# USING SKIP LISTS IN THE IMPLEMENTATION OF A HYPERTEXT TOOL FOR MAINTENANCE PROGRAMMERS

Bing Zhang

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

March 1999

0-612-39119-1

# Abstract

## Using Skip Lists in the Implementation of a Hypertext Tool for Maintenance Programmers

Bing Zhang

This thesis presents a hypertext browser tool mainly for maintenance phase of software development. As the maintenance phase is the most costly and time-consuming phase in the whole process of software development and system evolution, our tool is aimed at providing support to maintenance programmers for better understanding of existing code and maintaining of large applications. With this tool, programmers can setup link between identifiers and their definitions, browse through source code, have easy access to definitions of any user defined identifiers and routines through hypertext, and inspect each occurrences of an identifier, which can be highlighted in browser window, of any file. But our tool is not limited to the maintenance phase activities. It can be used in any phases with text documentation, such as the important phases like the design and implementation phases. Our tool can also provide assistance to designers and developers by supporting documentation inspection.

In this thesis, we surveyed software development environments and supporting tools. From our survey, we understand the development history and future directions in this area. This helps the design and implementation decisions of our tool.

Some future applications of this tool are also discussed. Our tool will be more completed and helpful to maintenance programmers with these future enhancement. We believe this will brighten the future of our tool.

# Acknowledgments

I would like to express my gratitude to my supervisor Dr. Peter Grogono for his enthusiastic support and consistent guidance. It is his valuable suggestions and encouragement that made this work possible.

I appreciate the helpful comments and suggestions from my committee members, Dr. Thiruvengadam Radhakrishnan and Dr. Joey Paquet.

My sincere thanks to my husband, Ping, for his encouragement, patience and moral support.

Finally I would like to dedicate this work to my parents who always stood behind me with their unwavering love and encouragement.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Software Development Process

Since the first computers were built in the 1940s, software has become part of our everyday lives recently. The scale and scope of software application has changed tremendously. Programs are often very large and complicated, and are developed by teams that collaborate over periods spanning months or even years.

We do not need to explain the enormous importance of the field of software engineering. The definition of software engineering given in the IEEE Standard Glossary of Software Engineering Terminology is as follows:

*Software engineering is the systematic approach to the development, operation, maintenance, and retirement of software.*

### 1.1.1 Software Development Process Model

In software engineering the development of software is divided into distinct phases. The phases are an abstraction of the complex process of software development, but these basic phases have to be completed in some sense in every project. The so-called **process model** consists of the basic components of these phases which are shown in Figure 1.

- **Requirement analysis** is the first phase of the process model. The goal of the requirement analysis phase is to get a complete description of the problem to be solved and environment requirements.

- The **Design phase** develops a model of the whole system and produces a global structure of modules and their interfaces.
- The **Implementation phase** concentrates on the individual modules. This phase actually realizes the design to fulfil the requirements. The result of the implementation phase is an executable program, which is the core of the software delivered to the customer. During this phase, programmers usually writes programs with one or more text editors of their choice.

The text editor is a very common tool for programmers, as all programmers need to edit the source code to accomplish their day-to-day tasks. Text editing is one of the main activities that programmers practice every day. The use of a good tool can affect the efficiency of programmers by an order of magnitude [McC89]. We will discuss this aspect in detail in later sections.

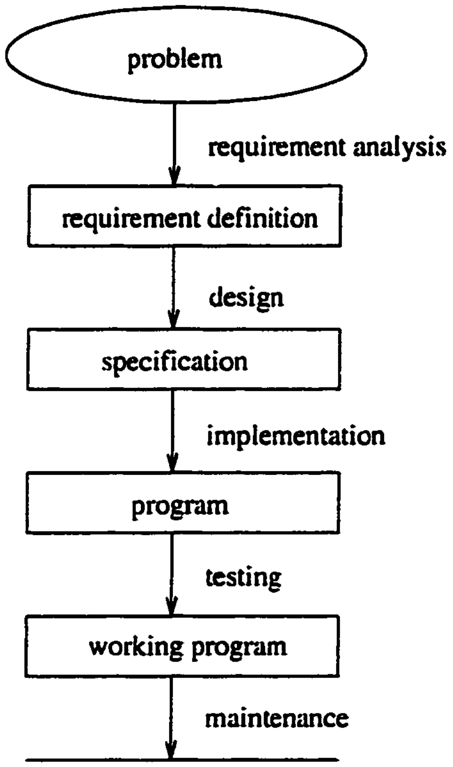


Figure 1: Software Development Process Model

- **Testing** is a phase to detect and correct errors. To fix the problems and issues found, programmers use a debugger to locate errors and editor to modify the source code accordingly.

- **Maintenance** is the last phase in the software development process model, as shown in Figure 1. After delivery of the software, there are often errors that have still gone undetected. But maintenance is not limited to the correction of these errors or faults. As shown in Figure 2, the fault correction activity only accounts for about a quarter of the total maintenance effort. In addition, the actual use of the system can lead to requests for changes and enhancements. Software maintenance is defined in IEEE Standard Glossary of Software Engineering Terminology as:

  > *the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.*

  Lientz distinguishes four kinds of maintenance activities [LS80]. The distribution of maintenance activity is shown in Figure 2.

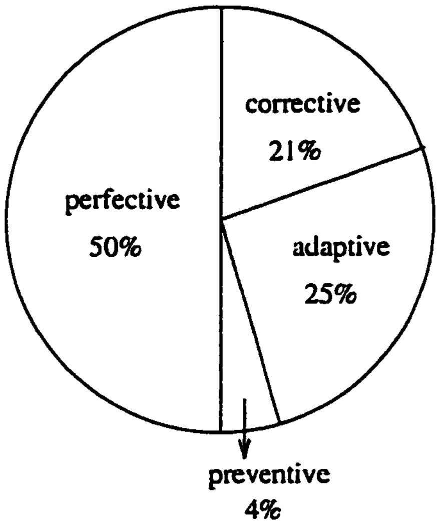


Figure 2: Distribution of Maintenance Activities

  - Corrective maintenance: the repair of actual errors. We will discuss a corrective maintenance tool in the next chapter.
  - Adaptive maintenance: adapting the software to changes in the environment, such as new hardware or the next release of an operating or database system.

- Perfective maintenance: adapting the software to new or changed user requirements, such as extra functions to be provided by the system. Perfective maintenance also includes work to increase the system's performance or to enhance its user interface.
- Preventive maintenance: increasing the system's future maintainability. Updating documentation, adding comments, or improving the modular structure of a system are examples of preventive maintenance activities.

We can summarise maintenance activities in the following way:

> "*Maintenance concerns all activities needed to keep the system operational after it has been delivered to the customer.*" [Vli93]

Customer service and support ensures customer satisfaction, which leads to real success for everyone. "*The only thing we maintain is user satisfaction*" by Lehman [Leh80] summarises the importance of maintenance activities in the software development process.

### 1.1.2 Cost of Software Development

The cost of software includes not only the cost of implementing the software, but also the cost of keeping the software operational after it has been delivered to the customer. In recent years hardware costs have decreased dramatically. Hardware costs now typically comprise less than 20% of total expenditure. The remaining 80% comprise all non-hardware costs: the cost of programmers, analysts, management, user training, administration cost, etc. Figure 3 shows the distribution of costs studied by Chikofsky and Rubenstein [CR88].

When we consider the total cost of software system over its lifetime, it turns out that on average maintenance alone consumes 50-75% of these costs [Boe76] [Leh80]. Thus, maintenance alone consumes more than all of the other development phases taken together.

## 1.2 Why Is the Tool Needed?

It is well known that software development, especially large project with many components, is difficult and expensive. And to make things worse, software evolution is inevitable.

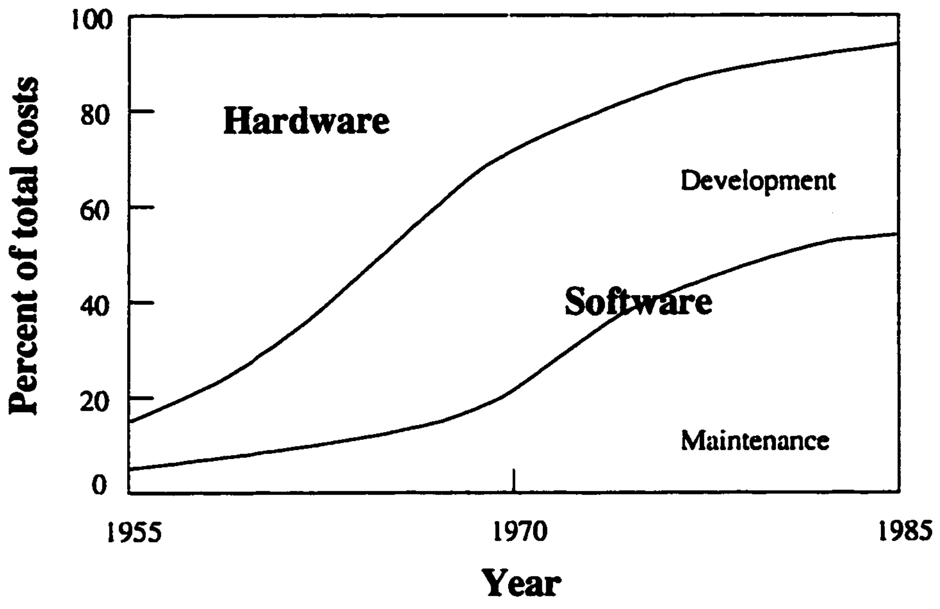


Figure 3: Relative Distribution of Hardware/Software Costs

Lehman and Belady [LB85] have extensively studied the dynamics of software systems that grow in size and need to be maintained. Based on those quantitative studies, they formulated the following laws of software evolution:

1. Law of continuing change: A system undergoes continuous change until it's more cost-effective to restructure the system or replace it by a complete new version.

2. Law of increasing complexity: A program becomes less structured and more complex after changes. One has to invest extra effort in order to avoid increasing complexity.

3. Law of program evolution: The growth rate of global system attributes may seem locally stochastic, but is in fact self-regulating with statistically determinable trends.

4. Law of invariant work rate: The global progress in software development projects is statistically invariant.

5. Law of incremental growth limit: A system develops a characteristic growth increment. When this increment is exceeded, problems concerning quality and usage will result.

So, these laws explain that a software system changes continuously towards increasing complexity, exactly as we see in the industry. We can not stop or slow down the trend, the only thing we can do is how to cope with it and how to work efficiently.

Even if we start with perfectly designed, implemented and documented software, the software will tend towards the cycle of decay and getting more difficult to maintain. Since most of the development time and cost are spent on software maintenance, tools to facilitate maintenance programmers to improve their efficiency will be very useful.

Better tools for software development may result in large financial savings, in more effective methods of software development, in systems that better fit user needs, in more reliable software systems, and thus a more reliable environment in which those systems function. Quality and productivity, the two most central themes in the field of software engineering, can be improved a great deal with a good tool.

Boehm estimates that even the use of very modest tools may result in cost savings of up to 10% [Boe81]. The study reported in [NN89] also shows that, in the perception of software engineers, productivity is improved with the use of automated tools. From this study, it appears that the data flow diagram and data dictionary functions of the CASE (Computer Aided Software Engineering) products contribute the most to the software engineer's productivity improvements over manual methods.

In the whole process of software development, several phases have a close relationship with source code browsing and editing.

- In the implementation phase, programmers translate the design document into code in a chosen language. In order to do so, programmers need to keep in mind the function dependencies, data structures and variable declarations in a particular scope, interface and utility procedures, parameters passed, etc. For a large software system with many components, each programmer is working on a small part of it and it is impossible for one programmer to understand and memorise every piece of code. Consequently, tools are used to let programmers browse through and understand the finished code, program and finish their code.

- In the testing phase, whenever a bug is found, a programmer will be notified. The programmer will investigate the cause of the bug, find the location of the bug, modify the code and retest it. The tools needed are a debugger and an editor.

In the maintenance phase, maintenance programmers spend a large proportion of their time browsing through source code, trying to understand the code which most possibly was written by someone else, looking for the bugs and finding a right place to put the new code or debug code in.

From the above, it is clear that programmers, especially maintenance programmers, need a tool to help them to browse the code and to understand the functionality and structure of the software system so that they can maintain the software. Our tool is designed to facilitate these activities. The tool allows maintenance programmer to inspect the code, to look up a routine or variable, to search a particular string and highlight it in the code, to learn function call relations, etc., and provides a user-friendly interface.

In the following chapter, we will review the history of software tool development and examine several tool systems in detail. Compared to those CASE systems, our tool is not as sophisticated and has less functionality, but its simplicity and its use of hypertext make it easier to use. And it has the most useful functions for maintenance programmers.

## 1.3 Structure of the Thesis

The thesis is organised as follows:

Chapter 1 introduces the background of software development. cost of software. importance of maintenance phase and usefulness of tools.

In Chapter 2, we first review the background of development of tools in many fields of software development process, such as editors, debuggers, integrated development environments, automated testing tool, etc. by analysing some tools. We then conclude which tool functions we think are really needed by programmers. Several tools reported are then explained in detail.

Chapter 3 reviews the design issues that are considered to complete this project. Decisions about user interface, data structures and function structures are discussed and illustrated.

Chapter 4 explains the implementation of this tool. Details are given about the organisation of the code, the access structure, the user interface, major functions, etc.

Chapter 5 gives a general conclusion to the thesis and presents some future work that needs to be done to improve this tool.

Finally, a simple user manual of the tool with an example is given in the Appendix.

# Chapter 2

# Background

In this chapter, we review the development of tools, what tools programmers really need, and various tools that have been developed.

## 2.1 Development of Tools to Assist Programmers

At an early stage, tools were used for the implementation of software, such as compilers, linkers and loaders, and test drivers. The development of tools to support each phase, especially early phases, of the software life cycle and the development of integrated tool sets to support the whole software life cycle is more recent and fast-growing.

The application of tools in the software development process is referred to as CASE (Computer Aided Software Engineering). The first tools to support design activities appeared in the early 1980s. In 1990s, the number of CASE tools is overwhelming.

### 2.1.1 Vliet's Tool Category

To structure the growing amount of available tools, Vliet gives a tool category system depending on the scope of the tool collection [Vli93]:

- Analyst workbench (AWB): a collection of tools that support the requirements specification/global design phase.
- Programmer workbench (PWB): tools that mainly support the implementation and test phases.

- Management workbench (MWB): tools to support management tasks.
- Integrated project support environment (IPSE) or Software development environment (SDE): a collection of tools is intended to support all phases of the software life cycle.

This classification suggests a certain relation between a given model of the software life cycle and a corresponding tool collection. Ideally, this would be the case, but the classic waterfall model of software development is often disputed, one reason being that the number of available tools is rapidly increasing, and those tools support other process models, such as prototyping and evolutionary development, etc. There is an interaction between trends in support environments and trends in process models. [Vli93]

### 2.1.2 A Taxonomy of Program Development Environments

Dart *et al* [DEFH87] gives a taxonomy to sketch the developments in this area and distinguishes four categories, based on trends that have a major impact on support environments:

- **Environments based on a specific programming language** contain tools specifically suited to the support of software development in that language.

  In a language-based environment, the set of tools supports software development in one specific language. Well-known examples of language-based environments are Interlisp and the Smalltalk-80 environment. Ada Program Support Environment (APSE) is another example that will be discussed in section 2.1.3.

- **Environments based on the syntax of programming languages** contain tools aimed at manipulating program structures.

  These environments can be generated from a grammatical description of those program structures. Some special editors with built-in knowledge of certain language are a good example of this kind of environment. For example, in Cornell Program Synthesizer [TR81], if the user types in an `if`, the template appears:

  ```
  IF (<condition>)
  THEN <statement>
  ELSE <statement>
  ```

The user is asked to fill in the holes. Other supports, like indentation, close parentheses, incremental program analyses, etc. are offered as well. Structure-oriented environments provide capabilities for the direct manipulation of program structures, multiple views of programs, incremental checking of static semantics, and program debugging.

- In **toolkits** we find tools that are generally not so well integrated.

  The support offered is independent of a specific programming language. A toolkit merely offers a set of useful building blocks. In particular, toolkits tend to contain tools that specifically support programming-in-the-large. UNIX is a prime example from this category. UNIX may be viewed as a general programming environment, not aimed at one specific programming language. UNIX offers a number of convenient and simple building blocks with which more complex things can be realized. Here are several examples:

  - The file system is a tree. Leaves of the tree are files, and directories are inner nodes, which are files as well. Files have a very simple structure and an I/O device has the same behaviour (from the user's point of view) as a file. Any file can be processed by the user as if it is a stream of bytes: its actual structure is hidden by the operation system.
  - UNIX offers a large set of small, useful programs, such as `wc`, `grep`, `lpr`, input and output redirecting.
  - UNIX programs can easily be combined to form larger programs. Pipe, denoted by |, allows users to try to reach their goals by gluing existing components together, rather than writing a program from scratch.

  On the other hand, there is little consistency in interfaces and the choice of command names. To stop a dialogue, you may try kill, stop, quit, end, and more likely CTRL-c. The average UNIX user knows only a fairly limited subset of the available commands and tools [Fis86]. So the facilities offered under UNIX are far from optimally used.

- **Tools based on certain techniques** are used in specific phases of the software life cycle.

  In early phases of software development (**requirements analysis and design**), tools, called Analyst WorkBenches (AWB) [Vli93], are used to support analysis and gathering design data. Often, a graphical image of the system is made, for instance in the form of data flow diagrams. The AWB usually contains tools to support the following types of activities:

- drawing, changing and manipulation of pictures.
- analysis of data produced, consistency and completeness.
- generating reports and documentation.

A Programmer WorkBench [Vli93] consists of a set of tools to support the **implementation and test** phases of software development. In a PWB, we find tools to support, among others:

- edition and analysis of programs;
- debug;
- generation of test data;
- simulation;
- test coverage determination.

In the above two cases, the support offered mostly concerns the individual programmer. These environments are dominated by issues of software construction. The emphasis is on tools that support software construction: editors, debuggers, compilers, etc.

A Management WorkBench [Vli93] contains tools that assist the manager during **planning and control** of a software development project. Tools in an MWB include:

- configuration control
- work assignment
- cost estimation
- reliability

- Finally, **integrated tool sets** aim at supporting the full spectrum of the software life cycle in a coordinated fashion.

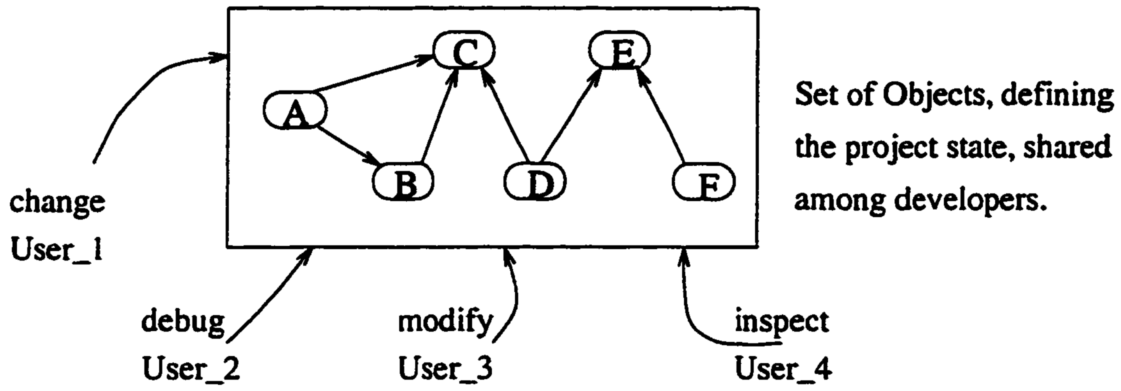
An Integrated project support environments is meant to support **all phases** of the software life cycle. Because of the complexity of this kind of system, we have a separate section to discuss its development.

### 2.1.3 Integrated System Development Environments

We first take a look at the environments from different point of views:

**Information management view** :

The philosophy behind this approach is that the key to automation in systems development is in systematically capturing and structuring the many different "objects" that are produced: requirements, design specifications, meeting records, software modules, test specifications, plans, etc. The state of a project is defined by the states of its objects and the progress is measured in terms of the growth of the objects in this information base. The idea is illustrated by Figure 4. This view has naturally led to database-centered approaches to environments.



