   oldpop, newpop: population;
      popsize, lchrom, gen, maxgen: integer;
      pcross, pmutation, sumfitness: real;
      nmutation, ncross: integer;
      avg, max, min:real;

{$I utility.sga}
{$I random.sga}
{$I interfac.sga}
{$I stats.sga}
{$I initial.sga}
{$I report.sga}
{$I triops.sga}
{$I generation.sga}

begin
  gen := 0;
  initialize;
  repeat
    gen := gen + 1;
    generation;
    statistics(popsize, max, avg, min, sumfitness, newpop);
    report(gen);
    oldpop := newpop;
  until (gen >= maxgen);
end.
```

Figure C.1
SGA Main Program [Goldberg 89].

```
function objfunc(x:real):real;
const coef = 1073741823.0; {coef to normalize domain}
      n = 10;
begin
  objfunc := power(x/coef, n);
end;



function decode(chrom:chromosome; lbits:integer):real;
var j:integer;
    accum, powerof2:real;
begin
  accum ;== 0.0; powerof2 := 1;
  for j:= 1 to lbits do begin
    if chrom[j] then accum := accum + powerof2;
    powerof2 := powerof2 * 2;
  end;
  decode := accum;
end;
```

Figure C.2
File: interface.sga [Goldberg 89].

```
procedure statistics(popsize: integer;
                     var max, avg, min, sumfitness : real;
                     var pop: population);
var j:integer;
begin
  sumfitness := pop[1].fitness;
  min := pop[1].fitness;
  max := pop[1].fitness;
  for j:= 2 to popsize do with pop[j] do begin
    sumfitness := sumfitness + fitness
    if fitness>max then max := fitness;
    if fitness<min then min := fitness;
  end;
  avg := sumfitness / popsize;
end;
```

Figure C.3
File: stats.sga [Goldberg 89].

```
procedure initdata;
var ch:char; j:integer;
begin
  rewrite(lst);
  clrscr;
  skip(con,9);
  writeln('***SGA Data Entry and Initialization****');
  writeln;
  write('Enter population size---->'); readln(popsize);
  write('Enter chromosome length---->'); readln(lchrom);
  write('Enter max. genrations---->'); readln(maxgen);
  write('Enter crossover probability---->'); readln(pcross);
  write('Enter mutation probability---->'); readln(pmutation);
  pause(5); clrscr;
  randomize;
  nmutation := 0;
  ncross := 0;
end;


procedure initreport;
begin
  writeln(lst,'Initial population maximum fitness= ', max);
  writeln(lst,'Initial population arverge fitness= ', avg);
  writeln(lst,'Initial population minimum fitness= ', min);
  writeln(lst,'Initial population sum of fitness= ', sumfitness);
end;


procedure initpop;
var j, j1: integer;
begin
  for j:= 1 to popsize do with oldpop[j] do begin
    for j1 := 1 to lchrom do chrom[j1] := flip(0.5);
    x := decode(chrom, lchrom);
    fitness := objfunc(x);
    parent1 := 0; parent2 := 0; xsite := 0;
  end;
end;


procedure initialize;
begin
  initdata;
  initpop;
  statistices(popsize, max, avg, min, sumfitness, oldpop);
  initreport;
end;
```

Figure C.4
File: initial.sga [Goldberg 89].

```
procedure writechrom(var out:text; chrom:chromosome;
                     lchrom:integer);
var j:integer;
begin
  for j:=lchrom downto 1 do
    if chrom[j] then write(out,'1');
    else write(out,'0');
end;


procedure reprot(gen:integer);
const linelength= 132;
var j:integer;
begin
  repchar(lst,'-',linelenght); writeln(lst);
  writeln(lst,'Population Report');
  writeln(lst,'Generation ', gen-1);
  writeln(lst,'Generation ', gen);
  for j:= 1 to popsize do begin
    write(lst,j:2,') ');
    with oldpop[j] do begin
      write(lst,' ',x:10,' ',fitness:6;4,);
    end;
    with newpop[j] do begin
      write(lst,j,parent1,parent2,xsite);
      write(lst,chrom,lchrom);
      writeln(lst,x,fitness);
    end;
  end;
  writeln(lst,gen,max,min,avg,sumfitness,nmutation,ncross);
end;
```

Figure C.5
File: report.sga [Goldberg 89].

