A DESIGN TOOL FOR OBJECT-ORIENTED DEVELOPMENT

Hanwei Ding

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

November 1994

ISBN 0-612-01363-4

Abstract

A Design Tool For Object-Oriented Development

Hanwei Ding

This thesis presents a supportive tool for object-oriented design and system evolution. As the design is the most important and effort consuming phase in object-oriented software development, our tool aims at providing automated support to empower the designers as much as possible during the design process. This is done by providing the designers with multiple views of the design, the facility to update the design easily, the ability to check the completeness and consistency of the design, and the ability to generate a printed report of the design. Maintenance is the most costly phase in system evolution; our tool provides directly assistances to the maintainers during system evolution by supporting automation in capturing and altering the design.

A carefully defined, consistently formatted user interface is developed with Motif as part of the tool, and as the medium for human interaction with the function modules of the tool. We considered the interface part to be the most significant and necessary part of the tool design, and emphasis is put on it to achieve the ease of use of the tool.

Some directions of further work about the tool, which will brighten the future of the tool, are also described.

Acknowledgements

I would like to express my sincere 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 would like to thank my husband Boqian Cheng for his encouragement and moral support.

Finally, I would like to dedicate this work to my parents who always stood behind me with great patience and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Why object-oriented design?

One major trend in the history of software engineering is the shift in focus from programming-in-the-small to programming-in-the-large. This trend changed the software engineer's way of thinking and eventually led to a new paradigm: object-oriented technology.

In the late 80's, almost every software developer experienced this "object-oriented" storm which touched every field of software engineering. Object-oriented technology provides a different approach to the software system. It relies on the three developing phases – analysis, design and implementation to carry out the major characteristics of object-orientation: data abstraction, information hiding, dynamic binding and inheritance.

Conventional functional approaches require the designer to first ask what the system does: the functional aspects. This is probably adequate if the target system solves a fixed problem once and for all. But changes happen over time. What the system will do in its first release is probably going to be a little different from what it was expected to do at requirement time, and very different from it will be expected to do several years later.

However, the categories of objects on which the system acts will probably be quite stable. A document processing system will always work on documents, chapters, sections paragraphs, and so on. Thus it is wiser in the long term to rely on categories of objects as a basis for decomposition than on functionality. The object-oriented design technique is defined in such a way that the principle of modular decomposition of the system is according to the classes of objects which the system manipulates[19].

Compared to the traditional functional design, object-oriented design aims for more robust software that can be easily reused, modified, maintained and extended. The greatest strength of this approach to development is that it offers a mechanism that captures a model of the real world which leads to greatly improved understandability and maintainability of systems.

But current approaches to object-orientation lack complete and standardized representations, and lack a complexity management mechanism to permit viewing the design at varying levels of details or from various perspectives. Some weaknesses of current object-oriented analysis and design lie in: identifying interfaces, application and system classes; identifying and representing different kinds of relationships; maintaining consistent and correct semantics for such relationships; representing dynamic views, and maintaining consistent levels of abstraction[20][1].

Although object-oriented design is criticized by different groups, no one has said that object-oriented design should be abandoned. One way to make it do the job better is to provide better techniques and better supporting tools.

1.2 Why is a tool needed?

When large systems are developed, all information should be recorded properly. A good tool will help to automate a large amount of such work, and provides efficient support to the life cycle of software development. Developing a large system is difficult and expensive, and maintaining it is even more difficult. As the existing knowledge changes over time, more detailed knowledge is required and more information will be added or existing information will be removed during the evolution of change. Tools empower the designer, freeing him or her to concentrate upon the truly creatively aspects of the designing and maintaining.

The object-oriented programming revolution has spawned a whole new industry devoted to object-oriented analysis and design tools and the corresponding methodologies they support. As in conventional structural design, there are different tools to support each phase of the software life cycle, i.e., analysis, design and implementation. But current tools are intolerant of changes[16]. When a client has used a system and changed the requirements after certain time, the maintenance programmers will respond to change the code to meet the new requirements. But in most cases, they forget to update the design. After evolving like this for a long time, this system will not be recognizable from the design, and the inconsistency between requirements, design and implementation will make the system extremely difficult to change and maintain.

To our knowledge, there are no convenient tools to help to capture the change of design during system evolution. No facilities are provided for integrating design changes with existing partial implementations. Since reuse is a fundamental aspects of object-oriented technology, better tools are needed to support re-analysis, re-design and re-implementation. Particularly, a good tool which is convenient to use and efficiently supportive for updating the design during system evolution is needed, and this latter issue is made the focus of this thesis.

The design tool described in this thesis is a software system which provides facilities for creating, examining, checking and modifying design. The tool provides a user friendly interface. It allows the user to display the design in different forms, as text or table. The user can select the level of detail to view the design when it is displayed in tabular form. The tool also allows the user to check the consistency and completeness of design, and it can generate high-quality printable output of the design.

Such a design helps to fill the process information gap and re-orient existing design to support software evolution more effectively. The designer and maintainer can simply use it as required. It supports system collection and maintains and coordinates design information.

It is clear that the designer's work load will be greatly reduced with the help of such a tool during software creation and maintenance.

1.3 The structure of the thesis

The rest of the thesis is organized as follows: Chapter 2 introduces a survey of background in object-oriented design. Various methods for object-oriented analysis and design and some supporting tools are overviewed; Chapter 3 outlines the issues that were considered during the design of the tool. The goal and objectives, design format, representation forms of the design and design decisions made for the tool are exploited and illustrated; Chapter 4 discusses the design and implementation of the Design Tool. The central data structure, the abstract syntax tree, the parser module and the user interface and major functional modules are presented. Chapter 5 describes a walk-through of the Design Tool with an example given in previous chapter. Some windows with displayed tables of a design which are constructed by the tool are described, and an example of a printable report of a design which is generated by the tool is also given. Chapter 6 gives conclusion of the study and suggests for further related works.

Chapter 2

Principles of Objected Oriented Design

In this chapter, we review various methods that have been proposed for object oriented analysis and design. These methods formed the basis for the approach to design which underlies our design tool. Some of the tools that have been developed to support object oriented program development are described in the second section.

2.1 Object-Oriented Analysis and Design

Although object oriented programming was introduced in the late sixties [21], the first references to object oriented *technology* appeared in the early eighties. In this section, we introduce some of the terminology that has evolved around object oriented technology, and we discuss some of the design methodologies that have been proposed.