A
B
C
D
E
F
Set of Objects, defining the project state, shared among developers.
change
User_1
debug
User_2
modify
User_3
inspect
User_4

Figure 4: The Information Management View

**Process support view** :

First define a precise development process to identify the necessary steps. actions. roles, and information flows throughout the complete life cycle. Taking the development process as a kind of blueprint. the environment should support the specified process, manifest it in executable software and computer stored structures. Sometimes the process engine is introduced. It would monitor the state changes. 'execute' the process model and finally give guidance to users and tools (see Figure 5 [SV93]).

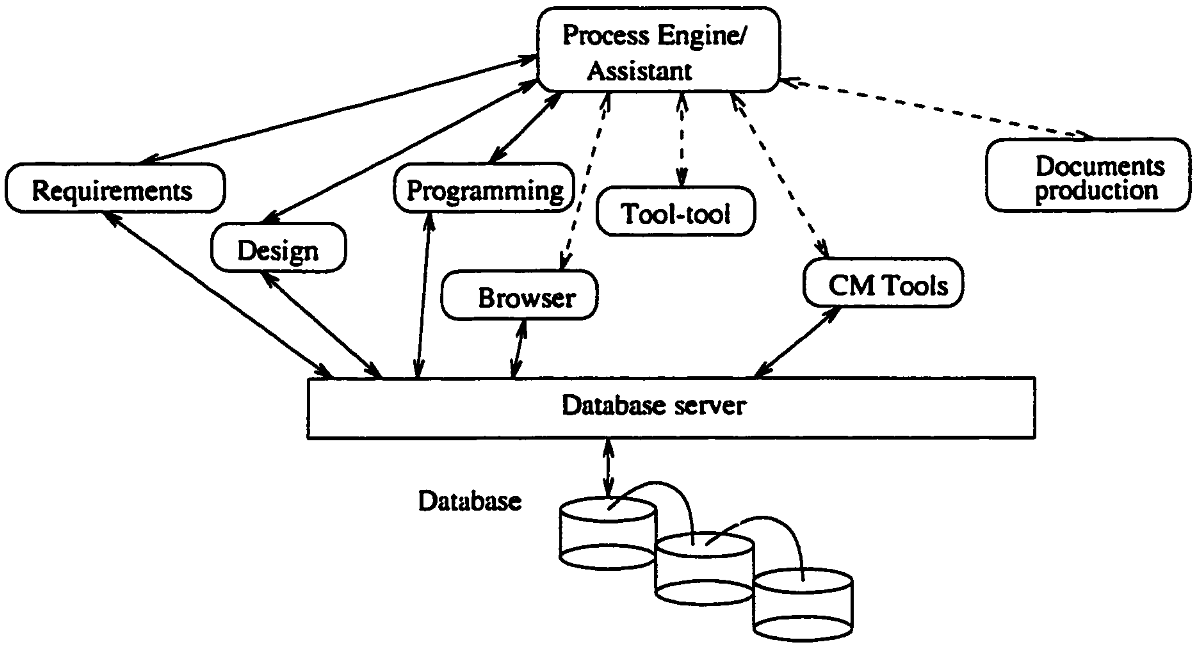


Figure 5: The Process Support View

## Models of Development Environments

Several generic models of environments have been suggested, at varying levels of abstraction and detail [SV93]. Let's look at some of them:

1. **Data orientation:**

   An early example, which has had significant impact, is the Stoneman document [Bux80] [Bux84]. This requirement specification for an Ada development environment established the APSE/MAPSE/KAPSE model, shown in Figure 6. Schefstrom [SV93] claims that 'Stoneman can be viewed as an early example of an environment reference model'.

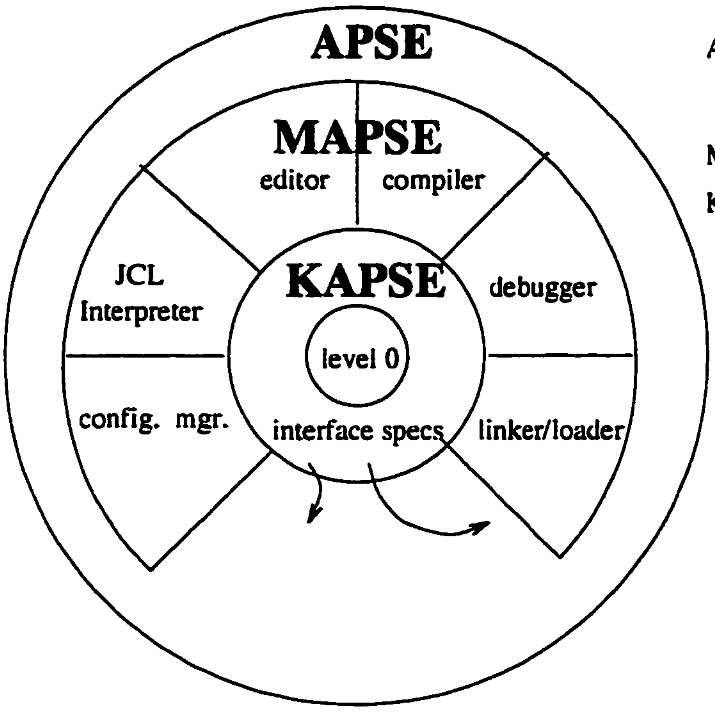
   *"The toolset must not only support the appropriate functions, but must be integrated into a consistent environment."*

   This sentence from Stoneman requirements emphasised *integration* – the recognition that for significant progress to be made, tools must cooperate towards a common goal to a higher degree than was currently the case.

The Stoneman requirements also emphasised the idea of the environment database. the all-encompassing structured data storage.

> *"The APSE must provide a well coordinated set of useful tools, with uniform inter-tool interfaces and with communication through a common database which acts as the information source and repository for all tools."*

All this was placed in the three level KAPSE/MAPSE/APSE model, with the database at the very kernel KAPSE level (see Figure6) .



APSE - Ada Programming Support Environment

MAPSE - Minimal APSE

KAPSE - Kernel APSE

Figure 6: The APSE Model

2. **Language orientation:**

Rational Environment [Rip88] [Mor88] is a highly integrated development environment dedicated to software development in Ada. It closely adhered to the original vision of an APSE and combined the large-scale project support attitude of Stoneman with ideas from early environments works, such as Interlisp.

Smalltalk, and language-oriented editors. Special purpose hardware supporting an operation system that was completely dedicated towards production of Ada software was used. The compiler was built to be incremental, with most tools supporting the internal representation of the Ada programs. The language, Ada, acts as the major integrating factor, resulting in a major example of the language oriented approach to environments. In addition, Rational environment has the unified internal form, that all data should be stored in a common format suitable for automatic processing. All tools should access the common representation, and thereby achieve both simplicity and uniformity, as indicated in Figure 7. With the language oriented and tight integration approach, it could also provide interactive cross-referencing and semantic completion, configuration control and documentation. But requiring special purpose hardware also introduces a problem: it may separate the environment from the continuous evolution of the rest of the world. However, the Rational environment is one of the few novel environments that have reached the state of a stable product in industrial use.

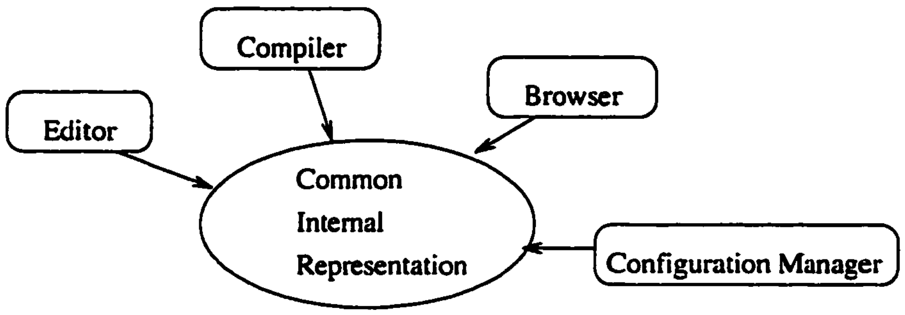
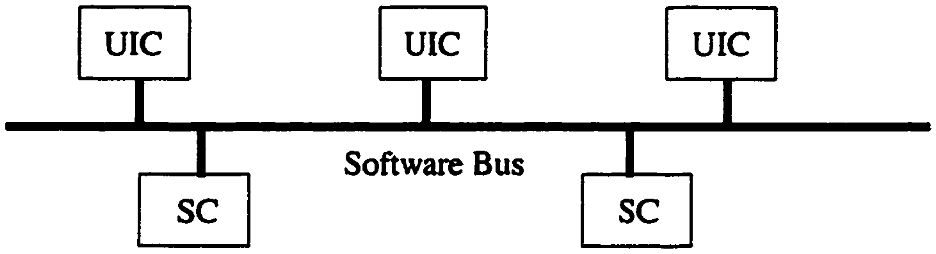


Figure 7: Common Representation Access: Simple and Uniform

3. **Communication orientation:**

A representative example of a development strategy based on communication is the European multi-company cooperation Eureka Software Factory, ESF [Fer88] [FO89]. ESF suggests a communications oriented architecture, where components communicate with each other over a software bus. The idea is shown in Figure 8.

This approach more or less rejects the idea of the central database, partly on the basis that a large number of companies are unlikely to agree on a single standard database and schema.



UIC: User Interface Components

SC: Service Components

Figure 8: ESF Communications Oriented Architecture

4. **Process orientation:**

An early example of an environments effort with a strong "process orientation" was ISTAR [Dow87]. The programming aspect of environments was de-emphasised, claiming that we should instead provide support for the orderly management of projects. The word, *contract*, was made a basic concept, around which any project should be organised. ISTAR therefore supported the organisation of a hierarchy of contracts, which defined what were the input and constraints and the expected deliverables. Figure 9 illustrates the idea.

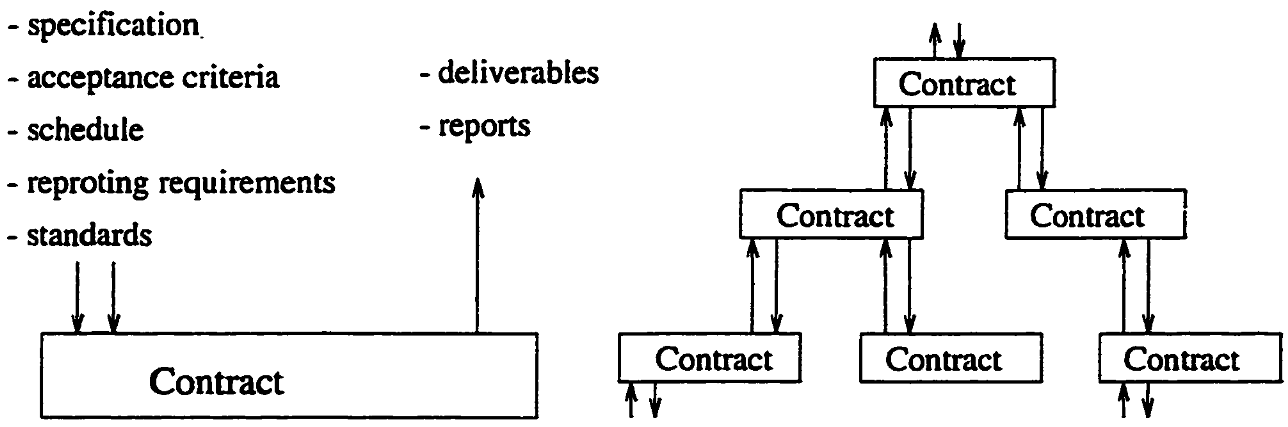


Figure 9: ISTAR: A Hierarchy of Contracts

5. **Model orientation:**

A more recent model-oriented work related to environments is the so-called ECMA Reference Model for Frameworks of Software Engineering Environments, which is a quite extensive enumeration of services that can be expected to be present in a CASE framework. The model now is getting close to the three dimensions of integration as originally introduced in [Sch89] (see Figure 10):

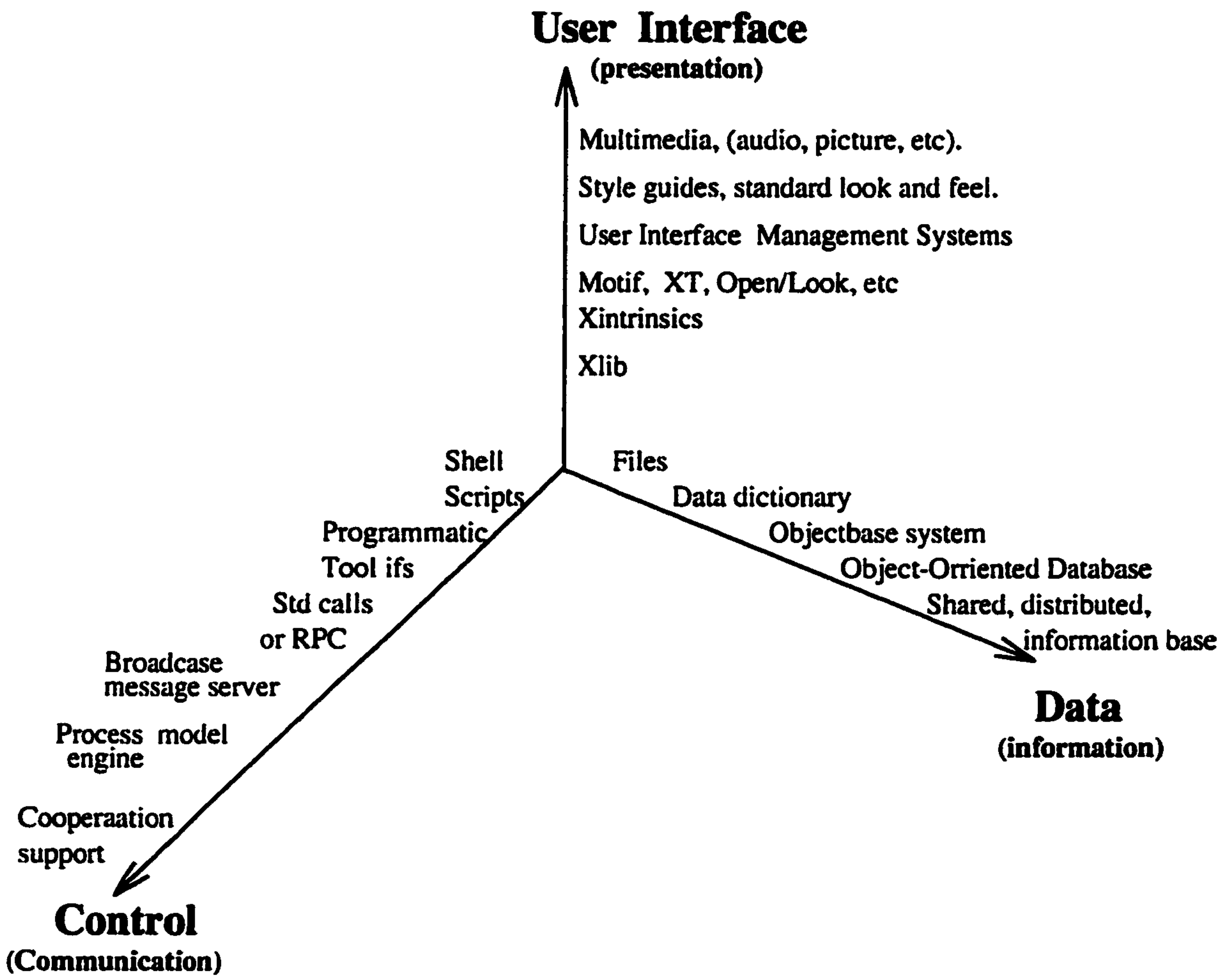


Figure 10: The Three Dimensions of Integration

- **Data**, that is intuitively understood as the degree of data representation uniformity among tools.
- **Control**, determines a tool's communicational ability, i.e. the degree to which it communicates its findings and actions to other tools, and the degree to which it provides means for other tools to communicate with it.

- **Presentation**, that is the extent to which the "user interface" is uniform among tools in an environment.

The model also addresses access and security issues, as well as administration of the environment. It is strong on elaborating the **data** aspect, but is less developed in the **control** and especially User Interface domains. It furthermore provides a single level abstraction of tools.

With the above examples of development environment orientation, we can see that they are based on different levels of abstraction and detail.

## Two CASE Cultures

Most of the discussion above has its roots in the culture and approaches related to the Stoneman document. Another trend has placed greater emphasis on the design phase, providing support for drawing of data flow diagrams, architecture diagrams, program decompositions, etc [SV93]. From initially being design methods, illustrated by more or less informal hand drawn diagrams, there evolved a need for easy maintenance of those drawings, and for making sure that they were made in a standardised way, using a proper format. Software tools supporting drawings were developed, and a market evolved offering products supporting a range of notations, such as Data Flow Diagrams, different forms of Structure Charts, and more or less standardised methods.

The concept of CASE has occasionally been associated with the market of those diagramming and graphics editing tools. Given the background presented above, it does however seem more appropriate to view CASE as the gradually evolving merger of two cultures, one which we call the **Environments Culture**, and another one which we call the **Diagramming Tools culture** [SV93]. Their relation could be characterised by Figure 11.

In the Environments area the original representatives, or carriers of culture, were Interlisp, Smalltalk, and later exemplified by systems like Rational. At the Diagramming Tools side, we have the graphical notations and methods that are the basis for the tools. While the Environments culture always has emphasised the "whole", using words like architecture, framework, and integration, the Diagramming tools are usually more localised in their scope, producing isolated tools rather than complete environments.

The Environments culture has been mostly honoured in the academic and technical context, while the Diagramming tools have had more of an industrial and administrative flavour. Referring to the phases of the traditional life cycle, the Environments were mostly interested in the later phases, while the Diagramming tools have usually focused in on the earlier requirements and design phases. Based on this positioning, they correspond to Back-End CASE and Front-End CASE respectively.

Fast turnaround time in the edit-compile-debug cycle was always a main goal within the Environments culture, while the Diagramming tools have had less emphasis on this issue, based on the assumption that a good work in the design phase makes the coding trivial. The Environments have been more interested in providing good prerequisites for allowing fast prototyping and more exploratory development, while the Diagramming tools culture fully accepts the traditional phased Waterfall model.

***The Two CASE Cultures***

| **Environments** | | **Diagramming Tools** |
|---|---|---|
| Interlisp, Smalltalk Rational. | ⟷ | Yourdon, Gane/Sarson, Jackson. |
| Architeccturally and Integration oriented. | ⟷ | Single tools. |
| Technical culture. | ⟷ | Administrative cculture. |
| Back_End CASE | ⟷ | Front-End CASE |
| Fast turnaround time | ⟷ | "Move work to spec/design phase" |
| Suited for prototyping | ⟷ | Suited for Waterfall model |

Figure 11: Relation Between Two CASE Cultures

Mutual influence, and the fact that the final goal is the same, has however gradually decreased the differences. The CASE tools have widened their scope, often by attempting integration with code-generation or programming support tool sets. The originally rather programming oriented environments have on the other hand evolved into broad concerns for full project support, which in turn forces a certain moderation of integration ambitions.

Flecher considers CASE tools to include project management, configuration management, testing, reverse engineering, and other related tools and tool sets [FH93].

## Integration in Development Environments

The essence of the idea of an environment, as opposed to a set of independent tools, is in the increased coordination and uniformity that we associate with the former concept. Schefstrom claims [SV93] that:

> *The purpose of any environment integration effort is to increase the productivity of the system developer.*

Since development is never truly automatic, the role of tools is most often in providing the user with the information he needs, and doing so with short response time and without requiring him to engage in error prone or context breaking dialogues with other different tools. Although it is hard to define the exact quantitative value of a service, one can observe that it is always relative to its cost, the latter of which has three components:

- the cost of activation service in terms of knowing about its need, finding it, and providing the right parameters
- the cost, in waiting time, for the user while the computer is working
- the cost of finally utilising and assimilating the result of the service in the context where the user needs it.

A purpose of integration is to minimise the sum of those costs, which we call the turnaround time. Clearly, cooperation among tools is a key issue. The turnaround cost can obviously be reduced by utilising context information -- making one tool explicitly utilise knowledge about the detailed state and context of the other tool. Also much less computer resources are consumed in a carefully built integrated system.