```
function  select(popsize:integer; sumfitness:real;
                 var pop:population) : integer;
var rand, partsum : real;
    j:integer;
begin
  partsum := 0.0; j:=0;
  rand := random * sumfitness;
  repeat
    j:= j+1;
    partsum:= partsum+pop[j].fitness;
  until (partsum >= rand) or (j = popsize);
select := j;
end;


funtion mutation(alleleval:allele; pmutation:real;
                 var nmutation:integer) : allele;
var mutate:boolean;
begin
  mutate := flip(pmutation);
  if mutate then begin
    nmutaton:= numtation +1;
    mutation:= not alleleval;  end
  else
    mutation:= alleleval;
end;


procedure crossover(var parent1, parent2,
                        child1, child2: chromosome;
                    var lchrom, ncross, nmutation,
                        jcross: integer;
                    var pcross, pmutation: real;)
var j : integer;
begin
  if flip(pcross) then begin
    jcross:= rnd(1,lchrom-1);
    ncross:= ncross+1; end
  else
    jcross:= lcross;
  for j:= 1 to jcross do begin
    child1[j]:= mutation(parent1[j], pmutation, nmutation);
    child2[j]:= mutation(parent2[j], pmutation, nmutation);
  end;
  if jcross<>lchrom then
    for j:= jcross+1 to lchrom do begin
      child1[j]:= mutation(parent2[j], pmutation, nmutation);
      child2[j]:= mutation(parent1[j], pmutation, nmutation);
    end;
end;
```

Figure C.6
File: triops.sga [Goldberg 89].

```
procedure generation;
var j, mate1, mate2, jcorss:integer;
begin
  j:=1;
  repeat
    mate1 := slect(popsize,sumfitness,oldpop);
    mate2 := slect(popsize,sumfitness,oldpop);
    crossover(oldpop[mate1].chrom, oldpop[mate2].chrom,
              newpop[j].chrom, newpop[j].chrom, lchrom,
              ncross, nmutation, jcross, pcross, pmutation);
    with newpop[j] do begin
      x:= decode(chrom,lchrom);
      fitness:= objfunc(x);
      parent1 := mate1;
      parent2 := mate2;
      xsite := jcross;
    end;
    with newpop[j+1] do begin
      x:= decode(chrom,lchrom);
      fitness:= objfunc(x);
      parent1 := mate1;
      parent2 := mate2;
      xsite := jcross;
    end;
    j := j + 1;
  until j>popsize;
end;
```

Figure C.7
File: generate.sga [Goldberg 89].

```
procedure pause(pauselength : integer);
const maxpause := 2500;
var j,j1: integer;
    x: real;
begin
  for j:=1 to pauselength do
    for j1:=1 to maxpause do x:= 0.0 + 1.0;
end;


procedure page(var out:text)
begin
  write(out,chr(12));
end;


procedure repchar(var out:text; ch:char; repcount:integer);
var j: integer;
begin
  for j:=1 to repcount do write(out,ch);
end;


procedure skip(var out:text; skipcount:integer);
var j:integer;
begin
  for j:=1 to skipcount do writeln(out);
end;