2.1.1 General Concepts

Object-oriented technology, since it was proposed, has experienced more than a decade's practice, and has steadily gained ground in the world of software development. Although the term *object-oriented* has become a buzzword that means different things to different people: it has been used synonymously with *modularity, information hiding, encapsulation* and *data abstraction*, but it always refers to a system design philosophy which is fundamentally different from the functional design.

System development is the process of producing descriptions of models. The development of a large complex system involves steps of decomposition of the system into desired model descriptions. The functional approach emphasizes the functional aspect of the system, which concerns what the system does. Therefore the decomposition of a system with a functional design is based on the model which forms the functionality of the system. This top-down functional approach is good if the target system solves a fixed problem once and for all. But such case does not happen regularly, the functional properties of a system do change over time. In a long view, what the system is to do in its first release is probably different from what it is expected to do at the requirement time, and very different from what it is expected to do a long time later. The easily changeable property of the functional aspect of a system limits the maintainability of the system. However, the categories of objects on which the system acts are quite stable. An operating system will always work on devices, memories, processing units, and so on. Therefore it is wiser in the long term to rely on the categories of objects as the basis for system decomposition. The object-oriented design technique is defined in such a way that the principle of modular decomposition of the system is based on the classes of objects which the system manipulates. This approach reveals what the system is, not what it does. The objects of a system, which are the building blocks in the system development, are therefore made central issue of the object-oriented design.

But what is an **object**? An object usually reflects the entity in the real world, like a car, a tree, a computer, etc. "An **object** is an entity able to save a state (*information*) and which offers a number of operations (*behavior*) to either examine or affect this state" [15]. In other words, an object encapsulates both functions and data which are conceptually related to each other. Encapsulation therefore is considered as one of the important characteristic of the object, while information hiding, introduced by Parnas[22], is another characteristic that is considered as central to the nature of the object.

Other concepts[25] of object-orientation are listed as follows:

Abstract data type — An abstraction that describes a set of objects in terms of an encapsulated or hidden data structure and operations on that structure.

Class — An abstraction of a set of objects that specifies the common static and behavioral characteristics of the objects, including the public and private nature of the state and behavior.

Generic definition — The ability to parameterize a class, like the *template* concept in C++.

Inheritance — (a). *Single*: A relationship between classes whereby one class acquires the structure of other classes in a strict hierarchy from a single parent.

(b). *Multiple*: A relationship between classes whereby one class acquires the structure of other classes in a lattice with multiple parents.

Instance — One object in the set of objects described by a class abstraction.

Instantiation — The creation of a new instance object of a class or a new specific class from a generic class.

Message — A request for an object to carry out the sequence of action in one of the operations of its class.

Method — An operation defined in a class abstraction that carries out a sequence of actions in the class.

Operation — A class-level abstraction describing a sequence of messages or actions that access or change the state of an instance of the class.

Overloading — A simple form of *polymorphism*; the ability to attach more than one meaning to the same name in the same scope as differentiated by type or some other class characteristic.

Polymorphism The ability of an entity to refer at runtime to instances of various class. Hence, the actual operation performed on receipt of a message depends on the class of the instance.

Currently there are many methods proposed by different developers for object-oriented development. The following section gives an overview of some of the methods that we have referred to as the state-of-the-art object-oriented technology.

2.1.2 Overview of Object-Oriented Design Methods

Booch's design method[6][5] is built upon the object model. The concept of an **object** is introduced that has *state, behavior*, and *identity*, and with two kinds of hierarchy relationships: *using* and *containing*. A **class** is defined as a set of objects that share common structure and a common behavior which has four kinds of hierarchy relationships: *inheritance, using, instantiation* and *metaclass*. The key abstractions are the classes and objects that form the vocabulary of the problem domain, and classification is applied in the process of abstraction.

A set of notations are defined for the design which include four basic diagrams (*class*, *object*, *module*, and *process*) and two supplementary diagrams (*state transition* and *timing*). Each of these notations is defined as follows:

- A *class* diagram is used to show the existence of classes and their relationships in the logical design of a system; a class diagram represents all or part of the class structure of a system.

- An *object* diagram is used to show the existence of objects and their relationships in the logical design of a system; an object diagram represents all or part of the object structure of a system and primarily illustrates the semantics of key mechanisms in the logical design. A single object diagram represents a snapshot in time of an otherwise transitory event or configuration of objects.

- A *module* diagram is used to show the allocation of classes and objects to modules in the physical design of a system; a module diagram represents all or part of the module architecture of a system.

- A *process* diagram is used to show the allocation of processes to processors in the physical design of a system; a process diagram represents all or part of the process architecture of a system.

- A *state transition* diagram is used to show the state space of an instance of a given class, the events that cause a transition from one state to another, and the actions that result from a state change.

- A *timing* diagram is used to show the dynamic interactions among various objects in an object diagram.

In addition to the above diagrams, a textual representation notation is also defined. This so-called *template* notation captures all the important aspects of a class which is less readable but more detailed than the diagram. During the design process, the diagrams and templates evolve as new design decisions are made and more detail is established.

Unlike structured design, the process of the object-oriented design is neither top down nor bottom-up; rather it can be best described as round-trip gestalt design, which emphasizes the incremental and iterative development of a system. A process of the object-oriented design is described by Booch as follows:

- The first step in the process of the design involves the identification of the classes and objects at a given level of abstractions and the invention of important mechanisms.

- The second step involves the identification of the semantics of these classes and objects; the important activity in this step is for the developer to act as a detached outsider, viewing each class from the perspective of its interface.

- The third step involves the identification of the relationships among these classes and objects; in this step, the ways in which things interact within the system are established, with regard to the static as well as the dynamic semantics of the key abstractions and important mechanisms.

- The fourth step involves the implementation of these classes and objects; the important activities in this step involve choosing a representation for each class and object, and allocating classes and objects to modules, and programs to processes; this step is not necessarily the last step, for its completion usually requires that we repeat the entire process, but at a lower level of abstraction.

The method proposed by Wirfs-Brock *et al.*[27], has been given the name **responsibility-driven** design. As usual, a set of basic concepts of the design are defined. In additional to the concepts described previously, such as *object, class, inheritance*, and *polymorphism*, the following concepts are defined:

Object accessing -- one object accesses another object by sending it a *message*, which consists of the name of an operation and any required arguments. A *method* is the step-by-step algorithm executed in response to the method received, and whose formal specification is called *signature.*

Contract -- the ways in which a given client can interact with a given server, i.e. the list of requests that a client can make of a server.

responsibility — responsibilities include two key items: the *knowledge* an object maintains, and the *actions* an object can perform. They are all the services an object provides for all the contracts it supports.