Integration efforts have their own costs, however. Integration is more complex, and requires significantly more overview and architectural skill to create an integrated system than a non-integrated one. And, there is an inherent tension between the desire to integrate components, in order to make the system more helpful, and the need to keep them separate and independent, in order to decrease overall complexity.

## Levels of Abstraction

Software development environments consist of a set of tools that are assumed to cooperatively support the development of software. In addition, in almost every environment, there are multiple levels of abstraction can be identified, and where the requirements for cooperation/integration differ depending on the level. Schefstrom [SV93] develops a multiple level tool model:

- the smallest entity of concern is called a *service*, it is an action performed by the computer that is of interest to a system developer. The *services* are partitioned into subsets that are called *tools*.
- A *tool* is a set of *services* that shows strong internal cohesion and low external coupling. Cohesion and coupling are well known within the area of software design: cohesion is a measure of the strength of the arguments on why certain entities should be grouped together, while coupling denotes the extent to which entities that we choose not to group together anyway depend on each other.
- A *toolset* is a set of *tools* that shows strong internal cohesion and low external coupling. Certain sets of tools are more closely related than others. For example a compiler and its associated debugger, a project planning tool and its critical path analyser. a graphical design editor and its associated analyser and code generator, a document editor, its spelling checker and picture drawing subsystem, etc.
- An *environment* is a set of *toolsets* that shows strong internal cohesion and low external coupling.
- In this model we need an element that explicitly identifies some basic pieces of software that we expect to be of use to many different tools. A *framework* is a set of software modules that is expected to be of interest to several tools. and is therefore especially well documented and supported. Utilizing a framework is

economical and imposes uniformity among its users. The model is summarised graphically in Figure 12. An example of framework is HyperCASE [CR92].

HyperCASE is an architectural framework for integrating the collection of tools. The system provides a visual, integrated and customisable software-engineering environment consisting of loosely coupled tools for presentations involving both text and diagrams.

It integrates tools by combining a hypertext-base user interface with a common knowledge-base document repository. It also includes extensive natural-language capabilities tailored to the CASE domain.

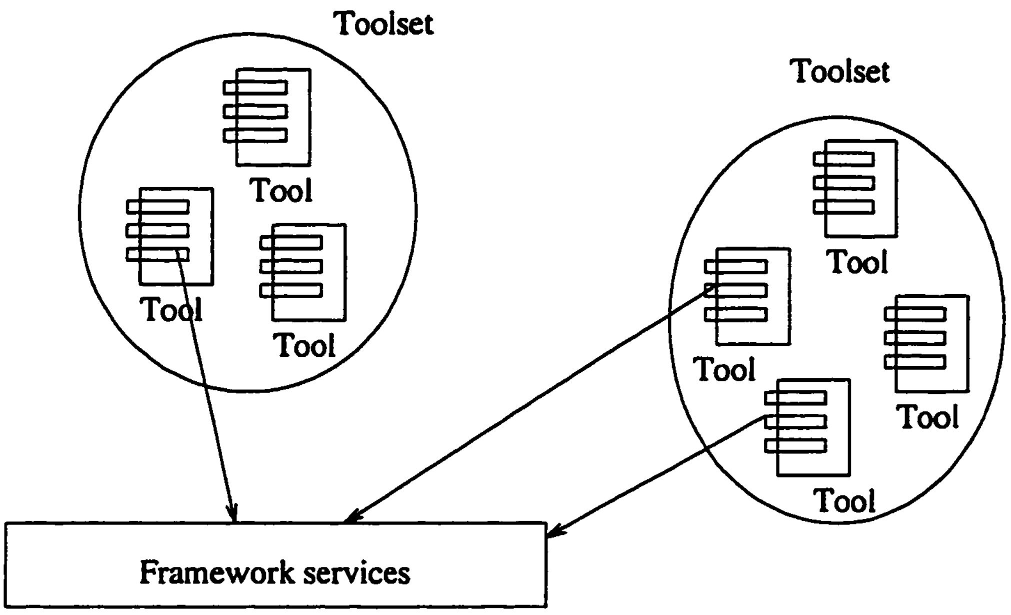


Figure 12: Levels of Abstraction in Development Environments

The HyperCASE has three subsystems:

- HyperEdit: the graphical user interface
- HyperBase: the knowledge base
- HyperDict: the data dictionary

### 2.1.4 Other Tools

As development of tools evolves rapidly, it is still too early to have a precise taxonomy, and some tools may fit in more than one category. We've discussed several examples of tools in each category. In the following paragraphs of this section, we will take several tools as examples, which might fit in several categories, to see how tools developed and functioned to help programmers.

#### DIF

Documents Integration Facility (DIF) [GS90] is a hypertext system for the development, use and maintenance of large-scale systems and their life-cycle documents. DIF helps integrate and manage the documents produced and used throughout the life cycle — requirements specifications, functional specifications, architectural designs, detailed designs, source code, testing information, and user and maintenance manuals.

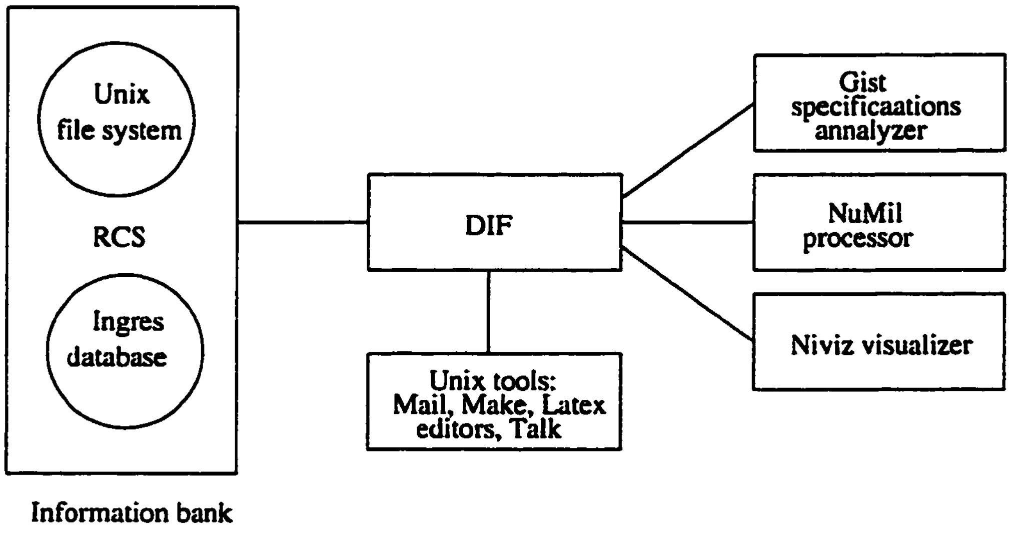


Figure 13: DIF Organisation

In DIF, software documents are considered as the objects to be stored, processed, browsed, revised, and reused. Links explain the relationship between the objects. DIF stores the objects in files and the relationships between the objects in a relational database (Ingres database). DIF allows two modes of operations: a superuser mode and general-user mode. Superusers define the project management information and structure of the documents. General users can create, modify, and browse through the hypertext base.

The superuser defines the forms and the basic templates, while the user can define links between basic templates. DIF provides several features that enable users to view system information in an integrated manner within and across projects.

At the heart of DIF's implementation are the UNIX file system and the Ingres database system. The file system provides a repository for the textural and graphical information; Ingres stores information-structure-level and project-level information.

DIF is considered to be a software-engineering environment as DIF can accommodate all life-cycle activities. DIF offers a uniform interface to access the appropriate tools, like functional specification analyser and an architectural design processor. DIF's organisation is shown in Figure 13.

## CIA

Analysing the structure of large programs is one of the most frustrating and time-consuming parts of software maintenance. Several program abstraction systems have been implemented and reported to aid the discovery process during software maintenance. MasterScope analyzes and cross-references user programs in the Interlisp environment [TM81]. FAST (Fortran Analysis System) is used to analyze Fortran programs [BH77]. OMEGA is an experimental system for a language called Model using INGRES for database management [Lin84]. Cscope creates a cross-reference file to allow user to browse C program and locate or modify function definitions [Ste85].

The most interesting of those program abstraction systems is CIA. The C Information Abstraction system (CIA) [CNR90] summarizes the structure information of programs in a relational database. Programmers can invoke relational queries to analyze various aspects of their software.

The construction of a program abstraction system involves three steps:

1. Form a conceptual model: this model defines a complete set of the software objects and relationships for C.

2. Extract relational views: according to the conceptual model, a parser converts the textual representation of programs to a relational database.

3. Construct abstract views: different levels of abstract views are provided based on the relational views.

The CIA system consists of three major components: the C Abstractor, the Information Viewer and Software Investigator (see Figure 14). It makes use of the dependency checking mechanism of the UNIX Make command to construct the program database incrementally. Object, its attributes and reference relationship between objects are defined in CIA conceptual model. Relational and textural views can be

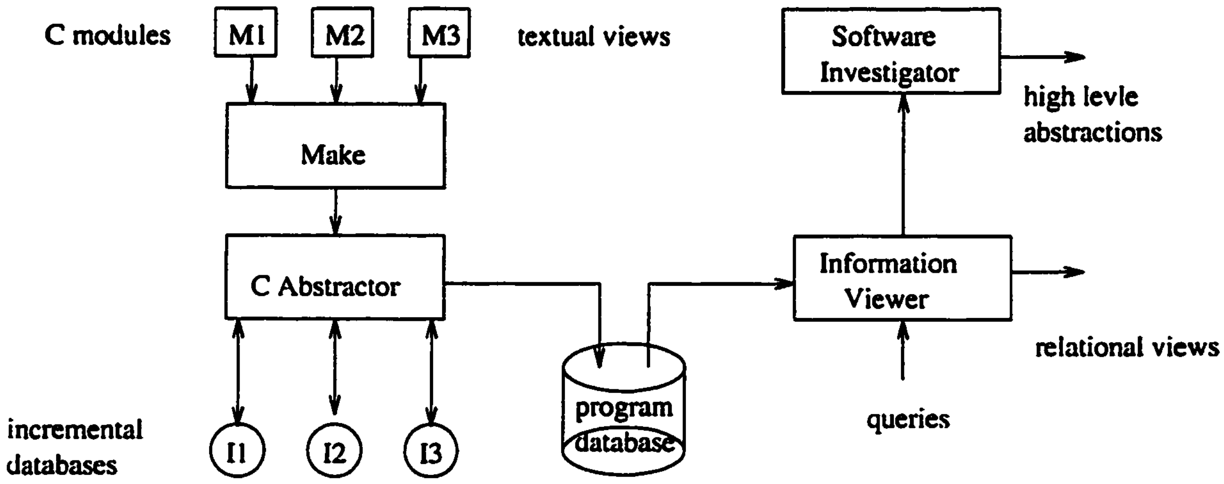


Figure 14: The C Information Abstraction System

created by retrieving and processing information in the database, such as retrieval of attribute information, retrieval of relationships between two object domains, view the definition of an object, etc. With the basis of program database, the software investigator is a set of software tools that can be used to examine the program structures: graphical views, subsystem extraction, program layering, dead code elimination and binding analysis.

## 2.2 What Do Programmers Actually Need?

In the ideal case, the choice for a specific set of tools will be made as follows. First, a certain approach to the software development process is selected. Next, techniques are selected that support the various phases in that development process. As a last step, tools are selected that support those techniques. Possibly, some steps in the development process are not supported by well-defined techniques. Some techniques may not be supported by tools. Thus, a typical development environment will have pyramidal shape as in Figure 15.

In practice, we often find the reverse conical form: a hardly developed model of the development process, few well-defined techniques, and a lot of tools. In this way, the benefits of the tools will be limited at best. As Schefstrom [SV93] paraphrase the situation: *"for many a CASE, there is a lot of Computer Aided, and precious little Software Engineering"*.

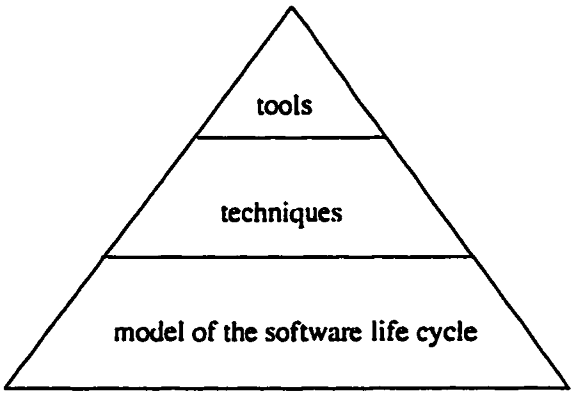


Figure 15: Support in a Typical Development Environment

With all these fancy tools available, what does a maintenance programmer actually need?

This is a typical pattern of work process: if the looking up of an imported interface is a common task, one could expect its turnaround time to be large in a traditional environment since it probably involves leaving an editor to go out in a file system, explicitly locating a file holding the interface, studying the interface and remembering it, and then returning to the first point of editing without any other result than what could be memorized during the break.

When viewing source code, programmers are not simply reading the text; they are also attempting to understand its structure. Those unique symbols appearing in the code are either user-defined or system or language keywords. User-defined symbols include routine names, variables, type and constant identifiers. Each symbol has properties associated with it, such as what it is, what it does, and what importance it has. For variables and types, these properties would include the specification of composite types. For routines, these properties would include a description of the parameter list and what type of value is returned.

A tool to make all these information be easily available is a real help to maintenance programmers. For them, the biggest challenge is to understand the code that is most probably designed and implemented by others. In the following section, we review tools that help programmers to understand code.

## 2.3 Tools for Code Understanding

In this section, we describe several tools we found in our survey. These tools are created to help programmers to understand code from different viewpoint. After this survey we have more clear idea of what our tool should accomplish.

### 2.3.1 VIFOR

VIFOR (Visual Interactive FORtran) [DLK90] is a typical language-centered programming tool. It's oriented towards maintenance of medium-to-large Fortran77 programs. With VIFOR, programs can be displayed and edited in two forms of representation: the code and the graph. It also contains transformations in both directions: from code to graph and from graph to skeletons of the code. The data model of VIFOR contains only four different classes of entities and three relations.

The data model consists of the following entity classes:

- modules: files of source code (both compilation units and include files)
- declarations: divided into two subclasses
  - processes: main program, subroutines, and functions
  - commons: global data elements

It also contains the following relations:

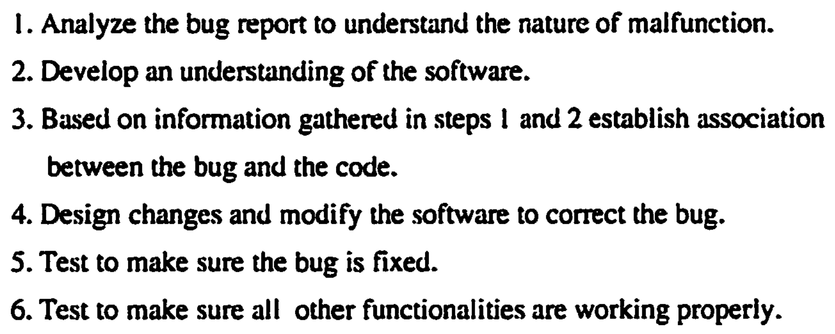
- The *belong to* relation that specifies whether any entities are parts of another entity. In particular, declarations belong to modules.

- The *call* relation interconnects processes. The processes and their call relations constitute a *call-graph*.
- The *reference* relation interconnects processes and commons. They define which processes have access to which commons.

This tool allows the programmer to deal with program architecture directly by maintaining the graphical representation of the program and providing a visual editor to build and modify the program.

## 2.3.2 SAMS

A systematic approach to corrective maintenance is suggested by K. Jambor-Sadeghi *et al* [JKCG94]. The general process for corrective maintenance is shown in Figure 16.



1. Analyze the bug report to understand the nature of malfunction.
2. Develop an understanding of the software.
3. Based on information gathered in steps 1 and 2 establish association between the bug and the code.
4. Design changes and modify the software to correct the bug.
5. Test to make sure the bug is fixed.
6. Test to make sure all other functionalities are working properly.

Figure 16: General Process for Corrective Maintenance

The software maintainer must complete steps 1, 2 and 3 in Figure 16 to build a cognitive model of expected behavior based on code and existing documents. The cognitive model that is developed in this fashion is used to fill the gap between the code and bug. The task of developing the cognitive model heavily relies on software understanding. Understanding software can be approached from two directions: the code-driven (or bottom up) approach and the functionality-driven (or top-down) approach. The two approaches are usually used together and iteratively.

Building this cognitive model for large and complex software systems is difficult. Furthermore, in the absence of correct and up-to-date documentation of software such cognitive models are often incomplete and inconsistent and may even be incorrect.

And, there is very little change that the knowledge and experiences gained in the process of fixing a bug by one maintainer is shared in subsequent maintenance activities by others.

The authors analyze the corrective maintenance activities and develop a model of corrective maintenance (see Figure 17). Based on this model, they develop a process for corrective maintenance. This process consists of a set of ordered steps that should be performed to complete a corrective maintenance task. The information needed at each step is identified and organized into information models. A set of tools that operated on the information models is identified. Realizing the information models and providing the tools through a uniform interface lead to a software maintenance system that supports corrective maintenance.

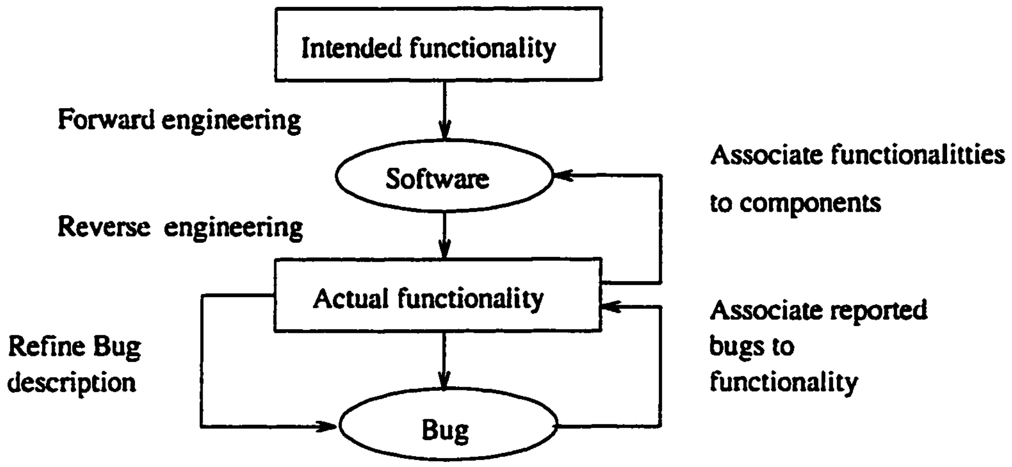


Figure 17: Model of Corrective Maintenance

SAMS is an integrated system for software maintainers that allows sharing of information about the software system among maintainers by supporting various views of the software system and by providing a set of tools that simplify performing and verifying maintenance activities. The SAMS allows integration of various aspects of the software system. The system facilitates analysis of the various aspects (source code, test suite, documentation and build procedures) and provides the necessary tools to establish logical associations among components of each aspect as well as related components across these aspects.

An integrated and complete set of tools that support the steps in the process is necessary, such as the bug browser, the testcase browser, the functionality browser, the structure browser, the execution path browser, the call graph and data flow information, etc.

The information needed for corrective maintenance is *structure, functionality, execution path, bug and testcase*. SAMS provides facilities for acquisition, representation, manipulation and browsing of such information.

### 2.3.3 GRAB

In a large software system there are complex relationships and data dependencies. The module dependency relationship can be clearly represented in a directed graph. A general-purpose browser for directed graphs, GRAB (GRAph Browser) [RDM+87], is designed and implemented by L. Rowe *et al.*

GRAB displays the nodes of a graph as icons and the edges as lines drawn between the icons that represent its endpoints. The nodes of the graph represent the procedures and the edges represent the CALLS relationship. The user can examine a graph, insert or delete nodes and edges, to move nodes and to change a node or edge label. GRAB provides an operation to lay out a graph automatically. In addition to these browsing and editing operations, a user can invoke an entity-specific browser for a node that opens a separate window through which detailed information about the node can be displayed.