funtion power9x,y,real):real;
begin
  power := exp(y*ln(x));
end;
```

Figure C.8
File: utility.sga [Goldberg 89].

Bibliography.

Abd-El-Hafiz, S., Basili, V.R., (1994), "A Tool for Assisting the Understanding and Formal Development of Software", in the Proceeding for The 6th International Conference on Software Engineering and Knowledge Engineering (SEKE), Jurmala, Latvia, June 21-23, pp. 36-45.

Alagappan, V., Kozaczynski, W., (1991) "The Evolution of Very Large Information Systems" in *Automating Software Design*, Lowry, Michael, R., McCartney, Robert, D. (Editors) AAAI Press / The MIT Press, pp. 3-24.

Alexander, H., Jones, V., (1990), *Software Design and Prototyping using me too*, Prentis Hall Ltd.

Bailes, P.A., Chapman, M., Gong, M., Peake, I, (1993), "GRIT- An Extended Refine for More Executable Specifications", in Proceedings of The Eighth Knowledge Based Software Engineering Conference (KBSE-93), Chicago, Illinois Sept. 20-23, pp. 123-132.

Balzer, R., Goldman, N., (1986), "Principles of Good Software Specification and their Implications for Specification Languages" in *Software Specification Techniques*, Gehani, N., McGettrick, A.D. (Eds.), Addison-Wesley, pp. 25-39.

Barnes, B. H., Bollinger, T. B., (1991), "Making Reuse Cost-Effective", *IEEE Software*, January, pp. 13-24.

Barr, A., Feigenbaum, E.A., (1982) *The Handbook of Artificial Intelligence Volume II*, William Kaufmann, Inc.

Barstow, D.R., (1979), "An Experiment in Knowledge Based Automatic Programming", *Artificial Intelligence*, 12, pp. 73-119.

Basili, V.R., (1985), "Quantitative Evaluation of Software Methodology", Tech. Report 1519, Computer Science Dept., University of Maryland, College Park, Md.

Basili, V.R., (1990) "Viewing Maintenance as Reuse-Oriented Software Development", *IEEE Software*, January, pp. 19-25.

Basili, V.R., Rombach, H.D., (1988), "Towards a Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment", Tech. Report CS-TR-2158, Dept. of Computer Science, Univ. of Maryland, College Park, MD. Dec.

Basili, V.R., Rombach, H.D., (1991), "Support for Comprehensive Reuse", *Software Engineering Journal*, September, pp. 303-316.

Basili, V.R., Turnner, A.J., (1975), "Iterative Enhancement: A practical Technique for Software Development", *IEEE Trans. Software Eng.*, December, pp. 390-396.

Bendifallah, S., Sacchi, W., (1987), " Understanding Software Maintenance Work", *IEEE Transactions on Software Engineering*, SE-13, No.3, March, pp. 311-323.

Berzins, V. Luqi, (1991), *Software Engineering with Abstractions*, Addison-Wesley Pub. Co.

Biggerstaff, T., Richter, C., (1987), "Reusability Framework, Assessment, and Directions", *IEEE Software*, Vol. 4, No. 2, pp. 41-49.

Boehm, B., (1977), "Seven Basic Principles of Software Engineering", in *Infotech State of the Art Reports: Software Engineering Techniques*, Maidenhead, England: Infotech International, pp. 77-113.

Boehm, B., Brown, J., Kaspar, J., Lipow, M., (1978) *Characteristics of Software Quality*, North Holland.

Boehm, B.W., (1975), "The High Cost of Software", in *Practical Strategies for Developing Large Software Systems,* Horowitz, E, (editor), Addison-Wesley Pub. Co.

Booch, G. (1987), *Software Components with Ada: Structures, Tools, and Subsystems*, Benjamin/Cummings, Calif.

Booch, G. (1994), *Object-Oriented Analysis and Design with Applications* 2nd Ed., Benjamin/Cummings Pub. Co. Inc. California.

Boyer, R.S., Moore, J.S., (1984), "Proof Checking the RSA Public Key Encryption Algorithm", *The American Mathematical Monthly* , 91(3), pp. 181-189.

Chandrasekaran, B., (1990), "Design Problem Solving: A Task Analysis", *AI MAGAZINE*, Winter, pp. 59-71.

Cheatham, T.E. Jr., (1984), "Reusability through Program Transformations", *IEEE Trans. on Software Engineering*, SE-10(5), pp. 589-594.

Chikofsky, E.J., Cross, J.H.II, (1990), "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software,* January, pp. 13-17.