Collaborations — collaborations represent requests from a client to a server in fulfillment of a client responsibility. Therefore an object can fulfill a particular responsibility itself, or it may require the assistance of other objects.

Subsystems — groups of classes, or groups of classes and other subsystems, that collaborate among themselves to support a set of contracts. And a class is part of a subsystem only if it exists solely to fulfill the goals of that subsystem.

Protocol — a set of signatures to which a class will respond.

The process of design involves six activates, which are described as follows:

- Finding classes by carefully and repeatedly examining the requirements specification, using abstract and modeling as the helping tools to find the candidate classes, and identifying candidate superclasses by grouping classes that share common attributes.

- Assigning responsibilities by determining the responsibilities of the system as a whole, and then assigning each of these responsibilities to a specific class; looking for relationships between classes to find additional responsibilities.

- Identifying collaborations between classes by analyzing the interactions of each class and then examining the responsibilities for dependencies: if a class is responsible for a specific action, but does not possess all the knowledge needed to accomplish that action, a collaboration is identified between the class and another class(es) that possess the knowledge; identifying additional collaborations by looking for the relationships between classes.

- Constructing Hierarchies examining the relationships between class and identifying the contracts of the classes using the following guidelines:

  ▷ model the "kind-of" hierarchy.

  ▷ factor common responsibilities as high as possible.

  ▷ ensure that no abstract class inherits from concrete classes.

  ▷ eliminate classes that do not add functionality.

- Identifying subsystems by examining the responsibilities and collaborations of the classes. Strongly interdependency between two classes is the major reason to group the two classes into one subsystem.

- Creating protocols of a class by refining the responsibilities of the class and providing the formal specification of the class interfaces.

The tools used in the design process are *class and subsystem cards*, *hierarchy graph*, *Venn diagram, collaboration graph* and *walk-through*. The result of the design process is a design document which includes: a graph of the class hierarchies, a graph of the paths of collaboration for each subsystem, a specification of each class, a specification of each subsystem and a specification of the contracts supported by each class and subsystem.

The Object Modeling Technique (OMT) presented by Rumbaugh *et al.*[23] is a method for collecting and representing information about requirement and design. The OMT method introduces three kinds of models to describe the system. They are *object* model, *dynamic* model and *functional* model. Each model is applicable during all stages of the development and acquires implementation details as development progresses. The three models and their representation notations are described as follows:

Object model: the purpose of object modeling is to describe the objects in a system. The object model describes the structure of the objects—their identity, their relationships to other objects, their attributes, and the operations they preserve. This model provides the essential framework into which the dynamic and functional models can be placed, since the objects are the units into which the world is divided, and the molecules of the whole system.

The object model is represented graphically with object diagrams containing object associated with other classes. There are two types of object diagrams: *class* diagrams and *instance* diagrams. A class diagram is a schema for describing object classes (a group of objects with similar properties (*attributes*) and common behavior (*operations*)). The class name, its attributes and operations are displayed in the class diagram. Also clear emphasis is placed on the associations between objects. One-to-one and one-to-many relationships can be represented in the diagram. An instance diagram describes how a particular set of objects relate to each other. It is used mainly to show examples of a class diagram to help to clarify the complexity of the class diagram.

Dynamic model: the changes over time to the objects and their relationships of a system are represented by the dynamic model. The dynamic model of a system describes the flow of control, the interactions, and the sequencing of operations in the system of concurrently-active objects. The major dynamic modeling concepts are *events*, which represent external stimuli, and *states*, which represent values of the objects.

An event is a one-way transmission of information from one object to another, and can be grouped into an *event class* and given a name to indicate the common structure and behavior. The time at which an event occurs is an implicit attribute of all events. A *state* is an abstraction of the attribute values and links of an object. Sets of values are grouped together into a state according to the properties that affect the gross behavior of the object. The state diagram is the graphical representation of finite state machines, and is a notation familiar to most software developers.

Functional model: the functional model describes the aspects of a system concerned with transformations of values—functions, mappings, constraints, and functional dependencies. It captures what a system does, without regard for how or when it is done.

The functional model is represented with data flow diagrams. The data flow diagrams show the dependencies between values and the computation of output values from input values and functions without regard for when or if the functions are executed.

The output from *analysis* is a specification of three models that captures the three aspects of the system: the objects and their relationships, the dynamic flow of control, and the functional transformation of data subject to constraints. The overall architecture of the system is determined during the *system design* where the system is organized into subsystem, concurrency is organized by grouping objects into con current tasks, and overall decisions are made about interprocess communication, data storage, and implementation of the dynamic model. The full definitions of the classes and associations, as well as the interface and algorithms of the methods used to implement the operations are determined in the *object design* phase. This phase adds internal objects for implementation and optimizes data structures and algorithms.

Jacobson's method[15] is given the name **use case driven** approach. The term

use case was introduced by Jacobson; it plays important role in the modeling of the dynamic behavior of a system. Usually, the entire behavior of a system is described by describing the behavior model by model in the system. Jacobson's method describes a system as a black box by describing a number of aspects of the system with each of these aspects corresponding to a behaviorally related sequence called *use cases.* The *use case model* consists of two items: the *actor* and the *use case.* The actors model anything that needs information exchange with the system, thus define the roles that the users of the system can play. The actors are the major tool for finding the use cases. A use case is a specific way of using the system by using some part of the functionality. Each use case constitutes a complete course of events initiated by an actor and it specifies the interaction that takes place between an actor and the system. The use case model forms a thread running through all the phases of the system development.

The system development involve two separate but interacting processes: system **analysis** and system **construction**. The analysis process aims at defining and specifying the system to be built and no requirements from the actual implementation environment are to be taken into account. The analysis process produces two models: the **requirement** model which specifies all the functionality that the system should be able to perform through the use cases in the use case model; and the **analysis** model which forms the basis for the system's structure and specifies all the logical objects to be included in the system and how they are related and grouped. The requirement model uses actors and use cases to describe in detail each and every way of using the system, from a user's perspective. The analysis model aims at forming a logical and maintainable structure in the system. Three object types are used to construct the analysis model: *interface* objects, *entity* objects and *control* objects. The interface objects model all the functionality that concerns the system interfaces; the entity objects model all functionality that handles the actual information kept in the system for a long period of time and the control objects model such functionality that is not naturally tied to any of other objects (often mainly behavior). These object types are identified when the use cases are analyzed and broken down. Subsystems which are formed by grouping the objects are used to structure the system in large units.