More recent visual programming environments (VPE) not only provide a visual interface that includes windows and views, but also provide object-oriented libraries and extensive tools (e.g., drawing tools, configurations, versions) to help the user create application specific programs; examples of such environments include Agentsheets and VPE. The domain of VPEs has also been extended to provide program visualization, which allows the state of a program (or state transitions, execution history) to be visualized [She97].

### 2.3.4 DOCKET

Evolutionary approaches to software development, together with an emphasis on reuse, has brought about the reevaluation of software maintenance tools. Such tools range from source code analysers to semi-intelligent tools which seek to reconstruct systems designs and specification documents from source code. Most of these tools rely solely upon source code.

DOCKET is a prototype environment which supports the development of a system model linking user-oriented, business aspects of a system, to operational code using a variety of knowledge source inputs: code, documents and user expertise. Its aim is to provide a coherent model to form the basis for system understanding and to support the software change and evolution process [LF93].

### 2.3.5 Summary of Our Survey

Maintenance typically requires more resources than new software development. A significant portion of the maintenance effort involves reverse engineering which depends heavily on the code comprehension process. Typical tasks that require understanding include troubleshooting, code leveraging (reuse with modification), and program enhancement. If we can present the maintenance programmer with information that best helps to understand code, we can significantly improve quality and efficiency of program understanding and thus maintenance [MV93].

As we discussed in Section 2.1.3, the two culture of CASE are competing from many aspects (see Figure 11), but gradually the differences are decreasing and both are developing towards the area of the opposite. In other words, the integration of development environments is more and more accepted by both cultures. The desire to integrate components, to make the system more helpful and the cost of integration efforts are two issues that every tool development programmer has to balance.

We have studied language-centered, program architecture graphical representation tools like VIFOR, browsers with directed graphs like GRAB, and more integrated system for maintenance programmers like SAMS.

Our conclusion from this survey is that maintenance programmers need browsing tools that help them to understand code rapidly and efficiently. In particular, the tool should relate each use of an identifier to its definition and should perfrom dependancy analysis.

# Chapter 3

# Design of the Tool

In this chapter, we discuss the issues that we considered during the design of our tool. The basic objectives of the tool are explained in the first section, then the details of design decisions we made for the tool are described.

## 3.1 Objectives

As we mentioned in previous chapters, maintenance programmers have to face these frustrating situations frequently:

**Obsolete documentation** :

The requirement specification and design documents often become obsolete when the system evolves.

**Quick fix or patch deteriorate original design** :

After software is delivered to customer, there are often bugs existing in the code. Sooner or later, some of the bugs will appear and they will be reported back to the maintenance team. At this time, finding a fix becomes urgent as the software load is active on customer site. In addition, there are more limitations on how to fix the problem. in many cases it is impossible to change the structure of the code dramatically as it is not affordable to stop the service/traffic and reload completely new software. So the fix has to be quick and small so that it has least impact on the customer and its end user.

Under this circumstance, the original design has to give way to how to make the fix work with this sort of limitations. Obviously a sequence of many small fixes will eventually destroy the original design.

**Different code styles and formats decrease readability** :

For large software system development, there are some coding protocols to begin with, such as indentations, variable and function names, comments, macros, etc. These may look trivial and may even be viewed as limitation to the freedom of some programmers, but these regulations build the uniform style of the code and greatly facilitate readability and understanding of the code. But some time later in the whole development process, more and more inconsistent programming model/style goes into the code, and less and less effort is put in for the strict code protocols.

**Lost expertise from the quick turnover of programmers** :

The software may be designed or implemented among several individuals, and it is easy both for information to be lost and for information bottlenecks to be created. Moreover, a trend of quick turnover of programmers is observed in the industry. As experienced programmers leave with their expertise, maintenance programmers lose the help they need to verify their understandings, and could modify code the way that is not consistent with the design due to misunderstanding.

The key point for maintenance programmers to work efficiently is to understand the code well. With all the difficulties listed above, the troublesome maintenance tasks need good tools to help maintenance programmers. This is exactly the goal of our tool. We want to help maintenance programmers to understand the code with a simple-to-use tool. With this tool, information essential to maintenance programmers can be easily available to them through mouse click on the appropriate identifiers.

## 3.2 Tool Design

### 3.2.1 Overview of the Tool

Our tool can be viewed as a four-layer hypertext tool illustrated in Figure 18. In Figure 18 the arrows represent flow of information.

1. the interface layer

2. the access structure layer

3. the dictionary

4. source code layer

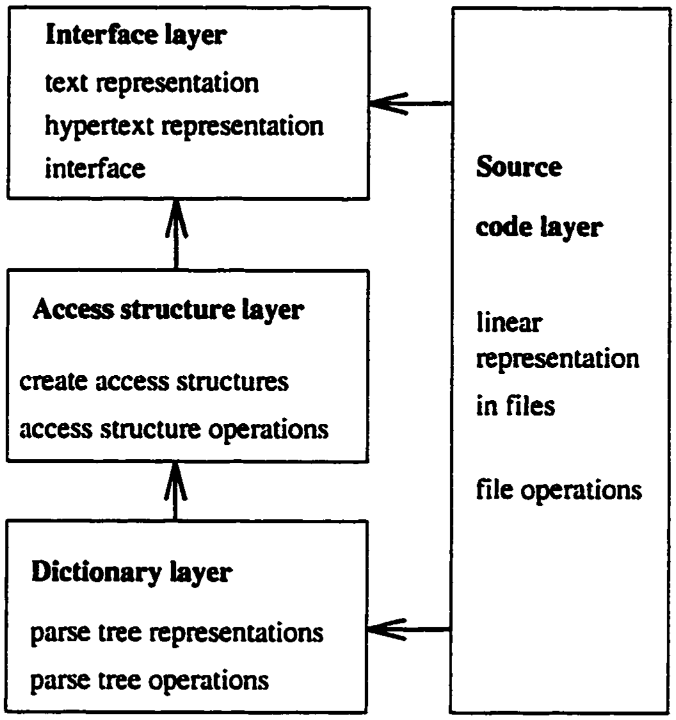


Figure 18: Layers of Our Tool

This is similar to the HyperSoft model of hypertext introduced by Jukka Paakki *et al* [PSK96]. The layers have a correspondence to the layers of the Dexter model of hypertext [HS94]. The Dexter model divides a hypertext system into three layers as illustrated in Figure 19.

1. the run-time layer

2. the storage layer

3. the within-component layer

The Dexter model run-time layer as well as our interface layer deal with the presentation of the hypertext and user interaction. The Dexter model storage layer models the basic essence of the hypertext in the same way as the access structure layer, containing the nodes and links. The Dexter model within-component layer is concerned with the contents and structure inside the nodes. We have two layers corresponding to the Dexter model within-component layer: the dictionary layer contains the parsed program text and the source code layer contains the source code files.

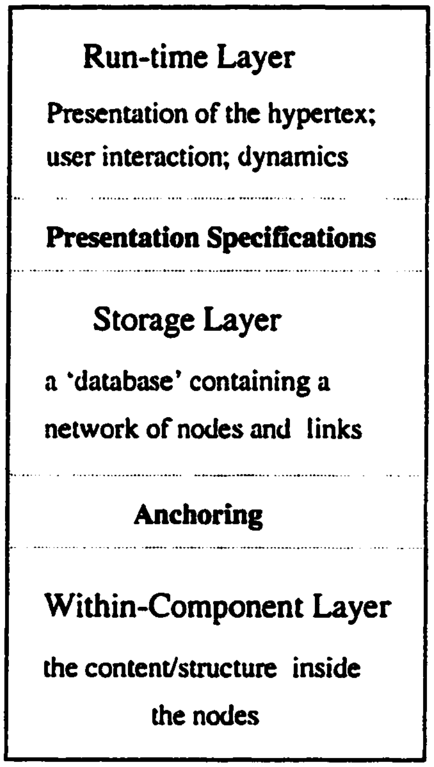

Run-time Layer
Presentation of the hypertex; user interaction; dynamics
Presentation Specifications
Storage Layer
a 'database' containing a network of nodes and links
Anchoring
Within-Component Layer
the content/structure inside the nodes

Figure 19: Layers of the Dexter Model

### 3.2.2 Function Structure

In order to meet the goals outlined in section 3.1, we decided that our tool should have the following functions:

- **Text browser:**

  In this browser, the programmer can open and browse a source code file. To support effective navigation of the source code, hyperlinks will be provided for each variables, function/procedure names, etc. The programmer simply moves the cursor onto the identifier in the context shown in the text browser window and clicks the left mouse button. This selection cause the properties associated with that identifier to be presented in the definition viewer window.

We considered incorporating the edit function into text browser, which would enable programmers to browse the source code and edit when necessary. This is not a difficult feature to add on. On the other hand, there are drawbacks of the text browser with edit function.

- First, we have to allow changes made to the source code through the editor to be propagated back to the dictionary, then to the access structure. Otherwise we lose the correct content of nodes and links.
- Another problem with the current implementation is that two editors can be spawned on the same file. A concurrency control mechanism is needed that will allow only one user at a time to update the source code.

For these reasons, we decided not to include any editing functions in the text browser.

- **Definition viewer:**

  This is the window to show definition context of an identifier or a routine that is pointed to by the hyperlink.

  How do we decide what is included in the definition context?

  - Identifier: when a user-defined variable, type or constant is selected, the declaration of that identifier surrounds the point at which it is declared. The declaration should include some comment on usage, but this inclusion depends on the parser we will select and how the parser works.
  - User-defined routines: As with identifiers, when a user-defined routine is clicked on, the declaration of that routine is presented, including formal parameters and a header comment describing the purpose and function of the routine. Such comments are assumed to be written within the code.

- **Procedure dependency viewer:**

  This is a useful tool for programmer to have a clear grasp of function dependencies between functions/procedures.

This part is not implemented. But the basic functionality that we think would be useful to programmers is followed. This is a hierarchical browser that represents the high-level information structure. It makes the program's struc-

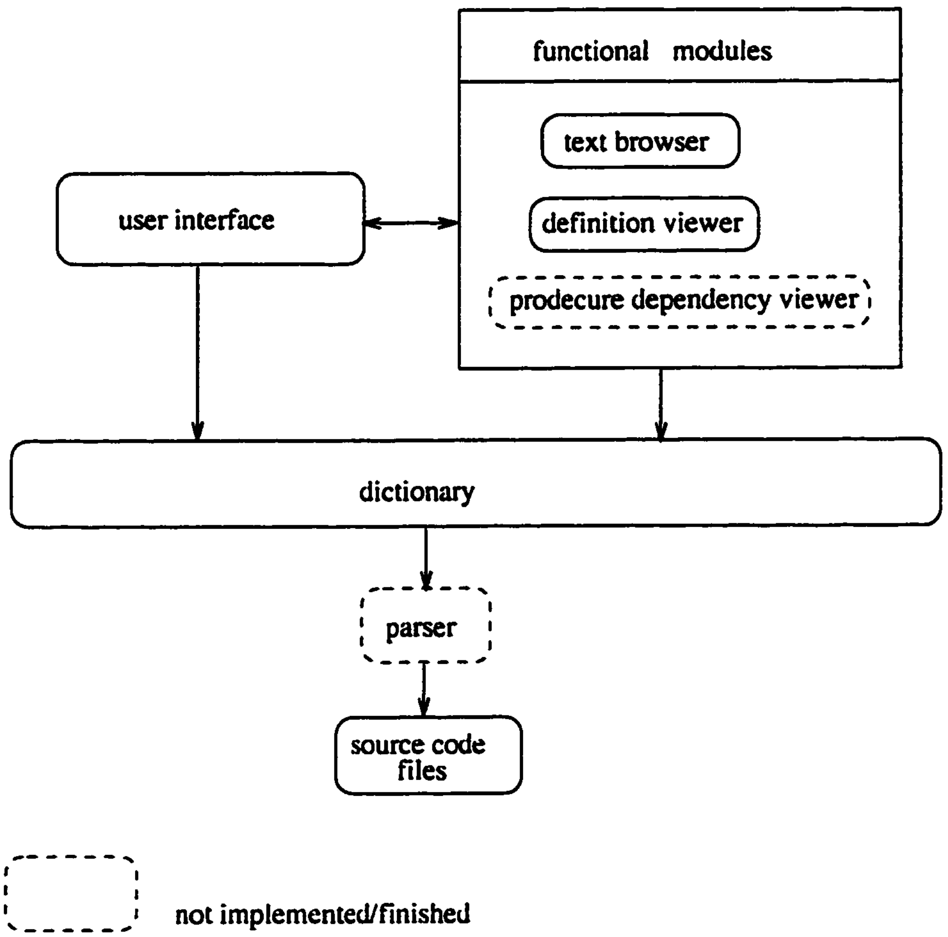


Figure 20: Organisation of Our Tool

ture more visible. The representation window displays the headings of the program routines and routines' nesting levels. The programmer sees only the highest level routines at first. If a routine is local to a global routine, it is not shown in the initial display. This structure may be expanded to include lower levels of nesting. The representation window allows the programmer to select the routine to be displayed in the source window. Note that this approach clearly supports the top-down design methodology.

The organisation of the tool is shown in Figure 20.

### 3.2.3 Data Structure

Our tool is based on the dictionary that stores all the information for the browser: for each identifier or procedures, we need to keep a significant amount of information in the dictionary. For example, what is the name of the identifier, which is the key of a node, where is the identifier defined, and how and where it is referenced each time, etc. So for large programs, the amount of information to be stored is huge. To accommodate the huge amount of information in the memory, we need a fast-access data structure to maintain satisfactory performance.

- **Hashing function**

  At first we considered using a hashing function. Hashing function takes a key as input and transform it into an integer address in a prescribed range [Col92] [GB91]. The function is designed so that the integer values it produces are uniformly distributed throughout the range. These integer values are then used as indices for an array called the hashing table. Records are both inserted into and retrieved from the table by using the hashing function to calculate the required indices from the record keys. There are many hashing algorithms that are studied, such as uniform probing hashing, double hashing, optimal hashing, extendible hashing, perfect hashing, etc. As McKenzie *et al* states [MHB90], the basic criteria for choosing a hashing function are:

  - the degree to which the algorithm uniformly distributes candidate keys over the possible values
  - the speed with which the algorithm executes

  So, we wanted to find such a hashing function that the transformation from keys into integers involves some quickly computable operation on the key and its method can handle the collision problem well. In addition, we want to have reasonable response time for searching. But consider the complexity of an optimal hashing function and the time needed to look for a good algorithm, we think hashing function is not a good choice for this tool.

- **Binary tree**

  Binary tree was also considered for our dictionary data structure. Binary tree usually works well with each node inserted in random order. However, if the input to be inserted is in order, the binary tree performs very poorly. So, the problem with binary tree is that we have to keep the tree balance using balanced tree algorithms to rearrange the tree in order to maintain certain balance conditions and assure good performance in searching and insertion. But the action to keep the tree balance itself is hard to implement and time-consuming.

- **B-tree**

  A B-tree is a balanced multi-way tree with the following properties:

  1. Every node has at most $2m + 1$ descendants.
  2. Every internal node except the root has at least $m + 1$ descendants, the root either being a leaf or having at least two descendants.
  3. The leaves are null nodes which all appear at the same depth.

  B-tree is used mainly as a primary key access method for large databases that cannot be stored in internal memory. B-trees are well suited to searches that look for a range of keys rather than a unique key. Furthermore, since the B-tree structure is kept balanced during insertions and deletions, there is no need for periodic reorganisations.

  Although B-trees can be used for internal memory dictionaries, this structure is most suitable for external searching. For external dictionaries, each node can be made large enough to fit exactly into a physical record, thus yielding, in general, high branching factors. This produces trees with very small height. As opposed to general B-trees, 2-3 trees are intended for use in main memory. 2-3 tree are the special case of B-trees when $m = 1$. Each node has two or three descendants, and all the leaves are at the same depth.

- **BB-tree**

  The BB-Tree (Binary B-Tree) access method, introduced in [And93], follows a more conventional structural approach to offer fast query-response time. The proposed method is based on a simple observation that limits the number of required restructuring operations. This observation suggests that before breaking up a node, we have to ensure that only right-hand edges are in the same level. In this way, the re-balancing of the tree can be achieved with only two simple operations. The BB-Tree is maintained by a set of elegant and simple-to-implement routines. The two main reported advantages of the BB-Tree are easy coding and satisfactory performance.

The BB-Tree could be characterised as a binary representation of the 2-3 tree. BB-Tree nodes maintain a record of balancing information along with their data. This balancing information simulates the behaviour of the 2-3 tree nodes and is called the 'level' of the node. The bottom layer of the structure has balance equal to 1. The root of the tree has the maximum level in the structure. Figure 21 depicts a BB-Tree. Each node contains the level information (next to the key). Note that only right-hand edges are allowed to be in the same level (i.e. 325 and 631).

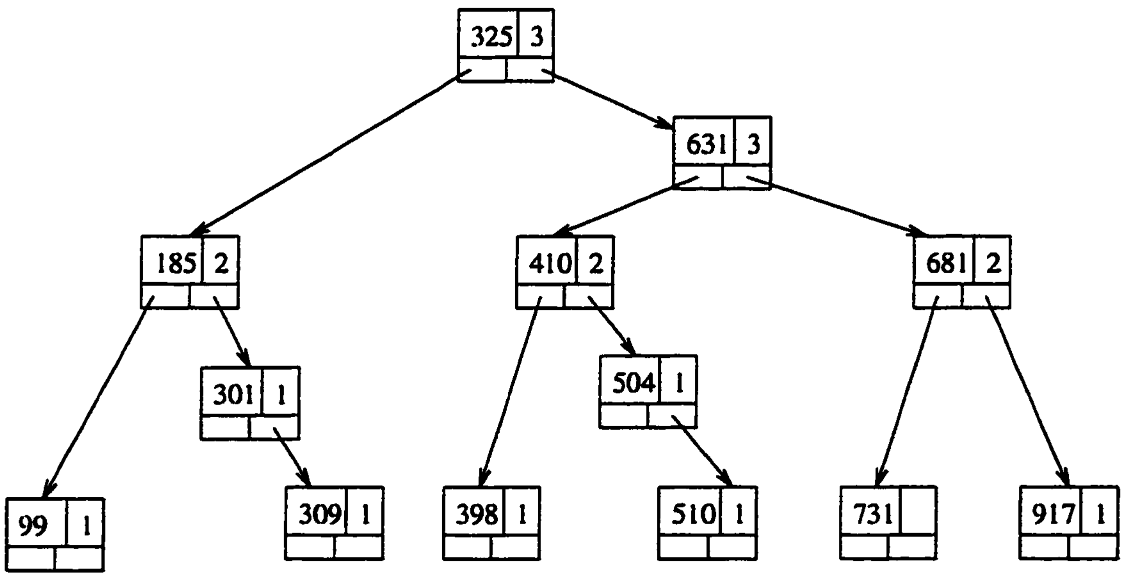


Figure 21: The BB-Tree Structure

Rearrangement occurs if there are more than two nodes with the same level value. This situation corresponds to the case of an overflowing node in a 2-3 tree. Sibling nodes belonging to the same level can be connected with left- and right-hand edges called 'horizontal edges'. In order to maintain balance in the BB-Tree, two cases have to be dealt with:

- firstly, all the horizontal left-hand edges have to be eliminated;

- secondly, the tree has to be rearranged if more than two siblings exist with the same level values (i.e. over-floating tree nodes at the same level).

The former action both checks and corrects against 'skewed' internal node arrangement in the BB-Tree and the latter provides for the balanced expansion of the tree upward through splitting. Therefore, only two operations are required to maintain the BB-Tree, namely `Skew(n)` and `Split(n)`, where n is a tree node. The former extends horizontal left-hand edges beneath *n*. The latter splits the pseudo-node n if it is too large by augmenting the level of every other node.