Claben, I., Ehrig, H., Wolz, D., (1993), *Algebraic Specification Techniques and Tools for Software Development: the Act Approach*, World Scientific Pub. Co. Pte. Ltd.

Darlington, J., (1981), "An Experimental Program Transformation and Synthesis System", *Artificial Intelligence*, No. 1, pp. 1-46.

Esteva, J.C., Reynolds, R.G., (1991) “Learning to Recognize Reusable Software by Induction”, *IJSEKE*, Vol. 1 No.3, pp. 271-292.

Feather, M.S., London, P.E., (1982), "Implementing Specification Freedoms", *Science of Computer Programming*, No. 2, pp. 91-131.

Fenton, N. E., (1991), *Software Metrics: A Rigorous Approach*, Chapman and Hall Press, London.

Firebaugh, M.W., (1988), *Artificial Intelligence A Knowledge-Based Approach*, PWS-Kent Publishing Company, Boston.

Fisher, D.H., (1990), "Knowledge Acquisition Via Incremental Conceptual Clustering", in *Readings in Machine Learning*, Shavlik, J., Dietterich, T. (editors), Morgan Kaufmann, pp. 267-283.

Gehani, N., McGettrick, A.D., (1986), *Software Specification Techniques*, Addison-Wesley.

Gluck, M.A., Corter, J.E., (1985), "Information, Uncertainty and the Utility of Categories" in Proceedings of the 7th Annual Conference of the Cognitive Science Society, Lawrence Erlbaum Associates, pp. 123-133.

Goldberg, D.E., (1989), *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley Pub. Co. Inc.

Good, D.I., (1984), "Mechanical proofs About Computer Programs", *Philos. Trans. Royal Soc. London.*, 312(1522), pp. 43-96.

Green, C., (1991) Forward to *Automating Software Design*, Lowry, Michael, R., McCartney, Robert, D. (Editors) AAAI Press / The MIT Press, pp. xiii-xvi.

Green, C., Westfold, S., (1982), "Knowledge Based Programming Self Applied", in *Readings in Artificial Intelligence and Software Engineering*, Rich, C., Waters, R.C., (eds.), Morgan Kaufmann Pub. Inc., pp. 259-284.

Hall, P.A.V., (1992) "Overview of Reverse Engineering and Reuse Research", *Information and Software Technology*, Vol. 34, No. 4, pp. 239-249.

Hall, R.J., (1993), "Validation of Rule-Based Reactive Systems by Sound Scenario Generalization", in Proceedings of The Eighth Knowledge Based Software Engineering Conference (KBSE-93), Chicago, Illinois Sept. 20-23, pp. 30-39.

Halstead, M.H., (1977), *Elements of Software Science*, Elsevier-North Holland, New York, NY.

Hamilton, H.J., Fudger, D.R., (1995), "Estimating DBLEARN's Potential for Knowledge Discovery in Databases", *Computational Intelligence*, Vol. 11, No. 2, pp. 1-18.

Harandi, M.T., Ning, J.Q., (1990), "Knowledge-Based Program Analysis", *IEEE Software*, January, pp. 74-81.

Hartman, J., (1991), "Automatic Control Understanding for Natural Programs," Ph.D. Dissertation, The University of Texas at Austin, AI91-161, Austin, Texas 78712.

Hausler, P.H., Pleszkoch, M.G., Linger, R.C., Hevner, A.R., (1990), "Using Function Abstraction to Understand Program Behavior", *IEEE Software*, January, pp. 55-63.

Horowitz, E., Munson, J.B., (1984), "An Expansive View of Reusable Software", *IEEE Trans. on Software Engineering,* SE-10, 5, Sept., pp. 477-487.

Horwitz, S., Reps, T., Binkley, D., (1990), "Interprocedureal Slicing Using Dependence Graphs", *ACM Trans. on Prog. Lang. and Systems*, Vol. 12, No. 1, pp. 26-60.

Johnson, W.L., Feather, M.S., (1991) "Using Evolution Transformations to construct Specifications" in *Automating Software Design*, Lowry, Michael, R., McCartney, Robert, D. (Editors) AAAI Press / The MIT Press, pp. 65-92.

Johnson, W.L., Soloway, E., (1985), "PROUST: Knowledge Based Program Understanding", *IEEE Trans. on Software Engineering*, SE-11(3), pp. 267-275.

Jones, C.B., Jones, K.D., Lindsay, P.A., Moore, R., (1991), *MURAL A Formal Development Support System*, Springier-Verlag London Limited.

Kant, E., (1983), "On the Efficient Synthesis of Efficient Programs", *Artificial Intelligence*, 20, pp. 253-306.