The construction process consists of two phases: **design** and **implementation**. The results of these two phases are two models called *design model* and *implementation model* respectively. Three steps are involved in the process of producing the design model: initially, the *actual implementation environment* is identified, such as how the processes should be handled, the constraints from the programming language, etc. This step aims at drawing conclusion on how these circumstance should be handled in the system; secondly, the conclusion drawn in the first step is incorporated into the design to develop a *first approach to a design model*. The analysis model is used as a base and is translated into design objects in the design model that fits the current implementation environment; finally, the object interfaces are constructed by describing how the objects interact in each specific use case. Therefore the design model is formalized to describe all stimuli sent between objects and to define what each operation will do to each object. From the design model, every detailed specification of all objects, including their operations and attributes are obtained. The implementation activity then implements each specific object with the selected programming language.

Champeaux and Lea *et al.*[11] state that "Object-oriented design is best characterized as a transformational process". The transformational process starts with a declarative, non-computational specification, and then applies methods and strategies that result in an implementable software design. The process is divided into three major phases: *class* design, *system* design and *program* design. The class design defines the representational and algorithmic properties of the classes obeying the declarative constraints specified with the object-oriented analysis; the system design maps objects to processors, processes, storage, and communication channels; the program design reconciles the functionality and resource mapping in order to meet the the performance requirements when expressed using the target implementation languages, tools, configurations, etc.

Buhr and Casselman[7] present a new design concept for systems with distributed control, called **causality flow**, a new notation for this control, called **timethreads**, and a design process based on the notation called "designing with timethreads".

"Causality flow" means the chains of causes and effects rippling through a system as a whole that triggered by the occurrence of stimuli from its environment. "Timethreads" is a visual notation for describing causality-flow scenarios. This notation provides high level abstraction which can be used as a reasoning tool to help drive the requirements-to-preliminary-design process, not just as a means of recording scenarios for a more-or-less completed design. For the purposes of "designing with timethreads", a notation *role architecture* is introduced to express the system organization at a high level of abstraction. Timethreads and role architectures have no built-in assumptions about software or other implementation techniques. In addition, Timethreads are self-similar at different levels of details, which means that they use the same notation and may show similar patterns. The details at different levels will be different, but there is nothing in the general "look" of timethreads at different levels to identify the level. This feature enables the timethreads to span a wide design range from requirements definition to detailed design.

Coleman *et al.*[9] introduce a notation called **Objectcharts** for specifying object classes. An objectchart diagram is an extended form of a Statechart. The Objectchart transitions correspond to the state-changing methods that the class provides as well as those that the class requires of other classes. Object attributes and observer methods annotate the Objectchart states.

Haythorn[13] indicates that it "is probably a marketing mistake" that "discussions of the benefits of object-oriented programming often emphasize reuse rather than extensible, maintainable systems". He argues that reuse applies to individual classes and pays off several years down the road, while maintainability applies to the whole system and pays off sooner. Since we are really concerned with building systems, maintainability is a more important goal than reuse, and clear guidelines for system design which focuses on the maintainability should be obtained.

Monarchi and Puhr[20] evaluate current research on object-oriented analysis and design (OOAD). Various OOAD techniques (i.e. processes or methods) and representations are compared, and some strong and weak areas in OOAD are identified. They state that the strengths of current OOAD research lie in: identifying semantic classes, attributes and behavior of a class; placing the methods to a class; identifying and representing generalization and aggregation structures; and representing static views of the system and classes (i.e. structures). Some weaknesses in current OOAD research identified by [20] are: the way to identify interface, application and system classes; the way to determine when an attribute, relationship or behavior should be a class; where to place the classes; the way to identify and represent other kinds of relationships than the inheritance and part-of; the way to maintain consistent and correct semantics for relationships; the way to represent dynamic views (i.e. message passing, control, etc.); the way to integrate static and dynamic models; and the way to maintain consistent levels of abstraction/granularity.

2.2 Supporting Tools for Object-Oriented Design

A lot of effort has been put on the development of the supporting tools for object-oriented development since the OO technology was proposed. Beck and Cunningham[4] have introduced **index cards** which are used as a simple tool for teaching object-oriented concepts to designers. This index card has been used by the responsibility-driven designers to capture initial classes, responsibilities, and collaborations, as well as recording inheritance relationships and common responsibilities defined by superclasses[26]. Index cards are helpful because they are compact, easy to manipulate, and easy to modify and discard with small amount of classes. They can be easily arranged on a tabletop and a reasonable number of them can be viewed at same time. However, with large, complex systems, such tools for recording design information will be clumsy and inefficient. Especially during the system evolution, updating the design will become a heavy burden to the maintainers.

Some developers have suggested the idea of **design record** for reducing software lifecycle costs[2]. A design record is a collection of information, often in an on-line repository, to support software evolution, which also provides a view of information (i.e., interactive screen of information) about the system or its evolution. By collecting a great deal information about the software design, much information is available and helpful to the software developers and maintainers. Capturing designs or design decisions was one of the primary motivators for the design record since the design decisions made throughout the life of a system are critical in defining the architecture of the system, and they provides clues to the maintenance programmers of the "*why*" of existing code. A design record helps to fill the process information gap and re-orient existing information to support software evolution more effectively. It supports system collects, maintains and coordinates design record information. However, the major problems with such design record tool are that it is high-cost to keep documents up to date as the system is maintained with on-line editing, and collecting massive information does not equate to maintenance effectiveness. In order to efficiently support the developing and maintaining process, the design record should meet the following criteria:

- The maintenance process step supported by a design record should be a cost driver for software maintenance.
- Design record information should be collectible as automatically as possible as a part of an instrumented process. This reduces the burden on the maintainers for making the design record a reality.
- Design record information should be updateable as automatically as possible. This helps to ensure that the information can be kept current without burdening the maintainers.

The above overview reveals for us that the many methods proposed by different people of object-oriented analysis, design and supporting share various similarities. These methods all involve building models based on the object, separate static features from dynamic ones, and all provide diagrams as well as text to represent the models. Meanwhile, these methods all assume a fairly "classsical" process in which analysis and design precede implementation and maintenance. The reality is that almost all successful software evolve overtime which indicates that the analysis and design phases are always overlap. A good design method should reflect this overlap.