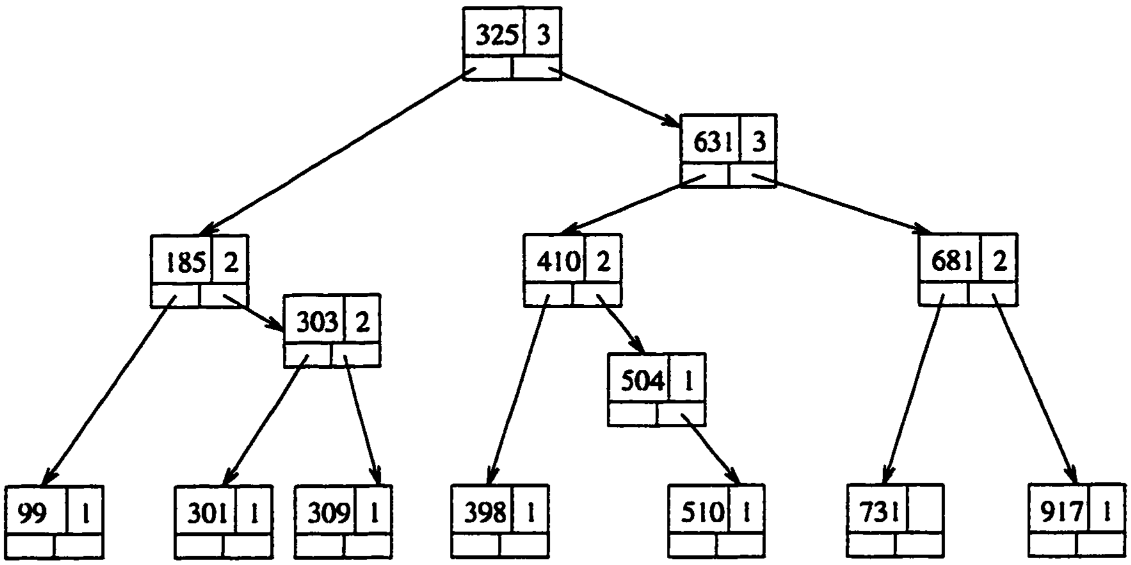


Figure 22: The BB-Tree Structure After the Insertion of Item '303'

The searching of the BB-Tree is similar to that of the binary tree. Insertions and deletions are constructed around the `Skew()` and `Split()` operations since updates are likely to violate the balancing relationship among the tree's nodes. Insertions occur initially at the first (lowest) level of the structure. Subsequently, the tree is traversed from this new node to the root and at each node both the `Skew()` and `Split()` operations are applied to rectify possible imbalances. Figure 22 shows the resulting structure if a node with the key 303 is inserted. Node deletion from the lowest level is followed by a traversal up to the root of the tree. While ascending in the structure, the level of the current node is checked against the levels of its children. If the level of the current node differs from a child by two then the level of the node is reduced by two and the `Skew()`

and `Split()` operations are performed. To handle deletions of internal nodes, two additional global pointers are used to keep track of the traversal.

- **Skip list**

  William Pugh introduced a self-adjusting access structure termed the skip list [Pug90]. Skip lists are a probabilistic alternative to balanced trees.

**What is Skip List?**

We start from a linked list. In Figure 23a, the linked list is stored in sorted order with $n$ nodes in the list. As every node has a pointer to the next node, we might need to traverse $n$ nodes to find the concerned one. If the list is stored in sorted order and every other node of the list also has a pointer to the node two ahead of it in the list, we have to examine no more than $\lceil n/2 \rceil + 1$ (see Figure 23b). Also giving every fourth node a pointer four ahead (Figure 23c) requires that no more than $\lceil n/4 \rceil + 2$ node be examined. Furthermore, if every $(2^i)$th node has a pointer $2^i$ nodes ahead (Figure 23d), the number of node that must be examined can be reduced to $\lceil log_2 n \rceil$ while only doubling the number of pointers. This data structure could be used for fast searching, but insertion and deletion would be impractical.

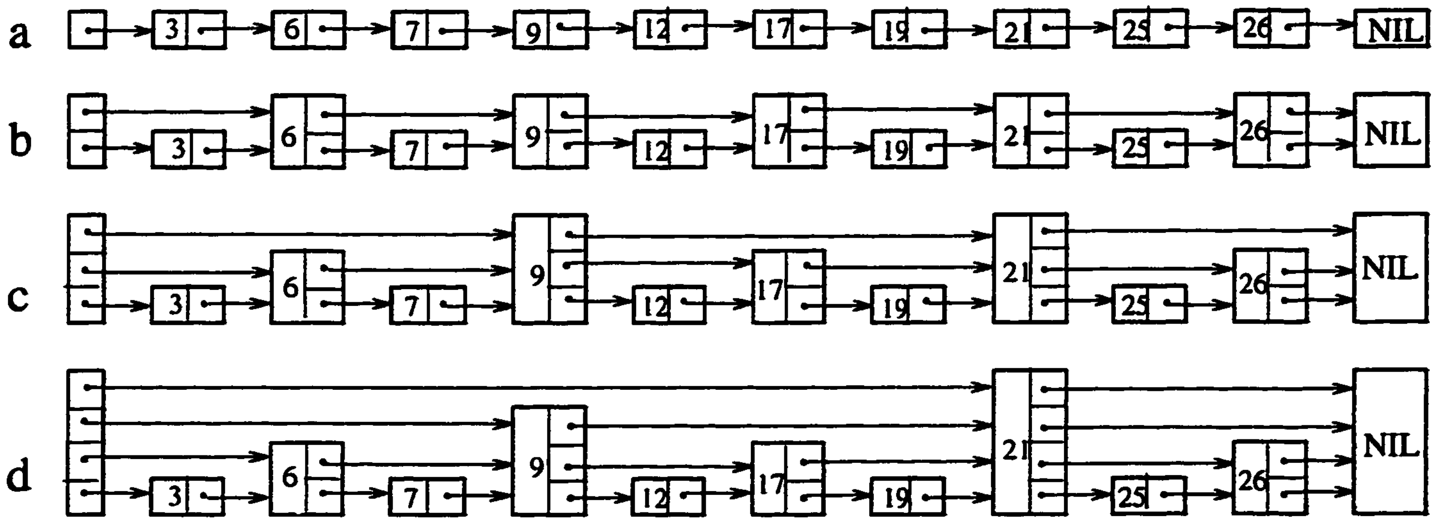


Figure 23: Linked Lists

A node that has $k$ forward pointers is called a level $k$ node. If every $(2^i)$th node has a pointer $2^i$ nodes ahead, then levels of nodes are distributed in simple pattern: 50 percent are level 1, 25 percent are level 2, 12.5 percent are level 3 and so on. What would happen if the levels of nodes were chosen randomly, but in the same proportions (as in Figure 24)? A node's $i$th forward pointer, instead of pointing $2^i - 1$ nodes ahead, points to the next node of level $i$ or higher. Insertions or deletions would require only local modifications; the level of node, chosen randomly when the node is inserted, need never change. Because these data structures are linked lists with extra pointers that skip over intermediate nodes, William Pugh named them **skip lists**.

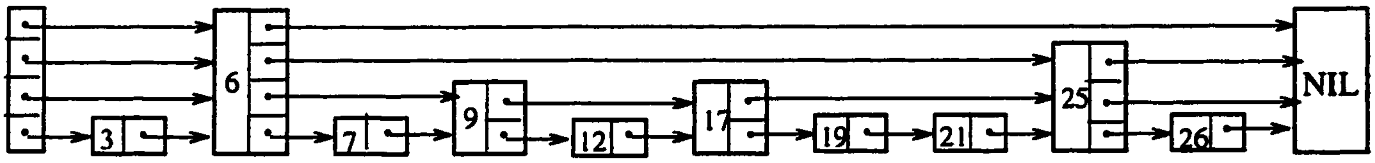


Figure 24: Skip List

Skip lists are balanced by consulting a random number generator. Although skip lists have bad worst-case performance, no input sequence consistently produces the worst-case performance (much like quick sort when the pivot element is chosen randomly). It is very unlikely a skip list data structure will be significantly unbalanced (e.g., for a dictionary of more than 250 elements, the change that a search will take more than three-times the expected time is less than one in a million). Skip lists have balance properties similar to that of search trees built by random insertions, yet do not require insertions to be random.

It is easier to balance a data structure probabilistically than to explicitly maintain the balance. For many applications, skip lists are a more natural representation than trees, and they lead to simpler algorithms. The simplicity of skip list algorithms makes them easier to implement and provides significant constant factor speed improvements over balanced tree and self-adjusting tree algorithms. Skip lists are also very space efficient. They can easily be configured to require an average of $1\frac{1}{3}$ pointers per element (or even less) and do not require balance or priority information to be sorted with each node.

*Skip lists are data structures that use probabilistic balancing rather than strictly enforced balancing. As a result, the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.* [Pug90]

## Skip List Algorithms

The *search* operation returns the contents of the value associated with the desired key or failure if the key is not present. The *insert* operation associates a specified key with a new value (inserting the key if it had not already been present). The *delete* operation deletes the specified key. It is easy to support additional operations such as "find the minimum key" or "find the next key."

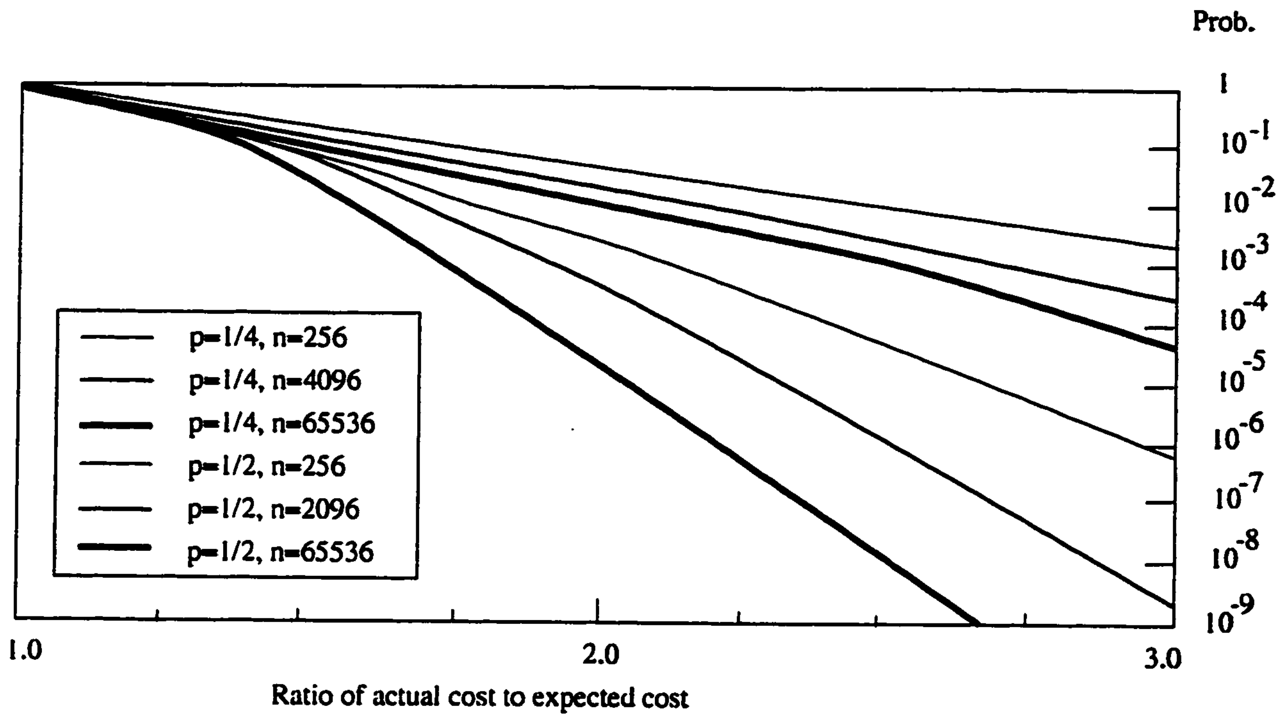
Each element is represented by a node, the level of which is chosen randomly when the node is inserted without regard for the number of elements in the data structure. A level $i$ node has $i$ forward pointers, indexed 1 through $i$. We do not need to store the level of a node in the node. Levels are capped at some appropriate constant MaxLevel. The level of a list is the maximum level currently in the list (or 1 if the list is empty). The header of a list has forward pointers at levels one through MaxLevel. The forward pointers of the header at levels higher than the current maximum level of the list point to NIL.

Detailed implementation will be explained in next chapter.

## Analysis of Skip List Algorithms

The time required to execute the *search*, *delete* and *insert* operations is dominated by the time required to search for the appropriate element. For the *insert* and *delete* operations, there is an additional cost proportional to the level of the node being inserted or deleted. The time required finding an element is proportional to the length of the search path, which is determined by the pattern in which elements with different levels appear as we traverse the list.

The structure of a skip list is determined only by the number of elements in the skip list and the results of consulting the random number generator. The sequence of operations that produced the current skip list does not matter.



This graph shows a plot of an upper bound on the probability of a search taking substantially longer than expected. The vertical axis show the probability that the length of the search path exceeds the average length by more than the ratio on the horizontal axis. For example, for p=1/2 and n=4094, the probability that the search path will be more than three times the expected length is less than one in 200 million. This graph was calculated using our probabilistic upper bound.

Figure 25: Probabilistic Analysis of the Expected Search Cost

Pugh did an analysis of the expected search cost. The result shows that the total expected cost of a list of $n$ elements is at most $O(\log n)$. The number of comparisons required is one plus the length of the search path (a comparison is performed for each position in the search path. the length of the search path is the number of hops between positions in the search path). From his probabilistic analysis. some results are shown in Figure 25.

Balanced trees and self-adjusting trees can be used for the same problems as skip lists. All three techniques have performance bounds of the same order.

But we found that what the author claims is true:

> *For most applications, implementers generally agree skip lists are significantly easier to implement than either balanced tree algorithms or self-adjusting tree algorithms.*

## Behavior Comparison Between Skip Lists and BB-Trees

In order to obtain a clear understanding regarding the merits of both the skip list and BB-Tree, and to see how these access structure behave under the presence of both queries and update (i.e. mixed workloads), Alex Dellis and Quang LeViet [DL97] carried out a large-scale comprehensive experimental study in the UNIX environment. Large, skewed and mixed-data workloads of varying compositions are used and the sensitivity of the access structures to distributions of input data is examined.

Their experience indicates that BB-Tree offers elegant maintenance routines. If these routines are implemented efficiently, they yield very competitive response times in a number of cases.

On the other hand, the skip list demonstrates consistently better creation time and improved response times for all experiments that involve mixed workloads with frequent updates. The skip list also has minimal space overhead requirements.

The major results are:

1. In environments where data rarely change, the iterative version of the BB-Tree consistently outperforms the skip list for queries on structures created from randomly ordered input data.

2. The skip list provides a much better alternative for mixed workloads with a high volume of updates. The use of the skip list is also advantageous when the input data used are highly skewed.

3. The iterative implementation of BB-Trees presents very competitive response times in the light of workloads, with a limited number of updating operations and structures generated from randomly ordered data. In this type of environment, they have found that the skip list offers inferior response items.

4. Under large space requirements, the skip list demonstrates smallest space overhead of the structure considered. This is due to high utilisation of the forwarding pointers of the skip list cells. Their experience also indicates that it is practically impossible to create a degenerate skip list.

5. The recursive maintenance routines of the BB-Tree produced consistently inferior performance than the skip list in all their experiments.

On the basis of this analysis, we decided to adopt skip lists as the principal data structure for our tool.

### 3.2.4 User Interface

#### Principles of UI Design

The user interface is an important part of any software system. Without the user interface, the system itself would be useless. There are many principles of software user interface design. We follow these principles as listed [Shn92] [Thi90]:

- 'know the user' was the first principle in Hansen's list of user-engineering principles. Most users of our tool will be mainly maintenance programmers. They are software-centered, having sufficient knowledge about all kinds of design and maintenance tools. So we design our tool with their tasks, needs, difficulties, knowledge level and habits in mind.

- A design should be task-specific: in other words, it should not only be designed for a certain purpose, but it also should be clear what that purpose is to its users. Just as a theory should be domain specific, it would be good if users could know what they are getting with a system.

- A design should be simple: A system should be simple enough for a user to be able to perform useful experiments. Any complexities confusing the user by preventing user perform their tasks are not designed or organised properly.

- Ease of use: Tools are design to provide some functionality to their users. Interface is designed to permit or aid users to perform more complex tasks than they would otherwise be able to undertake. So ease of use is by all means the basic goal for our tool.

- Productivity: The interface may permit the user to perform certain tasks faster and over longer periods without rest.

## Hypertext

Hypertext is a storage structure where information is stored in the nodes of a graph. Links between hypertext nodes allow efficient browsing of the information. In the hypertext timeline [Ber91], Hypertext was first conceived in 1945 [Bus45] as a way to store all kinds of information, both for ready access and cross-reference. Hypertext is a medium-grained, entity-relationship-like data model that lets information be structured arbitrarily and keeps a complete version history of both information and structure. "Researchers at Tektronix built Neptune, which demonstrates that hypertext provides an appropriate data model for CASE systems" [SSW86]. At a very abstract level, each of these hypertext systems provides its users with the ability to create, manipulate, and/or examine a network of information-containing nodes interconnected by relational links [HS94].

Sometimes a hypertext metaphor is used to emphasise the information browsing and navigation ability. "Such immediate availability of accurate information is believed to be the key to productivity increase" [SV93].

No restriction is put on the nature of information in the nodes, so the same hypertext may contain a node of natural-language text and a node of program code. This is the main advantage that hypertext systems have over conventional database-management systems, document bases and knowledge bases [GS90].

The hypertext offers a convenient way of viewing a program. The text browser of our tool can show the main program body. The programmer inspects it and selects a routine to display. The programmer is then shown the declaration of that routine, and is able to inspect its code. A second routine may also be easily selected and inspected. Having a question about a type, the programmer selects it and its declaration is displayed. The programmer can check on the exact syntax of the any user-defined routine. With this tool, details can be found easily, permitting the programmer to concentrate on the higher level structures. All these benefits aid the programmer in detecting and correcting faults.

## User Interface of Our Tool

The goal of our tool is to help maintenance programmers to work efficiently. The user interface part is an important part to achieve the goal. We design the window layouts in the tool carefully, bear in mind those principles of user interface design.

From the organisation of our tool (see Figure 20), we design the window layouts of our tool containing two main windows:

- Text browser window
- Definition viewer window

Detailed function and implementation of these two windows, as well as other windows, such as input prompt window, file selection window, dialog/message windows, help window, etc., will be discussed in next chapter.

# Chapter 4

# Implementation of the Tool

In this chapter, we discuss the implementation of the maintenance tool. First we explain what language we choose to use to implement the tool. Then we discuss the parsed dictionary and its layout format. In the third section we discuss the implementation of skip list. Finally, we describe the user interface of the tool in detail.

Our tool is based on X11 libraries and will operate on Sun SPARC station running under SunOS V4.4.1.

## 4.1 Implementation Language

We use C language to implement our tool and Motif to implement the user interface part.

C is a general-purpose programming language that was originally designed by Dennis Richies of Bell Laboratories and implemented there in 1972. It was first used as the systems language for the UNIX operating system. Now ANSI C (American National Standards Institute) is a mature, general-purpose language that is widely available on many machines and in many operating systems and it is one of the chief industrial programming languages of the world [KP95].

C is small language. And small is beautiful in programming. C is the native language of UNIX, which is the major interactive operating system on workstations, servers, and mainframes. C is portable, terse, modular and efficient on most machines

[KP95].

C++ is built upon the foundation of C. Several new features and extensions, such as steam I/O, operator and function overloading, classes, constructors, destructors, data hiding, inheritance and type hierarchies, designed to support object-oriented programming were added to C [Sch95].

Motif is a toolkit developed by the Open Software Foundation (OSF). It provides a set of guidelines that specifies how a user interface for graphical computers should *look and feel* -- how an application appears on the screen (the look) and how the user interacts with it (the feel). The OSF/Motif toolkit is based on the X Window System, a network-based windowing system that has been implemented for UNIX, VMS, DOS, Macintosh, and other operation systems [Del92].