Korel, B., Laski, J., (1990), "Dynamic Slicing of Computer Programs", *J. Systems Software*, 13, pp. 187-195.

Krueger, C.W. (1992), "Software Reuse", *ACM Computing Surveys*, Vol. 24, No. 2, pp. 131-183.

Lamb, D.A., (1988), *Software Engineering Planning for a Change*, Prentice Hall.

Lee, H.Y., Harandi, M.T., (1993), "An Analogy Based Retrieval mechanism for Software Design Reuse", in Proceedings of The Eighth Knowledge Based Software Engineering Conference (KBSE-93), Chicago, Illinois Sept. 20-23, pp. 152-159.

Letovsky, S.I., (1988), "Plan Analysis of Programs", Ph.D. Dissertation, Department of Computer Science, Yale University.

Liskov, B.H., Berzins .V., (1986), "An Appraisal of Program Specifications" in *Software Specification Techniques*, Gehani, N., McGettrick, A.D. (Eds.), Addison-Wesley, pp. 3-24.

Littman, D., Pinto, J., Letovsky, S., Soloway, E., (1987), "Mental Models and Software Maintenance", *The Journal of Systems and Software*, 7, pp. 341-355.

Lowry, M., Duran, R., (1990), *The Handbook of Artificial Intelligence Volume IV*, Barr, A., Cohen, P.R., Feigenbaum, E.A., (eds.), Chapter XX, Knowledge Based Software Engineering, Addison-Wesley Pub. Co. Inc., pp. 243-267.

Luger, G.F., Stubblefield, W.A., (1989), *Artificial Intelligence and the Design of Expert Systems,* The Benjamin/Cummings Pub. Co. Inc.

Lyons, M.J., (1981), "Salvaging Your Software Asset (Tools-Based Maintenance)", AFIPS Conference Proceeding on 1981 National Computer Conference Chicago, IL, Vol. 50, May 4-7, pp. 337-342.

Maletic, J.I., Reynolds R.G., (1994), "A Tool to Support Knowledge Based Software Maintenance: The Software Service Bay", in Proceeding for the 6th IEEE International Conference on Tools with Artificial Intelligence, New Orleans LA, November 6-9, pp. 11-17.

Maletic, J.I., (1989), *Refinement Metrics: Assessing a Programming Language's Support of the Stepwise Refinement Process*, M.S. Thesis, Department of Computer Science, Wayne State University, Detroit, Michigan.

Manna, Z, Waldinger, R., (1980), "A Deductive Approach to Program Synthesis", *ACM Trans. on Programming Languages and Systems*, 2(1), pp. 90-121.

Martin, J., McClure, C. (1983), S*oftware Maintenance, The Problem and its Solutions,* Prentice-Hall Inc.

McClure, C., (1976), "Formalization and Application of Structured Programming and Program Complexity", Ph.D. Dissertation, Illinois Institute of Technology.

Michalski, R.S., Stepp, R.E., (1983), "Learning from Observation: Conceptual Clustering", in *Machine Learning: An Artificial Intelligence Approach*, Michalski R.S., Carbonell, J.G., Mitchell, T.M. (Eds.), Morgan Kaufmann, pp. 341-364.

Michalski, R.S., Stepp, R.E., (1990), "Clustering", in *The Encyclopedia of Artificial Intelligence*, John Wiley and Sons.

Miller, G.A., (1956), "The Magical Number Seven, Plus or Minus two: Some Limits on Our Capacity for Processing Information", *The Psychological Review*, Vol. 63, No. 2, pp. 81-97.

Mills, H.D., (1993), "Zero Defect Software: Cleanroom Engineering", *Advances in Computers*, Vol. 36, pp. 1-41.

Osborne, W.M., Chikofsky, E. J., (1990), "Fitting Pieces to the Maintenance Puzzle", *IEEE Software*, January, pp. 11-12.

Posner, R.M., (1994), "Program Understanding through the Use of Cultural Algorithms", Masters Thesis, Computer Science Department, Wayne State University.

Pressman, R.S., (1992), *Software Engineering, A Practitioner's Approach* 3rd Ed., McGraw-Hill Book Co.

Prieto-Diaz, R., (1985), "A Software Classification Scheme", Ph.D. Thesis, Dept. ICS, University of California-Irvine.

Prieto-Diaz, R., (1993), "Status Report: Software Reusability", *IEEE Software*, May, pp. 61-66.

Prieto-Diaz, R., Freeman, P., (1987), "Classifying Software for Reusability", *IEEE Software,* January, pp. 6-16.

Quinlan, J.R., (1986), "Induction of Decision Trees", *Machine Learning*, 1:81-106.

Rajlich, V., Damaskinos, N., Linos, P., Khorshid, W., (1990), "VIFOR: A Tool for Software Maintenance", *Software- Practice and Experience*, Vol. 20, No. 1, pp. 67-77.