Chapter 3

Tool Design

In this chapter, we outline the issues that were considered during the design of the tool. The first section describes the goal and the objectives of the Design Tool, and the features of the target tool are listed. The second section presents the design format that is the theoretical basis for the tool, specifying what kind of design the tool will handle. The next section presents the search of suitable representation forms which will be used for display of the design in the tool. And finally, details of design decisions made for the tool are described.

3.1 Objectives And Expectations

As mentioned in Chapter 1, one typical situation in software design and maintenance can be described as follow:

Before a system is shipped for first release, the requirements, design and the code developed by analysts, designers, and implementors are made fully consistent with each other. But some time later, when the client has used the system and modified the requirements, the system maintainers have responded by changing the code to meet the new requirements, while the design of the system usually has not been updated to meet the changes in requirements and implementation. When this gap is increased beyond a certain extent, the system will become a complex entity which is unmaintainable and unrecognizable.

One major reason for this scenario is that design is not kept "light-weight": updating design during the system evolution is a heavy burden for designers and maintainers and may also be extremely costly. In order to keep consistency between requirements, design and implementation, the design must be light-weight[12] — easy to capture and easy to modify. If there is a computer-aided design tool that can be used by designers and maintainers, the tool should contribute to the speed, flexibility and ease of the design process and software evolution. The primary goal of our design tool is *to provide a facility for designers to explore and modify their designs as easily as possible and as quickly as possible.*

Some design tools are used to record decisions made during the design process, as well as other design information. In this way, the tool maintains a complete record of the design. But these tools are not effective on the following reasons:

- It depends largely on the developer's self-discipline to capture design information and keep it up to date. This is time consuming, costly, and therefore often neglected.

- To keep the design up to date is expensive, because it it not sufficient merely to add new information, it is also necessary to purge information that is no longer relevant. There is a need for mechanisms to keep the design current at reasonable cost, but on-line editing of the design does not satisfy this requirement.

A more advanced design tool is needed, which should provide facilities for creating, examining, checking, and modifying designs very easily.

The objectives of our design tool are as follows. It should:

▷ *Provide automated assistance for object oriented program development, with particularly strong support for the design phase.*

▷ *Support evolutionary development. We consider a software product to be a continuously evolving entity; changes affect all phases of development. At all times, the documentation of each phase should be consistent with the other phases.*

▷ *Help the designer to complete the design as quickly and easily as possible. The tool can do this by providing a variety of related functions that automate the tedious chores of design.*

We conclude this section with a brief outline of the features that we believe a design tool should provide. Each of these features is described in subsequent sections of the thesis. The tool should:

- provide the facilities to read, write, and display designs;
- provide multiple views of a design and provide facilities to change the design based on the view;
- provide the facility to check the consistency and completeness of the design upon request;
- provide the facility to generate high-quality printable reports for design archiving;
- respond to queries about the design, especially "what if?" queries;
- provide traceability to both requirements and code.
- designs provided by the tool can be implemented in any target language. The tool is biased, however, towards strongly-typed, class based languages, such as C++, Eiffel, and Dee. It would be straightforward to extend the tool to write class skeletons in any of these languages, but the present version of the tool does not do this.

3.2 Design Format

Rumbaugh suggests that a design should incorporate three distinct models of the target system[23]:

- The *functional model* describes the computations performed. A computation may be carried out by a single object or by an ensemble of objects working in harmony.

- The data or *object model* describes the way in which data is represented and managed within the system. In an object-oriented system, each object encapsulates a component of the data; objects may be parts of other objects.

- The *dynamic model* describes the flow of control within an object and between objects.

Our design notation does not explicitly differentiate between these models. It is organized around the object model. A system is composed of objects, so our design is essentially represented as a collection of class interfaces. The functional model can be extracted from the object model by examining descriptions of methods of the classes.

The dynamic model can not, in general, be extracted from object/class interfaces, therefore it must be described separately. Although state-charts are popular for dynamic modeling, other methods may be more suitable for object-oriented systems, like time-threads and uses cases. Our tool currently does not support the dynamic modeling of a system, and this is made an issue for future consideration.

A design, then, is a collection of class descriptions. Classes may be independent, but are usually components of subsystems, frameworks, and inheritance graphs.

Classes can be described at different levels of detail. The minimal description of a class could provide just its name. A full description includes the relation of the class to other classes, the instance variables, and a description of each method of the class. A method has a name and a signature, and may also have a specification and a list of methods that it may invoke. Obviously, the amount of useful feedback that the tool can provide to the designer should increase with the amount of information that the designer provides. Nevertheless, the tool's ability to process partial designs is crucial to its flexibility.

Our tool is based on a model of object-oriented computation, and assumes the following properties of a design:

- The design consists of a number of classes. Each class provides a number of methods, or services.

- There are currently two relations on classes: client/server (*uses*) and parent/child (*inherits*). The model may later be extended to provide other relations between classes.

- If class C uses class S, the client, C, needs one or more of the services provided by the server, S.

- There may be cycles in the *uses* relation: X may use Y and Y may use X.

- If class C inherits from class P, the child, C, provides all of the services of the parent, P, and may provide additional services.

- There are no cycles in the *inherits* relation.

- A class can provide three kinds of service:

  - ▷ a *constructor* creates a new object of the class.

  - ▷ an *observer* returns an attribute of the current object (for example, the age of a person).

  - ▷ a *mutator* changes the value of the current object (for example, changing the balance of an account).

- If the services provided by a class include one or more mutators, the class and its instances are called *mutable*. If no mutators are provided, the class and its instances are called *immutable*.

3.3 Design Display Forms

The principal purpose of the tool is to display the design in ways that permits the designer to explore and modify it as easily as possible. Consequently, the tool should provide different kinds of display.