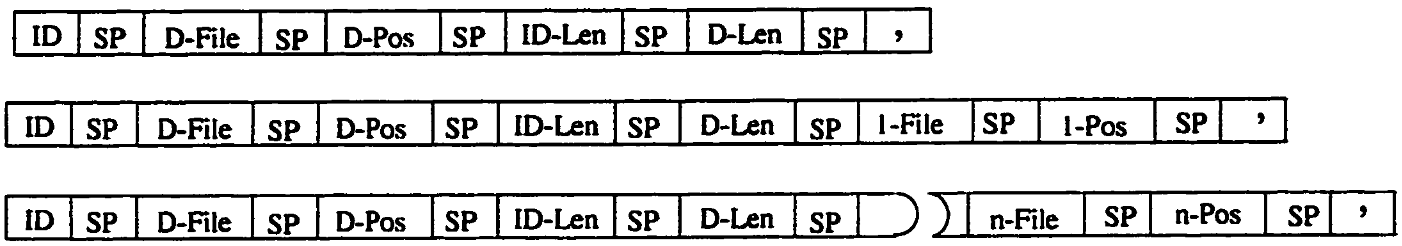
As a proposed standard for graphical user interfaces, Motif has been implemented on a wide range of computer platforms from large IBM mainframes down to PCs. To enhance portability and robustness, the Motif Graphical User Interface (GUI) was implemented by OSF using X as the window system and the X Toolkit Intrinsics (Xt) as the platform for the Application Programmer's Interface (API). Xt provides an object-oriented framework for creating reusable, configurable user-interface components called *widgets*. Motif provides widgets for such common user-interface elements as labels, buttons, menus, dialog boxes, scrollbars, text-entry or display areas, and widgets called managers to control the layout of other widgets. Xt defines certain base classes of widgets, whose behaviour can be inherited and augmented or modified by other widget classes (its subclasses). Xt also supports lighter-weight objects called *gadgets*, which look and act like widgets, but whose behaviour is actually provided by a manager widget that contains them. The object-oriented approach of Xt forces the programmer to think about the application in a more abstract and generalised fashion, which lead to better design in the long run, and fewer bugs in the short run.

Although Motif binds to many languages, C is by far the most common for building Motif applications [Bra92].

## 4.2 Dictionary and Parser

We assume that the parser module for the tool would be constructed in a conventional way, for example, using lex and yacc. The parser module for our tool should include three parts: a scanner for scanning the input file and generation tokens; a parser which uses the tokens to check the syntax of the input stream and report errors if there are any; and constructors which create the skip list (see Section 3.2.3) in memory. We also tried to find a good parser that could save us some time without repeating the work someone else has done, but we did not succeed in finding a parser that suited our requirements. The information in the web is so overwhelming that we decided to continue our work with the following assumptions:

- The source files are free of syntax errors. This assumption is justified by the fact that maintenance programmers deal with code that has been compiled and executed.
- We assume we have suitable parser which will scan all the input files and generate tokens, since there are no syntax error preventing us from constructing an output file with those tokens, the output file will be a dictionary file in fixed format.
- The fixed-format dictionary file will be the base for our tool to create the skip lists which are the data structure of the tool.


ID | SP | D-File | SP | D-Pos | SP | ID-Len | SP | D-Len | SP | ,
ID | SP | D-File | SP | D-Pos | SP | ID-Len | SP | D-Len | SP | 1-File | SP | 1-Pos | SP | ,
ID | SP | D-File | SP | D-Pos | SP | ID-Len | SP | D-Len | SP | n-File | SP | n-Pos | SP | ,

| | | | |
|---|---|---|---|
| ID | identifier name | D-Len | length of the definition text |
| D-File | file name where the identifier is defined | n-File | file name where the identifier occurs, ex. 1-File, 2-File, ... |
| D-Pos | position where ID is defined in D-File | n-Pos | position in the file n-File |
| ID-Len | length of the identifier name | SP | one space as seperator |

Figure 26: Format of the Dictionary

We do not have a complete implementation of the parser module.

### 4.2.1 Format of the Dictionary

The format of the dictionary is shown in Figure 26.

Let us consider a short example: file `main.c` contains the `main()` function which calls `fibonacci()` in file calculate.c to calculate the Fibonacci value and prints the final result returned. File `main.c` is shown in Figure 27 and File calculate.c is shown in Figure 28.

```
/************************************/
/*                main.c            */
/*  calculate Fibonacci numbers     */
/************************************/

int fibonacci(int n)

int main(void)
{
    int number;

    printf("input an integer: ");
    scanf("%d", &number);
    printf("The value of Fibonacci(%d)\n", number, fibonacci(number));

    return 0;
}
```

Figure 27: C Code of File `main.c`

After the parser processed the two source files, the output dictionary file should look like Figure 29. In dictionary file, the position of an identifier is the total number of characters counted from the beginning of the file, because this is how Motif widget handles the position of the mouse cursor.

## 4.3 Skip List

As we discussed in Chapter 3, skip lists are easy to implement. Let's look at the algorithms for initialisation, search, insertion and deletion [Pug90].

### 4.3.1 Initialisation

An element NIL is allocated and given a key greater than any legal key. All levels of all skip lists are terminated with NIL. A new list is initialised so that the level of the list is equal to 1 and all forward pointers of the list's header point to NIL.

```
/*************************************/
/*      calculate.c                 */
/*************************************/

int fibonacci(int n)
{
    if (n <= 1)
        return n;
    else
        return (fibonacci(n-1) + fibonacci(n-2));
}
```

Figure 28: C Code of File calculate.c

```
main main.c 27 4 172 ,
fibonacci calculate.c 4 9 105 calculate.c 74 calculate.c 92 main.c 4 main.c 165 ,
n calculate.c 18 1 1 calculate.c 30 calculate.c 49 calculate.c 85 calculate.c 102 calculate.c main.c 18 ,
number main.c 47 6 6 main.c 105 main.c 157 main.c 175 ,
```

Figure 29: The Dictionary File of File main.c and calculate.c

## 4.3.2 Search Algorithm

We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for (Figure 30).

```
Search (list, searchKey) {
    x = list->header
    /* loop invariant: x->key < searchKey */
    for i = list->level downto 1 do
        while x->forward[i]->key < searchKey  do
            x = x->forward[i]
    /* x->key < searchKey <= x->forward[1]->key */
    x = x->forward[1]
    if x->key == searchKey  then
        return  x->value
    else
        return  failure
}
```

Figure 30: Skip List Search Algorithm

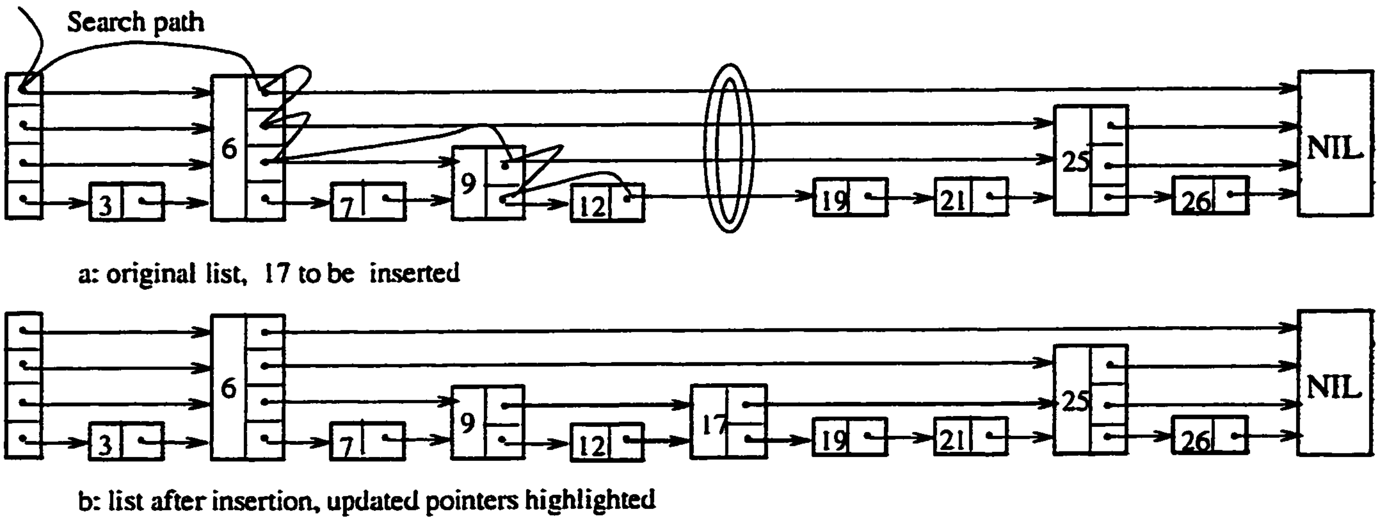

Search path
6
3
7
9
12
19
21
25
26
NIL
a: original list, 17 to be inserted
6
3
7
9
12
17
19
21
25
26
NIL
b: list after insertion, updated pointers highlighted

Figure 31: Pictorial Description of Steps Involved in Performing An Insertion

When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list). See Figure 31a for an example of search algorithm.

### 4.3.3 Insertion and Deletion Algorithms

To insert or delete a node, we simply search and splice, as shown in Figure 31. Figure 32 gives algorithms for insertion and deletion. A vector update is maintained so

```
Insert (list, searchKey, newValue) {
    local update[1..MaxLevel]
    x = list->header
    for i = list->level downto 1 do
        while x->forward[i]->key < searchKey do
            x = x->forward[i]
        /* x->key < searchKey <= x->forward[1]->key */
        update[i] = x
    x = x->forward[1]
    if x->key == searchKey then
        x->value = newValue
    else
        newLevel = randomLevel()
        if newLevel > list->level then
            for i = list->level + 1 to newLevel do
                update[i] = list->header
            list->level = newLevel
        x = makeNode{newLevel, seerchKey, value)
        for i = 1 to newLevel do
            x->forward[i] = update[i]->foward[i]
            update[i]->forward[i] = x
}
```

Figure 32: Skip List Insertion Algorithm

that when the search is complete (and we are ready to perform the splice), update[*i*] contains a pointer to the rightmost node of level $i$ or higher that is to the left of the location of the insertion/deletion.

If an insertion generates a node with a level greater than the previous maximum level of the list and initialise the appropriate portions of the update vector. After each deletion, we check to see if we have deleted the maximum element of the list and if so, decrease the maximum level of the list.

### 4.3.4 Choosing A Random Level

Initially, we discussed a probability distribution where half of the nodes that have level $i$ pointers also have level $i + 1$ pointers. To get away form magic constants, we say that a fraction $p$ of the nodes with level $i$ pointers also have level $i + 1$ pointers (say $p = 1/2$). Levels are generated randomly with an algorithm equivalent to the one in Figure 33. Levels are generated without reference to the number of elements in the list. Since the random new level of a skip list could be very large, the last line of code in Figure 33 makes sure that the new level will be MaxLevel + 1 at most.

```
randomLevel()
    newLevel := 1
    /* random() returns a random value in [0...1] */
    while random() < p  do
      newLevel := newLevel + 1
    return  min(newLevel, MaxLevel)
```

Figure 33: Algorithm to Calculate a Random Level

### 4.3.5 At What Level Do We Start A Search? Defining L(n)

In a skip list of 16 elements generated with $p = \frac{1}{2}$, we might happen to have 9 elements of level 1; 3 elements of level 2; 3 elements of level 3; and 1 element of level 14 (this would be very unlikely, but it could happen). How should we handle this? If we use the standard algorithm and start our search at level 14, we will do a lot of useless work.

Where should we start the search? The analysis [Pug90] suggests that ideally we would start a search at the level $L$ where we expect $1/p$ nodes. This happens when $l = \log_{1/p} n$. Since we will be referring frequently to this formula, we will use $L(n)$ to denote $\log_{1/p} n$.

There are a number of solutions to the problem of deciding how to handle the case where there is an element with an unusually large level in the list.

- Simply start a search at the highest level present in the list. As we will see in our analysis, the probability that the maximum level in a list of n elements is significantly larger than $L(n)$ is very small. Starting a search at the maximum level in the list does not add more than a small constant to the expected search time.
- Although an element may contain room for 14 pointers, we do not need to use all 14. We can choose to utilise only $L(n)$ levels. There are a number of ways to implement this, but they all complicate the algorithms and do not noticeably improve performance.
- If we generate a random level that is more than one greater than the current maximum level in the list, we simply use one plus the current maximum level in the list as the level of the new node. In this way, the level of a node is no longer completely random. In practice and intuitively, this works well. This is the solution we use in the implementation of our tool.

### 4.3.6 Determining Maximum Level of a Node

Since we can safely cap levels at $L(n)$, we should choose MaxLevel = $L(N)$ (where $N$ is an upper bound on the number of elements in a skip list). If $p = \frac{1}{2}$, using MaxLevel=16 is appropriate for data structures containing up to $2^{16}$ elements.

### 4.3.7 Choosing the Fraction of Nodes With Higher Level Pointers

As shown in Figure 34, Pugh gives the relative time and space requirements for different values of $p$ [Pug90].

Decreasing $p$ also increases the variability of running times. If $\frac{1}{p}$ is a power of 2, it will be easy to generate a random level from a stream of random bits (it requires an average of $(\log_2 1/p)/(1-p)$ random bits to generate a random level). Since some of the constant overheads are related to $L(n)$ (rather than $L(n)/p$ ), choosing $p = \frac{1}{4}$ ( rather than $\frac{1}{2}$ ) slightly improves the constant factors of the speed of the algorithms as well. So he suggests that a value of $\frac{1}{4}$ be used for $p$ unless the variability of running times is a primary concern, in which case $p$ should be $\frac{1}{2}$ [Pug90].

| p | **Normalized search time (i.e., normalized L(n)/p)** | **Avg. # of pointers per node (i.e., 1/(1-p))** |
|---|---|---|
| 1/2 | 1 | 2 |
| 1/e | 0.94 | 1.58 |
| 1/4 | 1 | 1.33 |
| 1/8 | 1.33 | 1.14 |
| 1/16 | 2 | 1.07 |

Figure 34: Relative Search Speed and Space Requirements Depending on the Value of p

## 4.4 Skip Lists in Our Tool

Based on the above skip lists algorithms [Pug90], skip lists for our tool are implemented in the following way.

**File list:** A node of file list corresponds to one file. It stores the file name, pointer to next file, pointer to its position list and its position list level.

**Position list:** All nodes of a particular position list are from the same file. Each node of the position list corresponds to one valid identifier position in this file. It stores position number, pointer to a **identifier list**, pointer to previous occurrence and next occurrence and a variable sized array of pointers pointing to other position node.

The **identifier list** stores information related to an identifier: such as identifier name, its definition file name, its length, its position number in definition file, pointer to next identifier, etc.

## 4.5 User Interface

In this section, we will first look into the two major windows, their structures, layouts and their functionality, as well as some important issues of the implementation; then some routines will be described.

Our tool has two major windows: one is the browser window, the other is the definition window.

### 4.5.1 Browser Window

The structure of widgets in the *Browser Window* is shown in Figure 35, and the layout of the window is displayed in Figure 36.

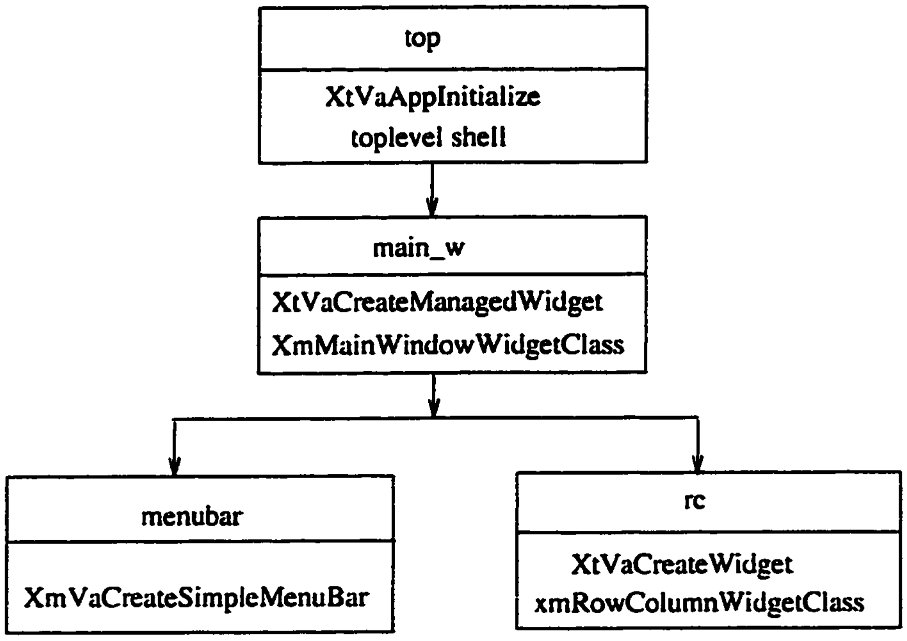


Figure 35: Structure of Browser Window

This is the main window of the tool. First, the top-level shell is created. xmMainWindowWidgetClass *main_w* for the application is built on top of the shell. In *main_w* we create *menubar* and xmRowColumnWidgetClass *rc*. The *menubar* contains three pulldown menus: *file*, *dict* and *help*. Before *rc* is realized, we call function `prompt_dictionary_name()` to create dialog window *Prompt Window* using XmCreatePromptDialog( ). In the upper part of *rc* we have widget *search_w*, *fillename_w* and *message_display*; in the lower part we have *reference_window* and ScrolledText widget *text_w* which is the working area of the *main_w* to display source code.

Now we will go through each widget in *main_w* in following sections.

## Prompt Window

We assume that a well-formatted dictionary is ready to use as input to our tool. This dictionary file contains all information about source files. So when we invoke our tool, the first window that comes up is the *prompt window* asking for dictionary file name. The structure of widgets and the layout of the prompt window are shown in Figure 37 and Figure 38 respectively.

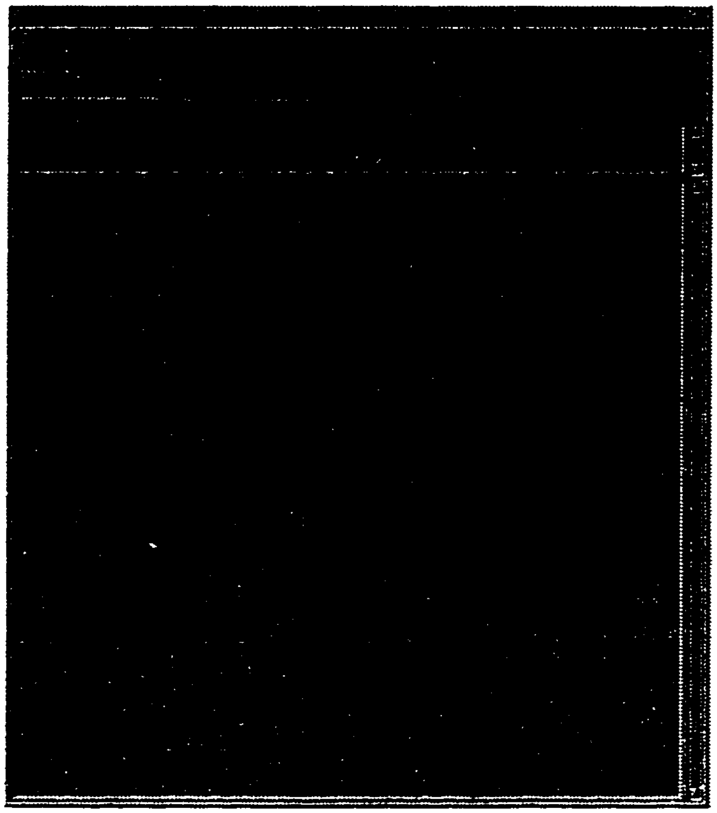


Figure 36: Layout of the Browser Window

This dialog window asks the user to type in dictionary name, accepts user input and passes it to call-back function `call_creater()` to create all data structure in skip list. If the user selects *Cancel*, the dialog will simply be destroyed.

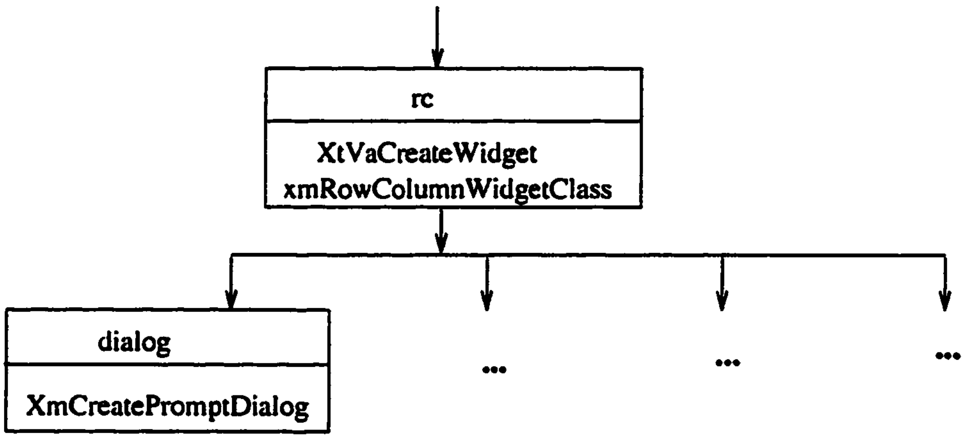


Figure 37: Structure of Prompt Window

## Menubar

Figure 39 shows the widget structure of *menubar*.