Ranganathan, S., (1967), *Prolegomena to Library Classification*, Asia Publishing House, Bombay, India.

Reddy, D.R., Erman, L., Fennell, R., Neely, R., (1976), "The HEARSAY Speech Understanding System: An Example of the Recognition Process", *IEEE Transactions on Computers* C-25, pp. 427-431.

Reynolds, R.G., Maletic, J.I., (1990), "An Introduction to Refinement Metrics: Assessing a Programming Language's Support of the Stepwise Refinement Process", in Proceedings of 18th Annual ACM Computer Science Conference, Washington D.C. Feb. 21-23, pp. 82-88.

Reynolds, R.G. Zannoni, E., (1992) "Extracting Procedural Knowledge From Software Systems Using Inductive Learning in the PM System", *International Journal on Artificial Intelligence Tools*, Vol.1, No. 3, pp. 351-367.

Reynolds, R.G., (1994), "An Introduction to Cultural Algorithms", in *Proceedings of the Third Annual Conference on Evolutionary Programming*, A. V. Sebald and L.J. Fogel (Eds.), World Scientific Pub. Co. Ltd., pp. 131-139.

Reynolds, R.G., Maletic, J.I., (1993), "The Use of Version Space Controlled Genetic Algorithms to Solve the Boole Problem", *International Journal on Artificial Intelligence Tools (IJAIT)*, Vol. 2, No. 2, June, pp. 219-234.

Reynolds, R.G., Maletic, J.I., Porvin S.E., (1992), "Stepwise Refinement and Problem Solving", *IEEE Software*, September, pp. 79-88.

Reynolds, R.G., Maletic, J.I., Porvin, S., (1990), "PM: A System to Support Acquisition of Programming Knowledge", *IEEE Transactions on Knowledge Engineering and Data Engineering*, Vol.2, No. 3, Sept., pp. 273-282.

Reynolds, R.G., Sverdlik, W., (1993) "Solving Problems in Hierarchically Structured Systems Using Cultural Algorithms", in Proceedings Of The Second Annual Conference on Evolutionary Programming ,La Jolla, CA, pp. 144-153.

Reynolds, R.G., Zannoni, E, Posner, R (1994), "Learning to Understand Software Using Cultural Algorithms", in *Proceedings of the Third Annual Conference on Evolutionary Programming*, A. V. Sebald and L.J. Fogel (Eds.), World Scientific Pub. Co. Ltd., pp. 151-157.

Rich, C, Waters, R.C., (1988) "The Programmer's Apprentice: A Research Overview", *IEEE Computer*, Vol. 21, No. 11, November, pp. 10-25.

Rich, C., Wills, Linda M., (1990), "Recognizing a Program's Design: A Graph-Parsing Approach", *IEEE Software*, January, pp. 82-89.

Rich, C., Waters, R.C., (1988), "Automatic Programming: Myths and Prospects", *IEEE Computer*, Vol. 21, No. 8, August, pp.40-52.

Rist, R., (1992), "Plans in Program Design and Understanding", in Workshop Notes for AI & Automated Program Understanding, AAAI-92, July 12-16, San Jose, CA, pp. 98-102.

Rugaber, S., Ornburn, S.B., LeBlanc, R.J. Jr., (1990), "Recognizing Design Decisions in Programs", *IEEE Software*, January, pp. 46-54.

Schach, S.R., (1993), *Software Engineering* Second Edition, Aksen Associates Incorporated Publishers.

Schneidewind, N.F., (1987), "The State of Software Maintenance", *IEEE Transactions on Software Engineering*, SE-13, No. 3, March, pp. 303-310.

Schonberg, E., Schwartz, J.T., (1981), "An Automatic Technique for Selection of Data Representations in SETL Programs"," *ACM Trans. on Programming Languages and Systems,* 3(2), pp. 126-143.

Simon, H., (1967), *The Science of the Artificial*, Cambridge, Massachusetts, M.I.T. Press.

Simon, H., (1986), "Whether Software Engineering Needs to be Artificially Intelligent", IEEE Trans. on Software Eng., Vol. 12, No. 7, pp. 726-732.

Smith, D.R., (1990), "KIDS: A Semiautomatic Program Development System", *IEEE Trans. on Software Engineering*, Vol. 16, No. 9, Sept., pp. 1024-1043.

Smith, T.E., Setliff, D.E., (1993), "Towards Supporting Design Phase Synthesis", in Proceedings of The Eighth Knowledge Based Software Engineering Conference (KBSE-93), Chicago, Illinois Sept. 20-23, pp. 20-29.

Sommerville, I., (1989), *Software Engineering* 3rd Ed., Addison-Wesley Publishing Co.