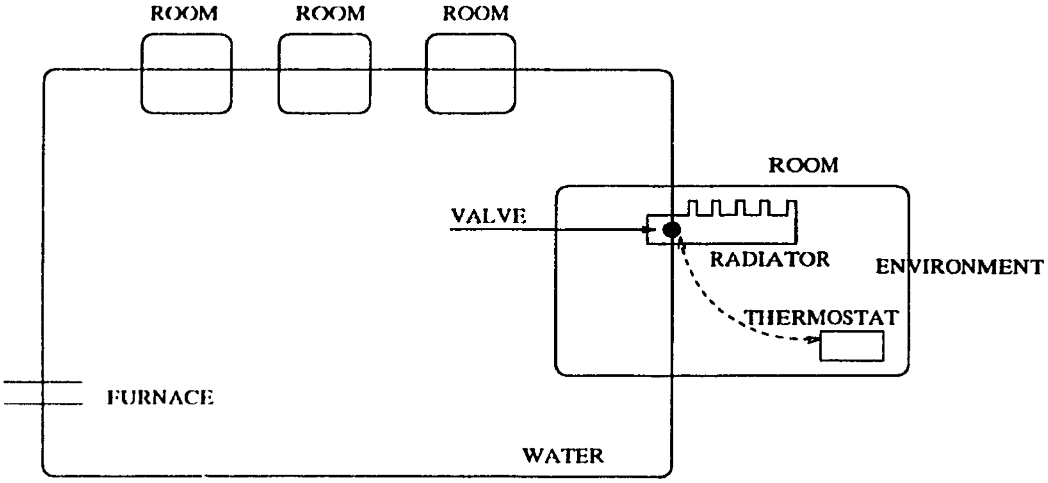


Figure 1: Simulation of a domestic heating system

Before we move to different kinds of display form, we introduce an example that we will use in the coming sections.

3.3.1 Example: a Heating System

Specification

The problem of the simulation of a simple central heating system can be described as follow:

There are several rooms in the house. The central heating system consists of a furnace which heats the water. The water runs through each room to pass the heat. There are a valve, a radiator and a thermostat which work in each room. The process can be described as. the furnace heats the water; the water heats radiators; the radiator heats rooms; the valve is set to on/off to control the heat into the room; the thermostat reflects the temperature in the room; and the rooms radiate heat into the environment. There can be from 1 to 8 rooms.

The user controls only the external temperature. If the external temperature is close to zero, the radiators go on and off and the temperatures oscillate. If it is very cold outside, say −20 degrees, the furnace runs all the time and the rooms reach an equilibrium temperature below their thermostat settings.

object	attribute
room	temperature, valve, radiator, thermostat, K(room,env.)
valve	switch (on/off)
radiator	temperature, K(radiator,room), K(water,radiator), switch(on/off)
thermostat	temperature-set, tolerance
furnace	switch (on/off), K(furnace,water)
water	temperature, pressure
environment	temperature

Table 1: Possible objects caught in heating system

The result of the simulation can be viewed in three different ways: analog, bar-chart, and analog graph.

Figure 1 is a diagram of a simulated domestic heating system.

Analysis

Using the method provided by [27], we first collect the objects of the system (since it's not a big system, no subsystem level decomposition is needed).

Possible objects may used in design of the system are:
room, valve, radiator, thermostat, furnace, water, environment.
Now we can list the attributes of each object obtained as shown in Table 1.

The `K(room, env.)` in Table 1 indicates that the temperature of a room is affected by the temperature of the environment, and this relationship can be represented by a constant value `K` which is an attribute of the room. Therefore, during the computation of a room, the effect of the environment to the room will be taken in consideration through the value of this `K(room, env.)` attribute. Other `K`s in the table have similar meanings.

Since object *valve* only has attribute *switch (on/off)*, we may make it an attribute of object *room.* Actually, *radiator* and *thermostat* can also be considered as attributes of *room* but, since they will take more complicated responsibilities and have additional attributes of their own, we leave them as independent objects. The object *environment* is taken in consideration by the room through the constant value

object	attribute
room	temperature, valve, radiator, thermostat, K(room,env.)
radiator	temperature, K(radiator,room), K(water,radiator), switch(on/off)
thermostat	temperature-set, tolerance
furnace	switch (on/off), K(furnace,water)
water	temperature, pressure
view	room temperature, time period

Table 2: Obtained objects of heating system

K(room, env.) and therefore can be eliminated from the list of objects. Meanwhile, the object *view* is added into the list which is responsible to display the result of the simulation.

Table 2 lists new objects and their attributes after we have made the above changes.

The next step is to assign proper responsibilities to each object that we have found. The *Responsibilities* of an object are all the services it provides for all the contracts (requests made by a client to a server) it supports. They include two key items: the knowledge an object maintains and the actions an object can perform. Responsibilities are meant to convey a sense of the purpose of an object and the role it will play in the system.

Followings are the responsibilities that we have identified for each object

room

Read room temperature, take into consideration the effect of the environment, send messages to radiator and thermostat for their control.

radiator

Acts as server of *room*. When the room temperature is above the setting temperature, set radiator/valve off, or when the room temperature is lower than certain value and while the switch is off, set it on.

thermostat

When temperature setting is within tolerance, set the valve on or off according to the temperature-setting and current temperature.

furnace

When *water* temperature is over certain value, set the switch off, or when heat is needed, set it on.

water

Maintain the temperature and pressure of the water.

view

Display the result of the simulation in three different ways: analog, barchart, and analog graph.

With the above analysis result at hand, we can start our object oriented design of the heating system.

Design

The system architecture is described in Figure 2.

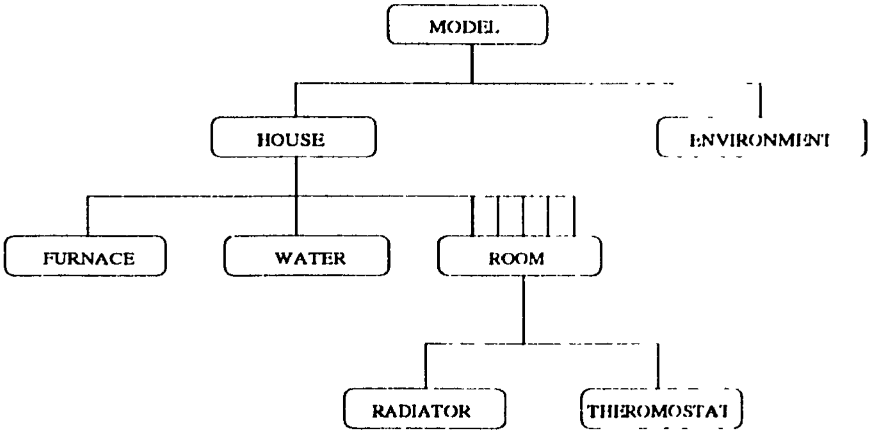


Figure 2: Architecture of the heating system

But Figure 2 does not give enough information about the system. In particular, it does not provide the information about how the components of the system communicate with each other, and what the control sequences in the process and among objects look like. We use Buhr's timethreads notation to add in this part.