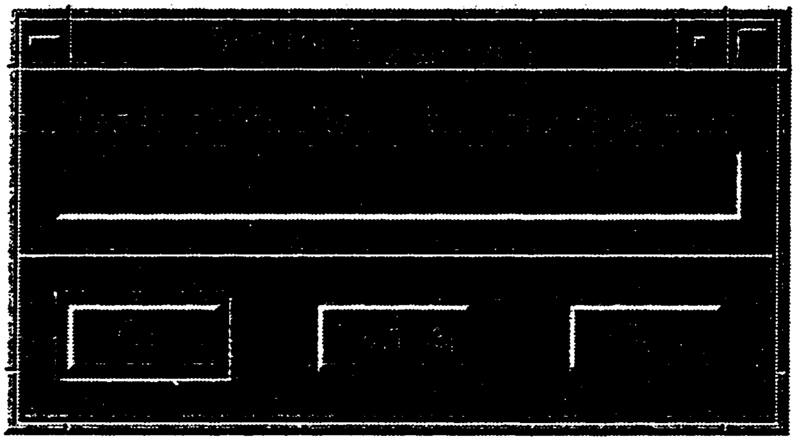


Figure 38: Prompt Window Layout

The *menubar* has three pulldown menus:

- *File*: under this menu, there are two sub-menus *Open* and *Exit* associated with call-back function `file_cb()`.

– when the user selects *Open* menu

The `file_cb()` creates a FileSelectionDialog widget (Figure 40) listing

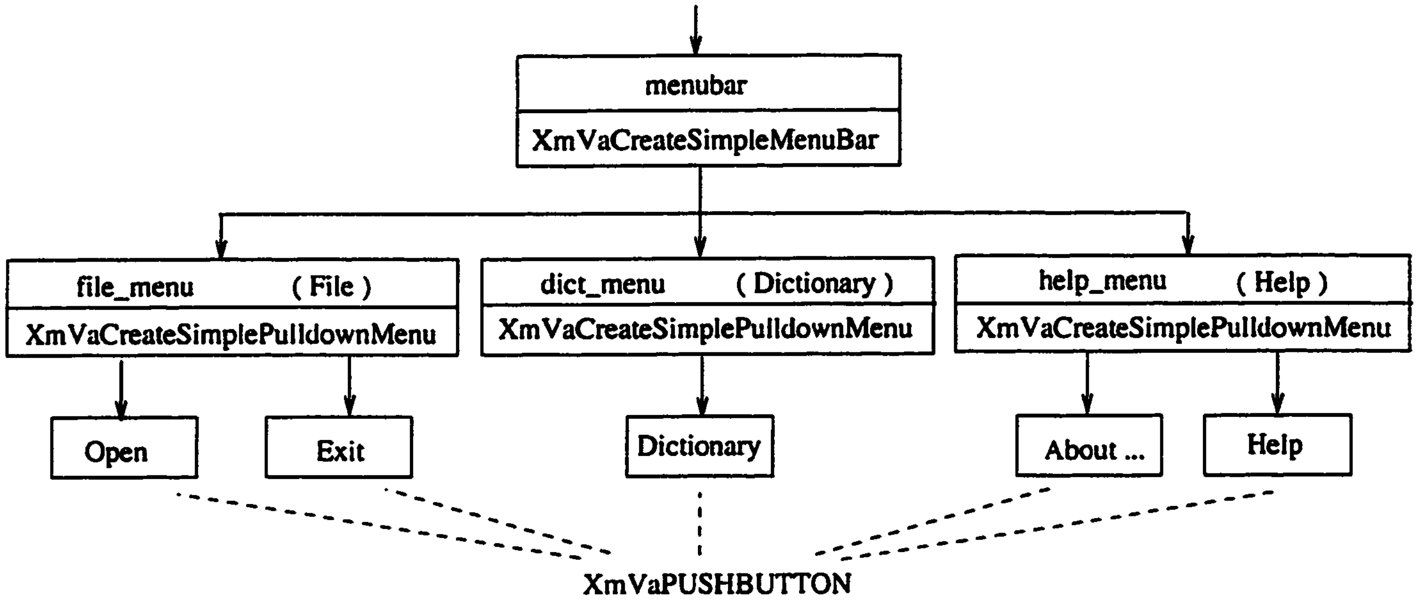


Figure 39: Structure of Menu Bar

files and directories for the user to choose from, and it associates call-back function `read_file()` to *OK* button and `XtUnmanageChild` with *Cancel* button. When a readable regular file is selected and *OK* button is clicked, `read_file()` will check the validity of the skip list data structure in memory and display the content of selected file in the working area of *main_w*, i.e. *text_w*. If the data structure has not been created properly in memory, the user will be advised to re-enter a correct dictionary file name by a reminder popup window (Figure 41) followed by the prompt window (see Figure 38). When *OK* button is clicked with no file selected, nothing will be done except returning back to `file_cb()`. When the user clicks *Cancel* button, the FileSelectionDialog widget will disappear and return back to `file_cb()`.

– when user selects *Exit* menu

The `file_cb()` creates a warning dialog window to confirm user's intention. Figure 42 shows the layout of the warning dialog. We quit our tool when user confirms by clicking *OK* button, otherwise nothing is changed except that the warning dialog window is removed.

- *Dict*: When the *Dict* menu item is selected, the prompt window (See Figure 38) will appear for user to enter a new dictionary name. All this is handled by call-back function `dict_cb()`.

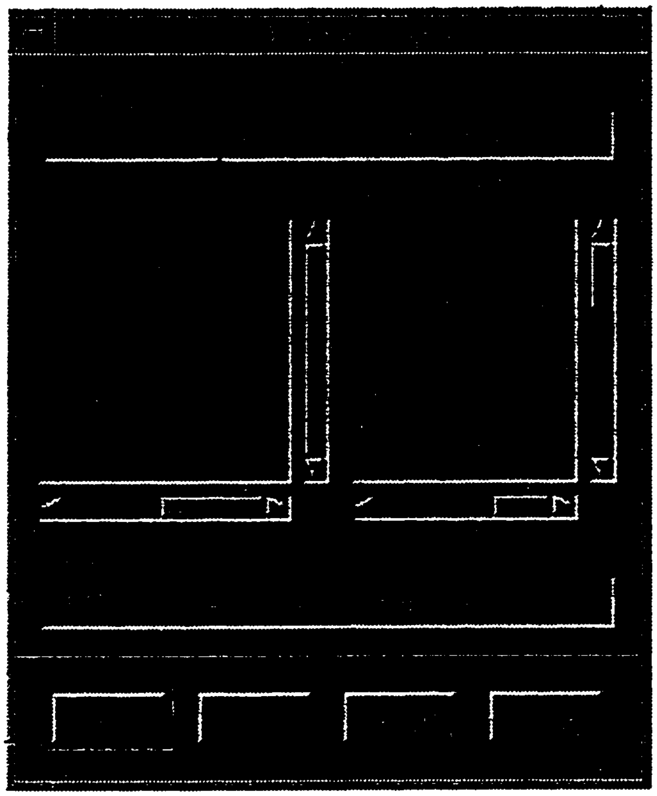


Figure 40: Layout of FileSelectionDialog Widget

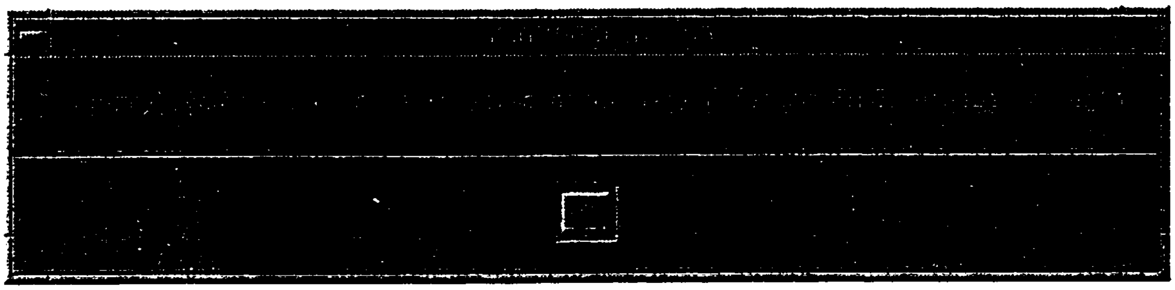


Figure 41: Layout of Reminder Popup Window

- *Help*: Figure 43 shows the layout of help window. The call-back function `help_cb()` will create a help window by calling XmCreateInformationDialog with detailed information displayed in it.

## Details of the Main Window

Figure 44 and Figure 45 shows the structure and layout of the *main_w*.

Figure 42: Layout of Warning Dialog

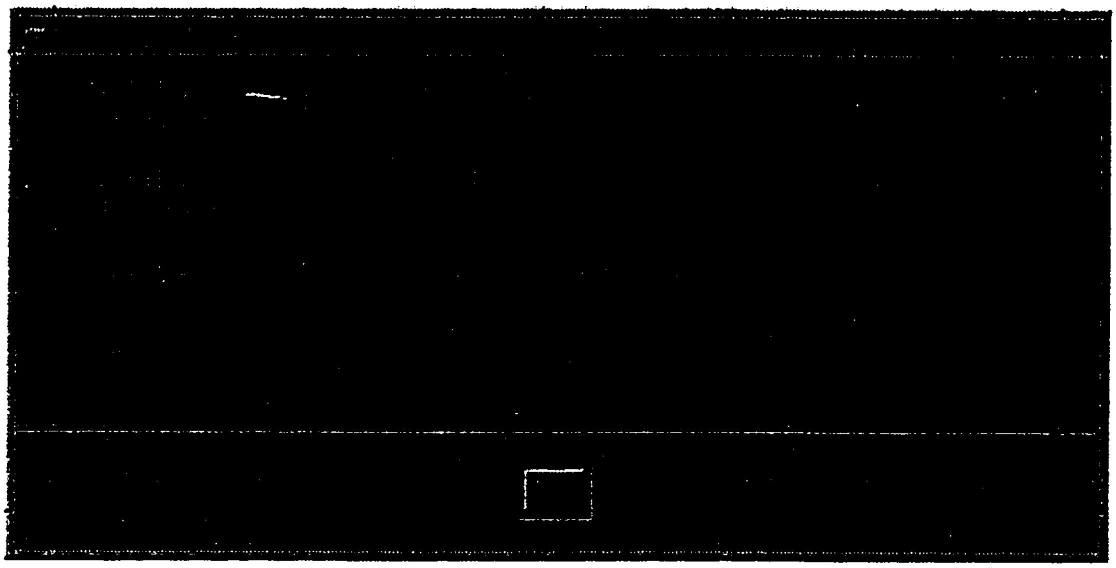


Figure 43: Layout of Help Window

- *filename_w*: This window allows user to type in filename directly. It has the exact same functionality as the sub-menu *Open* under *File* menu and the same function `read_file()` is invoked.
- *search_w*: This window enables user to search for certain string in the opened file, which is currently displayed in browser window, by typing in the search pattern.

- *message_display*: This message window displays search result messages. If a given search string is not found, "Pattern not found" will be displayed. If the search string is found, its occurrence numbers will be displayed, such as "int is found 22 occurrences".

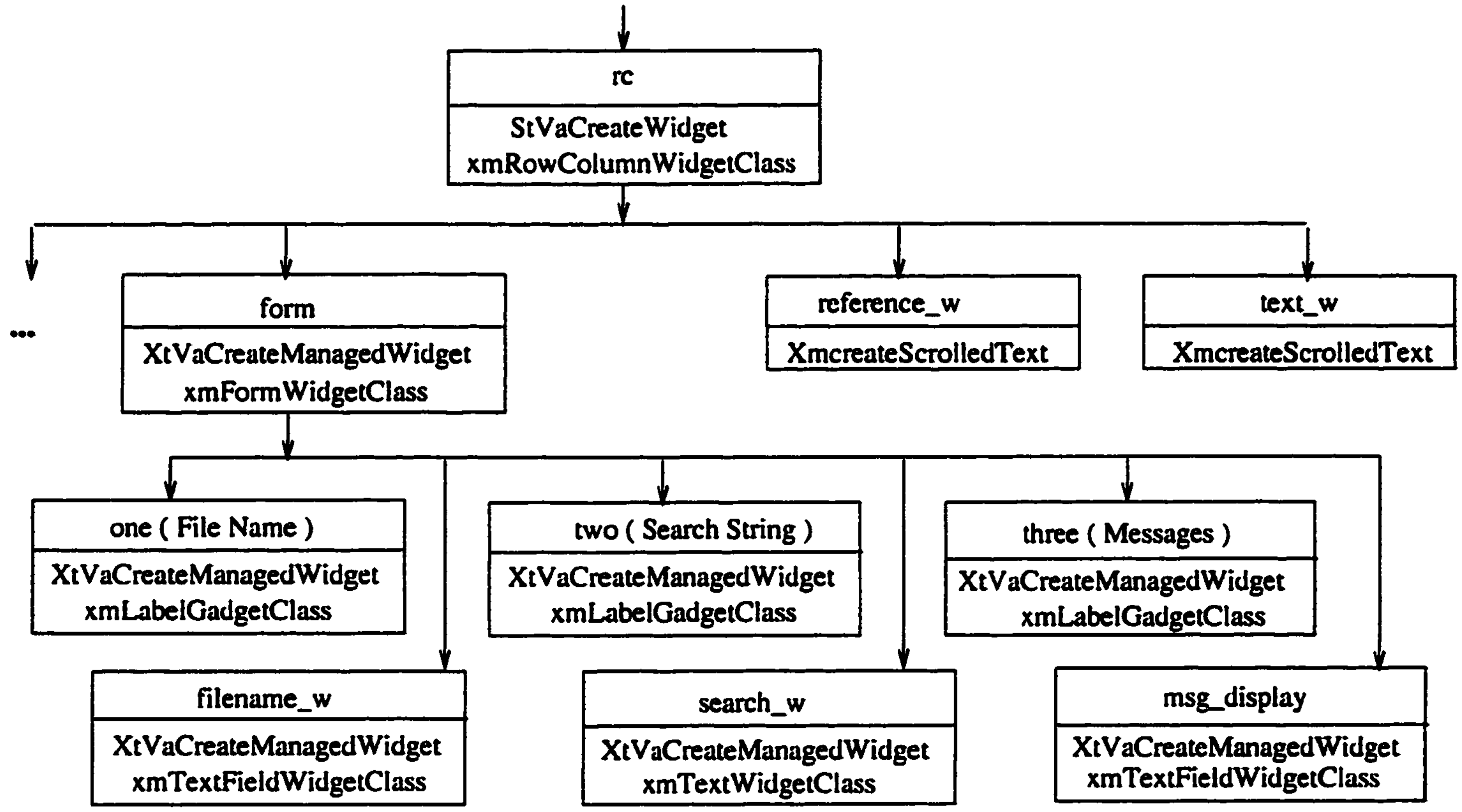


Figure 44: Structure of the Main Window

- *reference_window*: This window shows the history of user references. In Figure 45, we can see each columns in *reference_window*:
  - **Browsed** gives number of references/queries.
  - **Identifier** gives the name of the identifier.
  - **DefinitionFilename** shows the name of the definition file.
  - **Position** displays the position number of this identifier in current file.
- *text_w*: This is the core area for our tool as it is where file can be opened and displayed. We called it *working area*. In working area, a click of cursor positioned on any identifier/routine will invoke the event handler function **`show_definition()`**.

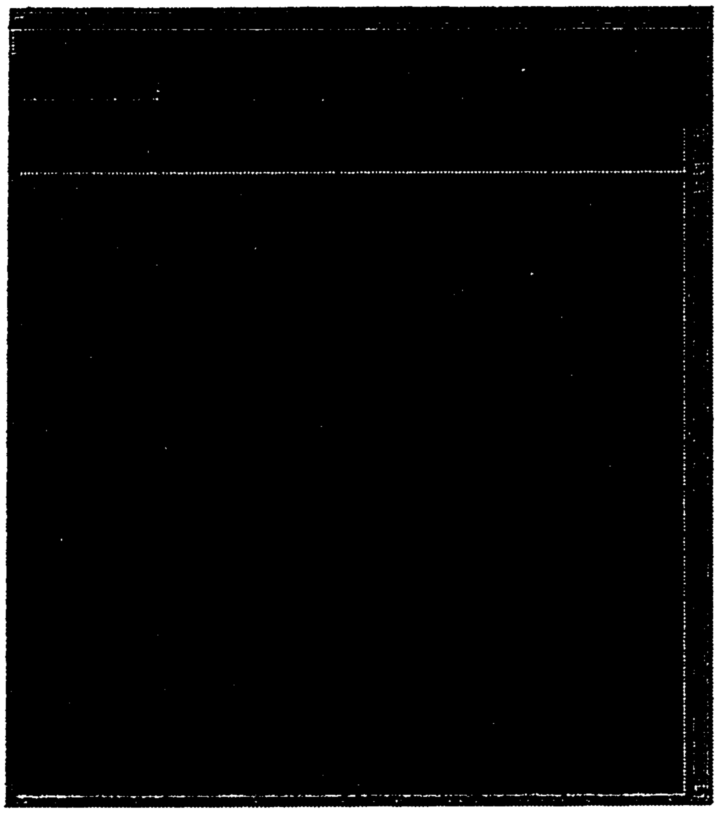


Figure 45: Main Window With Examples

### 4.5.2 Definition Window

Figure 46 and Figure 47 shows the layout and structure of the *Definition window*.

#### MenuBar

From Figure 47 we know that the menubar of *Definition window* has a similar structure to that of *browser window*. So we skip it and go through other parts of *definition window*.

#### Textbar

The *textbar* has following sub-field:

- **FileName** gives the name of current file.

- **Identifier** gives the name of the identifier.
- **Char#** shows the start position in the definition file.

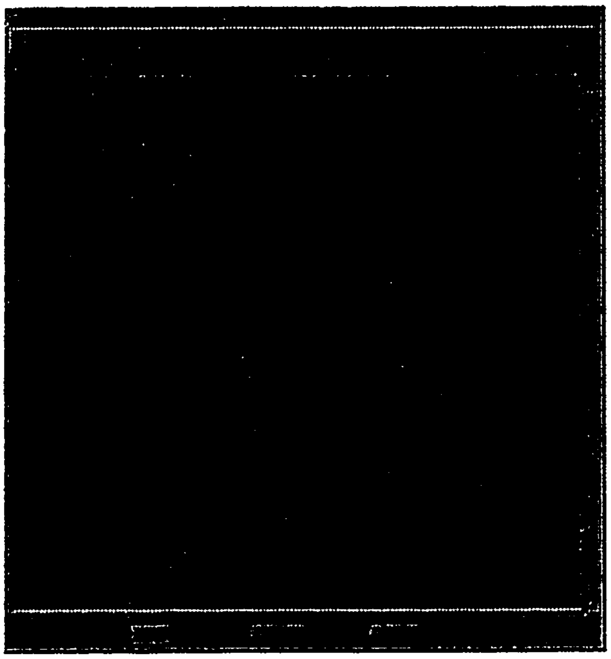


Figure 46: Layout of the Definition Window

### Text_win

This window displays definition text. When left mouse button is clicked, the event handler **show_definition()** will find out what identifier is clicked, if it is an identifier or routine it will call **create_def_w()**, which brings up definition window, and **read_definitio()**, which displays the concerned definition in *text_win*.

### Buttons

- **Next** button: call-back function **view_next()** brings the next occurrence in current file, if any, into display.
- **Prev** button: call-back function **view_prev()** brings the previous occurrence in current file, if any, into display.
- **Dismiss** button: call-back function **close_def()** closes the *definition window*.

### 4.5.3 Miscellaneous Functions in User Interface Module

`update_ref_w()`

This function updates the reference count in *referenc_w*. The counter is 1 when user first reference an identifier, increment by 1 each time the user references the identifier.