Swanson, E.B., (1976), "The Dimensions of Maintenance", in Proc. 2nd IEEE Intl. Conf. Software Engineering, pp. 492-497.

Veryard, R., (1985), "Methods of Maintenance", *Data Processing*, Vol. 27, No.9, pp. 22-26.

Wang, F., (1991) "Cognitive Models of Software Reuse: Extracting Procedural Knowledge in a Multiple Pass Knowledge Compiler", M.S. Thesis, Wayne State University, Detroit, Michigan.

Weiser, M., (1984), "Program Slicing", *IEEE Trans. on Software Eng.*, Vol. 10, No. 4, pp. 352-357.

Wickey, M.C., (1994), "Reuse Library System, Software classification Through Conceptual Clustering", Masters Thesis, Computer Science Department, Wayne State University.

Yau, S.S., Nicholl, R.A., Tsai, J.J.P., Liu, S.S., (1988), "An Integrated Life-Cycle model for Software Maintenance", *IEEE Trans. on Software Eng.*, Vol. 14, No. 8, August, pp.1128-1144.

Zeil, S.J., Wild, C., (1993), "A Knowledge base for Software Test Refinement", in Proceedings of The Eighth Knowledge Based Software Engineering Conference (KBSE-93), Chicago, Illinois, Sept. 20-23, pp. 50-59.

ABSTRACT

THE SOFTWARE SERVICE BAY: A KNOWLEDGE BASED SOFTWARE MAINTENANCE METHODOLOGY

by

JONATHAN IRVINE MALETIC
May, 1995

Advisor: Dr. Robert G. Reynolds

Major: Computer Science (Artificial Intelligence)

Degree: Doctor of Philosophy

The maintenance of software has been found to be the most costly part of the software life cycle. Little work explicitly addresses a framework for software maintenance problems solving. A software maintenance methodology, the Software Service Bay, is introduced which focuses on the different type of problem solving required in software maintenance. The Software Service Bay is a knowledge based maintenance methodology analogous to the automotive service bay. Within an automotive service bay there are many different auto mechanic experts, each with specific knowledge on different maintenance problem, be they transmission, brakes, engine, etc. Each mechanic solves problem with their domain using different knowledge, tools, and methods. Software maintenance is much the same way. The variety of problem types (perfect, corrective, etc.) lends to a specific set knowledge, specific set of tools, and specific problem solving methods for each type. A set of software specialists is described which have knowledge of the software reuse process. These specialists are

combined into a blackboard architecture to facilitate opportunistic problem solving for maintenance problems.

Autobiographical Statement.

Jonathan Irvine Maletic was born on June 5th, 1964 in Flint, Michigan to John and Alice Irvine Maletic. Jonathan graduated from Lake Fenton High School, near Fenton Michigan, in 1982. During his High School years he received a number of awards in mathematics and science. Jonathan went on to attend The University of Michigan-Flint and received a B.S. in Computer Science, with a Mathematics Minor, in December of 1985. In the fall of 1986 Jonathan started his graduate studies at the Department of Computer Science at Wayne State University, located in Detroit, Michigan. He completed the M.S. in December 1989 and the Ph.D. in May 1995. His advisor in both degrees was Dr. Robert G. Reynolds. Jonathan's research interests center around Artificial Intelligence and Software Engineering. In the 1991-92 academic year Jonathan participated in the Graduate Student Internship Program at Honeywell Inc.'s Systems & Research Center in Minneapolis, MN. Jonathan was awarded a competitive NASA Research Fellowship through NASA's Graduate Student Researchers Program in 1993 and 1994. He also held a number of Graduate Research and Teaching Assistantships during his time as a graduate student at WSU. Jonathan spends his spare time racing/building sail and ice boats. He is also a collector of art deco, streamline, arts and crafts, and mid-century modern furniture, pottery, and art.