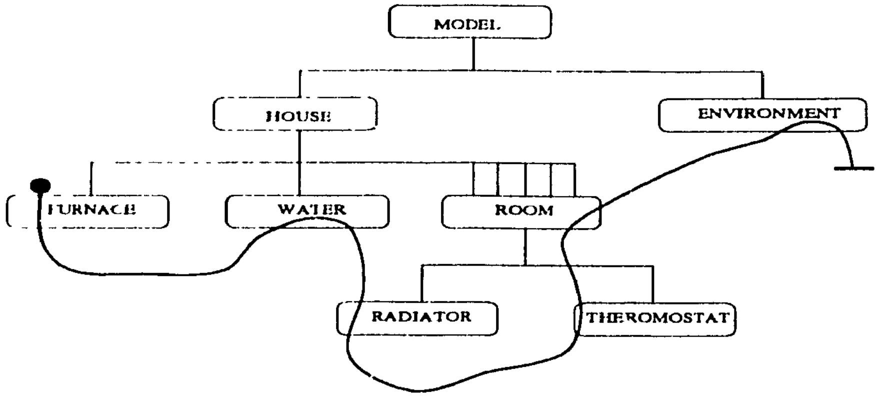


Figure 3: Timethread of the heating system

As mentioned in Chapter 2, timethreads notation is a design approach which jointly manipulates causality-flow scenarios and schematic components of organizations to make design decisions[8][7]. A timethread of the heating system is given in Figure 3, which indicates that when the heating system starts, first the furnace is heated and then the water; later, the radiator is heated through the water and change in the thermostat is made accordingly; finally the room is heated, and some heat is lost in the environment.

Figure 2 shows the Model component of an Model-View-Control system; Figure 4 shows the entire system.

Finally, we conclude the design by giving a list of class specifications based on [27]. These are given in Appendix A.

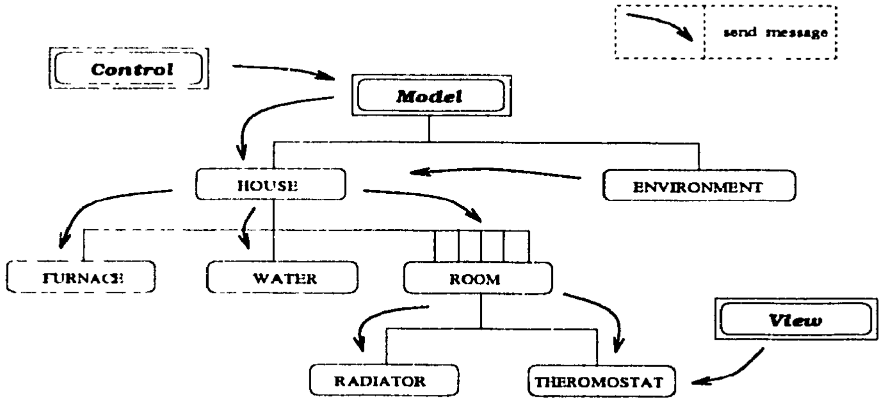


Figure 4: MVC view of architecture of the heating system

3.3.2 Textual Form

Textual form is the basic form provided for any kind of document, since text can be formal or informal, and can be handled by both human and computer. If we define the syntax of a design document, then the document can be read easily by a parser based on that syntax. For our design format, we define the syntax for the design which will be handled by our design tool.

The syntactic conventions that we use for design documents are listed below.

- Use lower-case characters for keywords. The only allowed keywords are: **system, class, inherits, uses, var, method** and **end**.

- A design may contain *comments*. Each comment begins with two dashes ("--") and continues to the end of the line containing the dashes. A group of lines beginning with two dashes is considered to be a single comment.

- The description of a system begins with the keyword **system**, followed immediately by the name of the system.

- A system has at least one class, and each class has a unique name inside the scope of the system; the description of a class begins with the keyword **class**, followed immediately by the name of the class.

- Inside a class, there is at most one inherits-list; the description of the inherits-list begins with the keyword **inherits**, followed immediately by one or more class names; if there is more than one class name in the list, class names are separated by commas or spaces.

- Inside a class, there is at most one uses-list; the description of the uses-list begins with the keyword **uses**, followed immediately by one or more class names; if there is more than one class name in the list, class names are separated by comma or space.

- Inside a class, there are zero or more var-lists; the description of each var-list begins with the keyword **var**, followed immediately by a *name:type* pair; the name indicates the variable name and type indicates a class name; name and type are connected by a colon (":").

- A class may contain zero or more methods. Each method is introduced by the keyword **method** and has a name that must be unique within the class. A method also has an optional argument list and an optional return type. The argument list is a list of *name:type* pairs enclosed in parentheses. The return type follows the argument list and is separated from it by a comma.

  Example: **method** *get* (*index*: *Int*): *String*

- Inside a method, there are zero or more uses-lists; the description of each uses-list begins with the keyword **uses**, followed immediately by a *name::type* pair; the name in the pair indicates the method(service) being used, and the type is a class name in which the method(service) is provided; name and type are connected by a double colon ("::").

- Each class is ended with the keyword **end** and followed by the class name.

- Comments are only allowed at certain places: after system name, after class name, after variable name and after method name.