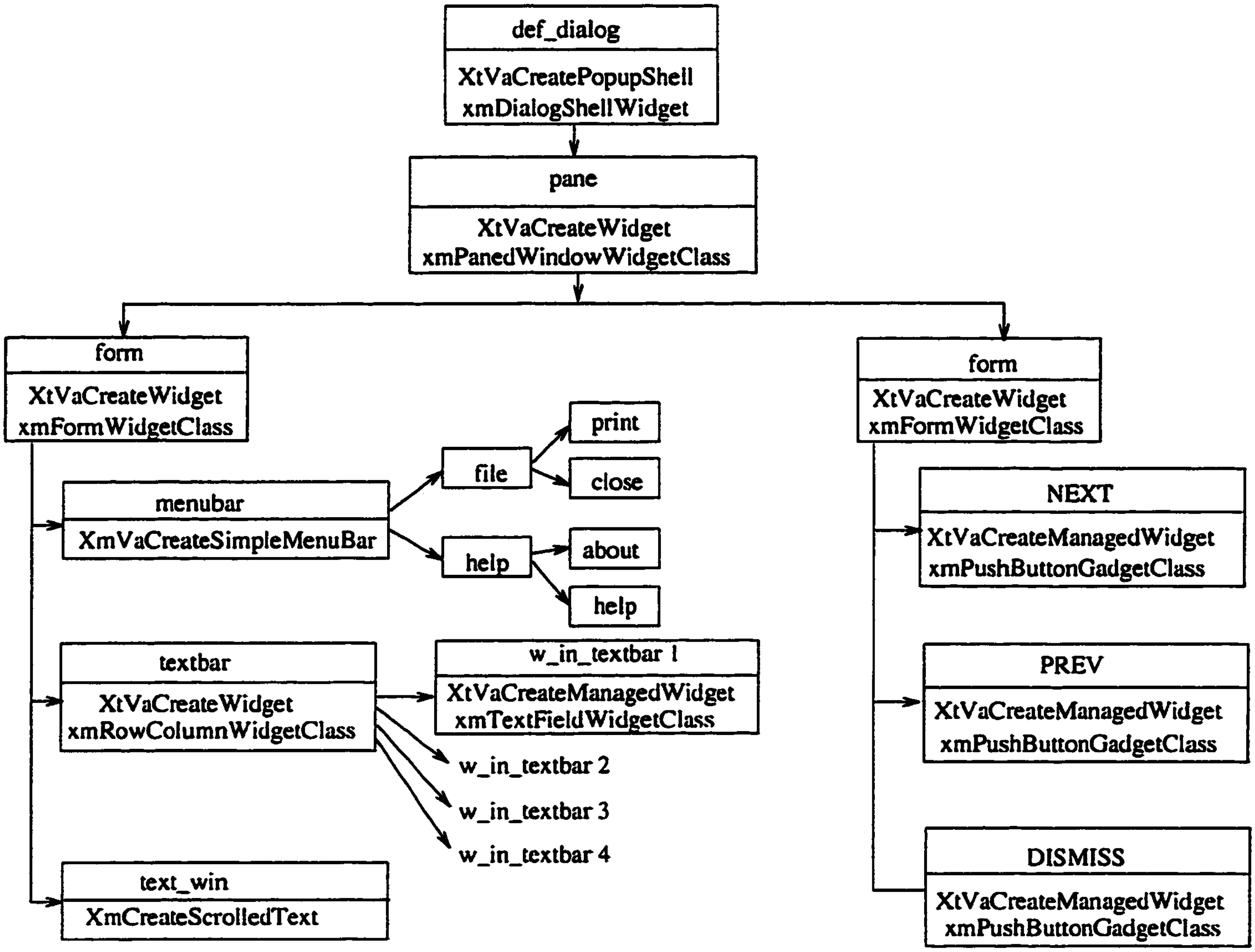


Figure 17: Structure of the Definition Window

`set_txtbar_name()`

Display given string in designated text bar widget. It is called in `create_def_w()` three times to show the right values in *textbar*.

`modify_filename()`

Remove the unnecessary characters from the full-path file name until only the file name is left. This is called in `show_definition()`.

### 4.5.4 Functions in Data Structure Module

In this module we have a group of functions to initialise and build up the skip list data structure in memory, and we also have some important interface functions between the data structure module and interface module.

`creater()` is the function that get called from the interface module. In order to create the skip lists, it calls `init_lists()` to initialise global variables and calls `build_lists()` to read each line of the dictionary file and process them by invoking `process_line()`. `process_line()` process and analyses the input string 'line' and store values to appropriate field in those skip lists.

In `process_line()`, when a node is read from the 'line', we use `search_file()` or `search_position()` to determine whether the node exists in the lists. If a new node is found, `new_id_element()`, `new_file_element()` and `new_pos_element()` are used to allocate memory for a new identifier/file/position structure and initialise its fields where necessary. `enq_id()`, `enq_file()` or `enq_pos()` will insert the new identifier/file/position structure into global skip lists.

When the user has clicked on an identifier, `show_definition()` uses the function `search_by_pos_in_file()` to find out related information. Given file name and position in this file, function `search_by_pos_in_file()` retrieves the identifier name, definition file name and definition position.

#### Debug Routines

Following are some debug routines we used during implementation:

- `print_pos_list()`
- `print_occurrence()`
- `print_file_list()`
- `print_id_list()`

- **get_prev_occ()**

- **get_next_occ()**

# Chapter 5

# Conclusion

In this thesis, we present a browsing hypertext tool for maintenance programmers. As the maintenance phase is the most costly and time-consuming phase in software system development, our tool is intended to help maintenance programmers. The tool:

- It is easy-to-use;
- It has no learning curve;
- It has reasonable performance;
- It has no language dependency.

The design and implementation of our tool are presented in the thesis. We use the concept and algorithm of skip lists by Pugh to implement the data structure of our tool. From our research we found that skip list is easy to implement and has good performance compared to other data structures.

We use the concept of hypertext to implement the interface of our tool, which enables our tool to be easy to use. But we do not change the source files like many hypertext tools do, and we do not create a set of files for this purpose either so that no extra space is needed.

Our tool uses a dictionary file to build data structure in memory. The dictionary file is created by parser and is a plain text file with certain format. So our tool is language independent as long as the dictionary file is given.

Our tool does have some limitations. Moreover, our tool can be made better by future improvement work. Some suggestions for future work are listed below:

- Parser:

  This is the biggest limitation in our tool implementation. Due to following reason, we haven't had this part completed.

  - It is easy to find identifiers in source code: if it is not a keyword or in a comment or a string and it begins with a letter, it is an identifier.
  - Finding identifiers is not enough for our tool because we must distinguish declaration and use of an identifier. In some languages, such as Pascal, it is easy to distinguish because Pascal has keyword (VAR, FUNCTION, PROCEDURE) before declarations. But it is difficult in C because there are no such obvious 'syntactic markers' in C language and even harder in C++, which has an ambiguous grammar. Therefore, it is necessary for us to find a complete C parser to extract declarations and uses of identifiers for our tool.
  - In order to find such a parser, we looked on the Internet. We found a C++ parser but it was too large, which has several hundred C++ classes, to be incorporated into our tool. We could not find a C parser that would fit comfortably into the tool. So we decided not to incorporate a parser into the tool.

  Therefore, in this thesis we present a simple example to show how our tool behaves, and the dictionary file is created manually. Unfortunately this might greatly affect the utilisation and persuasiveness of our tool for now. With a suitable parser incorporated in future version of this tool, our tool can become useful and be more appreciated.

- Procedure dependency viewer:
  Our tool provides programmers the ability to inspect each occurrence of certain routine and its definition. This only concerns details on each individual routine. On the other hand, the procedure dependency viewer can enable programmers to stand on a higher level and grasp a bigger picture of the whole system, instead of being lost in too many details.

Moreover, we could add some features, such as print the graph of function dependencies, convert between the text and graph representation of function dependencies, convert between the function call relationship and source code, etc.

- Add some new features or new functionality:
  We could add the ability to edit source files in browser window and definition windows. This requires not only a set of editing related functions to be provided, such as cut and paste, undo, save, etc, but also a proper notification to other users and an automatic update of dictionary file as well as data structures in memory. One direction to add the edit ability is to maintain two stated of the too: one is the plain text state in which source code can be edited and after modification the dictionary and data structures can be updated accordingly; the other is the hypertext state in which source code can only be inspected.

  New features like adding book mark to help user remember some special locations in files, saving reference list, and differentiate the browsed and unbrowsed occurrences in the saved reference list, discard reference history in browser window, etc.

- Become part of a software environment:
  The tool could be added to with a system interface, so that it could be incorporated with other tools to form a unified development and maintenance environment [Din94].

- Application to documentation:
  With proper parser, our tool can also be used in design document inspection. Given the dictionary file generated from a set of files, no matter they are source files or design documents or product verification documents, or all above, our tool can browse and jump between design and code or between code and test plan.

# Appendix A

# User Manual

## A.1 Introduction

A typical scenario that a user would use this tool is like this:

- Invoke HyperTool from UNIX by typing 'hypertool' and hit return
- Choose Open from the HyperTool menu to open a file
- Select a file from the FileSelection Dialog, click OK
- The selected file is displayed in HyperTool browser window (see Figure 36). User can look through the text until he finds an unclear identifier
- Click on this identifier to find out its definition
- The definition text is shown in Definition Window (see Figure 46)
- To find out an example usage of this identifier, click Next or Previous icon at the bottom of the Definition Window

## A.2 Major Services Our Tool Provides

### A.2.1 Entering HyperTool

#### Invocation

At UNIX prompt, the user types "hypertool".

**Effects**

- Two windows appear on the screen: prompt window (see Figure 38) followed by hypertext browser window.

### A.2.2 Opening File

The open command allows user to open a file in current directory.

**Invocation**

User have two methods to open a file: invoke the file selection pop-up window (see Figure 40) from the File pull-down menu and double click on a file name; or type in a file name in Filename sub-window.

**Effects**

- The selected file is displayed in the browser window.
- If the browser window has text in it already, the old text is replaced by the selected file text.

### A.2.3 Exit HyperTool

The Exit command allows user exit from the HyperTool

**Invocation**

User selects the Exit sub-menu from File pull-down menu.

**Effects**

- User will be prompted confirmation message (see Figure 42).
- If confirmed, Definition windows will be closed if exist.
- If confirmed, Browser windows will be closed.
- If cancelled, all windows remain unchanged.

### A.2.4 Hypertext File Browsing

Acts like a normal text browser. A typical scenario is provided at the beginning of this appendix.

### A.2.5 Getting Help

User can get help information.

#### Invocation

User clicks on the Help menu in the Browser Window.

#### Effects

- A Help Window (see Figure 43 ) appears. User can find relevant info in this window.

### A.2.6 Search a String

A string can be searched in the browser window and highlighted if it is found.

#### Invocation

User enters a string in the search window and return.

#### Effects

- If the string is found, it will be highlighted for each occurrence in this file
- In the message window, a message, indicating the number of occurrences of this string in this file, will be shown if the string is found; otherwise message 'No such pattern' will be displayed.

### A.2.7 Showing Definitions

Browsing through the hypertext, user is allowed to click on any identifier/function.

#### Invocation

User clicks on an identifier

**Effects**

- The browsing space stays unchanged and it's ready to accept any other operation.
- If the position clicked on is not a valid identifier - for example: user might click on '+', an addition operator or 'int' integer type, the Message sub-window should show 'No such id in dictionary'. There will be no definition window created if it didn't exist. But if definition window already exists, the text displayed in the definition window will be cleared.
- if the identifier is found in dictionary, the definition window will be created if it did not exist.
- The definition context of clicked identifier will be displayed in the definition window. Old content in definition window if any will be overwritten.

### A.2.8 Close the Definition Window

This function allows user to close the Definition Window.

**Invocation**

User selects the Close icon in Definition Window File menu or clicks on the DISMISS button at the bottom of Definition window.

**Effects**

The Definition window will be closed.

### A.2.9 Displaying Next Occurrence

This function allows user to find next occurrence of this identifier in this file

**Invocation**

User pushes the NEXT button in Definition Window.

### Effects

The next occurrence of this identifier will be displayed in Definition window. (The sequence of 'the occurrence' is defined by the dictionary: firstly definition occurrence. secondly the smallest position number. then the next smallest position number in the rest. and so on.)

### A.2.10 Displaying Previous Occurrence

This function allows user to find previous occurrence of this identifier in this file

#### Invocation

User selects the Previous button in Definition Window.

#### Effects

The previous occurrence of this identifier will be displayed in Definition window. (The sequence of 'the occurrence' is defined by the dictionary: first definition occurrence, secondly the occurrence whose position number is the closest smaller number, then the next closest smaller position number in the rest, and so on.)

# Bibliography

[And93] Anderson, A. Balanced search trees made simple. *3rd Workshop on Algorithms and Data Structures (WADS)*, 1993.

[Ber91] Berk, E. Devlin, J. *Hypertext/Hypermedia Handbook.* Intertext Publications, HcGraw-Hill Publishing Company, Inc, 1991.

[BH77] Browne, J.C. and Hohnson, David B. FAST: A Second Generation Program Analysis System. *Proceedings of Second International Conference on Software Engineering*, 1977.

[Boe76] Boehm, B.W. Software Engineering. *IEEE Transactions on Computers*, C-25(12), 1976.

[Boe81] Boehm, B.W. *Software Engineering Economics.* Prentice-Hall, 1981.

[Bra92] Brain, M. *Motif Programming, The Essentials ... and More.* Digital Press, 1992.

[Bus45] Bush, V. As We May Think. *The Atlantic Monthly*, 176(1):101–108, July 1945.

[Bux80] Buxton, J. DoD Requirments for ADA Programming Support Environments. *STONEMAN. DoD High Order Language Working Group*, February 1980.

[Bux84] Buxton, J. *Rationale for Stoneman. Interactive Programming Environments.* Mcgraw-Hill, 1984.

[CNR90] Chen, Yih-Farn, Nishimoto, Michael Y., and Ramamoorthy, C.V. The C information Abstraction System. *IEEE Trans. On Software Engineering*, March 1990.

[Col92] Collins, W.J. *Data Structures, an Object-Oriented Approach.* Addison-Wesley Publishing Company, 1992.

[CR88] Chikofsdy, E. J. and Rubenstein, B. L. CASE: Reliability Engineering for Information Systerm. *IEEE Software*, pages 11–16, March 1988.

[CR92] Cybulski, Jacob L. and Reed, Karl. A Hypertext based software Engineering Environment. *IEEE Software*, March 1992.

[DEFH87] Dart, S.A., Ellison, R.J., and Feiller, P.H.and Habermann, A.N. Software development environments. *IEEE Computer*, 20(11), 1987.

[Del92] Deller, D. *Motif Programming Manual, Volume Six.* O'Reilly Associates, Inc., 1992.

[Din94] Ding, Hanwei. A Design Tool for Object-Oriented Development. Master's thesis, Concordia University, 1994.

[DL97] Delix, A. and LeViet, Q. Contemporary Access Structures under Mixed Workloads. *The Computer Journal*, 40(4), 1997.

[DLK90] Damaskinos, V.R.N., Linos, P., and Khorshid, W. VIFOR: A Tool for Software Maintenance. *Software - practice and experience*, 20, January 1990.

[Dow87] Dowson, Mark. Integrated project support with istar. *IEEE Software*, November 1987.

[Fer88] Fernstrom, C. ESF Desgin Guidelines and Architecture. *ESF Consortia*, 1988.

[FH93] Flecher, T. and Hunt, J. *Software Engineering and CASE. Bridging the Culture Gap.* McGraw-Hill, Inc, 1993.

[Fis86] Fischer, G. From interactive to intelligent systems, in Software System Design Methods. *NATO ASSI Series F: Computer and Systems Sciences*, 22(Springer):185 212, 1986.

[FO89] Fernstrom, C. and Ohlsson, L. The ESF vision of Software Factory. *Proceddings of the International Conference on System Development Environments and Factories*, May 1989.

[GB91] Gonnet, G.H. and Baeza-Yates, R. *Handbook of Algorithms and Data Structures in Pascal and C.* Addison-Wesley Publishing Company, 1991.

[GS90] Garg, Pankaj K. and Scachi, Walt. A hypertext system to manage software life-cycle documents. *IEEE software*, May 1990.

[HS94] Halasz and Shwartz. The Dexter Hypertext Reference. *Communications of the ACM*, 37(2), Feburary 1994.

[JKCG94] Jambor-Sadeghi, Kamyar, Ketabchi, M.A., Chue, Junjie, and Ghiassi, M. A systematic Approach to corrective Maintenance. *The computer Journal*, 37(9), 1994.

[KP95] Kelley, A. and Pohl, I. *A book on C.* The Benjamin Cummings Publishing Company, Inc, 1995.

[LB85] Lehman, M.M. and Belady, L.A. Program evolution. *Academic Press. APIC Studies in Data Processing (27)*, 1985.

[Leh80] Lehman, M.M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), 1980.

[LF93] Layzell, P. J. and Freeman, M. J. DOCKET: A CASE Tool and Method to Support Software system Understanding and Modification. *IEEE*, 1993.

[Lin84] Linton, M.A. Implementing Relational Views of Programs. *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development Environment*, May 1984.

[LS80] Lientz, B.P. and Swanson, E.B. *Software Maintenance Management.* Addison-Wesley, 1980.

[McC89] McClure, C. *CASE is Software Automation.* Prentice-Hil, 1989.

[MHB90] McKenzie, B.J., Harries, R., and Bell, T. Selecting a Hashing Algorithm. *Software - practice and experience*, 20(2), Febuary 1990.

[Mor88] Morgan. Configuration Management and Version Control in the Rationnal Programming Environment. *Proceedings of the Ada-Europe Conference*, June 1988.

[MV93] Mayrhauser, A. von and Vans, A. M. . From Code Understanding Needs to Reverse Engineering Tool Capabilities. *IEEE*, pages 230–239, 1993.

[NN89] Norman, R.J. and Nunamaker, J.F., Jr. CASE productivity perceptions of software engineering professionals. *Communications of the ACM*, 32(9), 1989.

[PSK96] Paakki, Jukka, Salminen, A., and Koskinen, J. Automated Hypertext Support for Software Maintenance. *The computer Journal*, 39(7), 1996.

[Pug90] Pugh, W. Skip lists: a probabilistic alternative to balanced trees. *Commnun. ACM,*, 33:668–676, 1990.

[RDM+87] Rowe, L.A., Davis, M., Meesinger, E. Meyer, C., Spirakis, C., and Tuan, A. A browser for Directed Graphs. *Software - Practice and Experience*, 17(1), January 1987.

[Rip88] Ripken, K. Automated Support for Design and Documentation of Large Ada Systems. *Milcomp'88 Conference*, September 1988.

[Sch89] Schefstrom, D. Building a Highly Integrated Development Environment Using Preexisting Parts. *IFIP '89*, September 1989.

[Sch95] Schildt, Herbert. *C++: The Complete Reference*. Osborne McGraw-Hil, 1995.

[She97] Sheu, P. C.-Y. *Software Engineering and Environment*. Plenum Press, 1997.

[Shn92] Shneiderman, B. *Designing the User Interface, Strategies for Effective Human-Computer Interaction*. Addison-Wesley Publishing Company, 1992.

[SSW86] Shneiderman, B., Shafer, P., and Weldon, L. Display Strategies for Program Browsing: Concepts and Experiment. *IEEE Software*, May 1986.

[Ste85] Steffen, J.L. Interactive examination of a C program with Cscope. Proceeding of USENIX Assoc. Winter Conf., January 1985.

[SV93] Schefstrom, D. and Van den Broek, G. *Tool intergration: Environnets and Frameworks*. Wiley prefessional computing, 1993.

[Thi90] Thimbleby, H. *User Interface Design.* ACM Press, 1990.

[TM81] Teitelman, W. and Masinter, L. The Interlisp programming environment. *Computer*, 14 (4), 1981.

[TR81] Teitelbaum, T. and Reps, T. The Cornell Program Synthesizer, a syntax-directed programming environment. *Communicaations of the ACM*, 24(9):563–573, 1981.

[Vli93] Vliet, Hans Van. *Software Engineering.* Principles and Practice, 1993.