Now we can generate the source text for the heating system according to the design in Section 3.3.1.

```
system  HeatingSystem
-- an OO design for a domestic heating system

class HeatSim
-- ancestors Program StdErrors
inherits StdErrors Program
uses Bool Textin Buffer ViewDigital ViewBarChart Float Water
     View Furnace Int ViewGraph Iterator Room List String
     Keyboard Window
method entry
```

```
end HeatSim

class View
-- An instance of a View class can display changing values of
-- several floating-point variables. This class, View, is
-- abstract: it defines the minimal protocol for a class that
-- provides views. A view has several channels, each
-- corresponding to a particular variable. Channels are
-- initialized and updated independently in any sequence. The
-- display is updated as a unit.
method init (channel:Int value:Float)
-- Initialize the given channel with the given value.
method set_val (channel:Int value:Float)
-- Update the given channel with the given value.
method set_title (channel:Int title:String)
-- Provide the given channel with the given title.
method calibrate (value:Float str_value:String)
-- A calibration point is used to label the axis of a graph or to
-- perform a similar service for another display mode. This
-- method uses the given value to position the calibration mark
-- and writes the string at that position.
method update
-- Update the display for all channels.
method message (txt:String)
-- Display a message other than channel data
method close
-- Null method to close display. Descendant classes which require
-- a closing action should redefine this method.
end View
......
class Water
```

```
-- Simulates the water in a heating system. The only interesting
-- attribute of the water is its temperature.
var temp:Float
method make (water_temp:Float)
-- Set the initial temperature of the water.
method change_temp (new_temp:Float)
-- Set temperature of water tc given value.
end Water
......
```

Although text is easy for the computer to handle and process, text is not always the ideal medium for presenting information to people. Readers must derive the information they need sentence by sentence, and deriving information from large blocks of text is sometimes inefficient. There are ways of presenting text, however, that allow people to extract the information they need very efficiently. To overcome the shortcomings of plain text displays, our tool can also present information in the form of tables.

3.3.3 Tabular Form

Tables are formal structures that can be processed easily and efficiently by computers. Tables can also be viewed as a form of picture, and it is well-known that people assimilate pictorial information very efficiently. A person can quickly obtain information from a table by scanning its rows and columns. Tables therefore provide information directly and in a straightforward manner. Besides, tables have several advantages when they are used in a human-computer interface:

- tables are easy to create;
- large tables, like spreadsheets, are easy to navigate by simple scrolling operations;
- tables make efficient use of precious screen space; microscopic fonts are not required; and

- small changes to the design tend to cause small changes to the tables.

Questions need to be answered before we start to build tables for the design.

▷ We know that the design is composed of classes of a system (the object-model), but what should the table look like so it can make the information contained by the design well organized and fully contained?

▷ One obvious way to display aspects of a large system in tabular form is to provide multi-level tables. But what levels should be provided, and how should the user select them?

▷ Finally, what are the attributes of each table at different levels?

The following considerations about tables refer to the questions above.

1. There are at least two kinds of tables. One is a *system-level* table which lists all the classes in a system and their descriptions. So when we get the table, we know what classes are in the system and what are they by one quick glance at the table; another is a *class-level* table which lists information about all the methods in the class, so we can get detailed information about any specific class that we choose.

2. The attributes of a system-level table should contain at least class name, its description, a list of inherit class names, and a list of the class names it uses. Thus the classes of a system and their relationship with other classes are obvious at a glance.

3. The attributes of a class-level table should at least consist of the method name, its description, its return type, its argument-list and list of the method::class pairs which the method uses.

4. There should be a system name and its description in the system-level table, so we know which system we are looking at.

5. There should be a class name and its description in the class-level table, so we know which class we are looking at without making any confusion.

Class name	Inherits Class	Uses Class	Description
HeatSim	StdErrors, Program	Bool, Textin, Buffer, ViewDigital, ViewBarChart, Float, Water View,Furnace, Int,ViewGraph, Iterator,Room, List, String Keyboard, Window	The simulation of a central heating system.
View			An instance of a View class can display changing values of several floating-point variables. This class, View, is abstract: it defines the minimal protocol for a clas that provides views. A view has several channels, each corresponding to a particular variable. Channels are initialized and updated independently in any sequence. The display is updated as a unit.
ViewDigital	View		Implements a View by displaying floating-point variables as strings.
ViewBarChart	View		Implements a View using horizontal bar charts.
ViewGraph	View		Implement a view by displaying variables on an X-Y graph.

Table 3: System-level table for the heating system

Class name	Inherits Class	Uses Class	Description
Furnace			Simulates the furnace in a simple central heating system. The furnace provides heat to the water water if any heat is needed. If no heat is needed, the water cools down.
Water			Simulates the water in a heating system. The only interesting attribute of the water is its temperature.
Room	Heatable, Any		Simulates a room which obtains heat from a radiator and loses heat to the environment. Room inherits from Any in order that we can declare an array of Rooms.
Radiator	Heatable		Simulates a radiator which receives heat from the water supply and provides heat to a room. The object simulated is actually a rather complicated radiator, complete with thermostat and valve. You could make the simulation more realistic by splitting this calss up into three components: a thermostat which switches on or off depending on the room temperature; a valve which is controlled by the thermostat; and a radiator which is controlled by the valve.

Table 4: System-level table for the heating system (*continued*)

Class name	Inherits Class	Uses Class	Description
Heatable			An abstract class which captures the basic property of a heatable object. A heatable object is connected to a source, from which it gains heat, and a sink, to which it loses heat. The object has a temperature and coupling coefficients to its source and sink.

Table 5: System-level table for the heating system (*continued*)

According to the above considerations, we can build tables for the heating system based on the design we developed in Section 3.3.1.

3.3.4 Diagram Form

Many of the popular design techniques are based on pictorial notations. Pictures are useful because they efficiently communicate information about the overall structure of a system. Pictures are an efficient form of information description, especially when they are associated with a more formal representation in another medium. Diagrams can be derived from text and are useful in providing quick confirmation that the expected relationships are present. The essential characteristic of a diagram is that it can be sketched quickly: this is why we use diagrams as aids to thinking and designing.

But pictures also lose some of their effectiveness when they become constrained—all structure charts tend to look the same—or cluttered with a plethora of symbols. Or when they are hard to draw. Diagrams are not an efficient way of communicating with a computer. Although computers can generate a visual representation of data that may help a person, they are not yet capable of extracting useful information from a rough sketch. This ineffectiveness of diagrams is particularly limiting if diagrams are the only, or even the dominant, way of representing designs.

MethodName	ReturnType	Parameter	Uses	Description
init		Int,Float		Initialize the given channel with the given value
set_val		Int,Float		update the given channel with the given value
set_title		Int,String		Provide the given channel with the given value
calibrate		Float, String		A calibrate point is used to label the axis of a graph or to perform a similar service for another display mode. This method uses the given value to position the calibrate mark and writes the string at the position.
update				update the display for all channels.
message		String		Display a message other than channel data
close				Null method to close display. Descendant classes which require a closing action should redefine this method

Table 6: Class-level table for the class *View*

MethodName	ReturnType	Parameter	Uses	Description
make		Float		Set the initial temperature of the water.
change_temp		Float		Set temperature of water to given value.

Table 7: Class-level table for the class *Water*

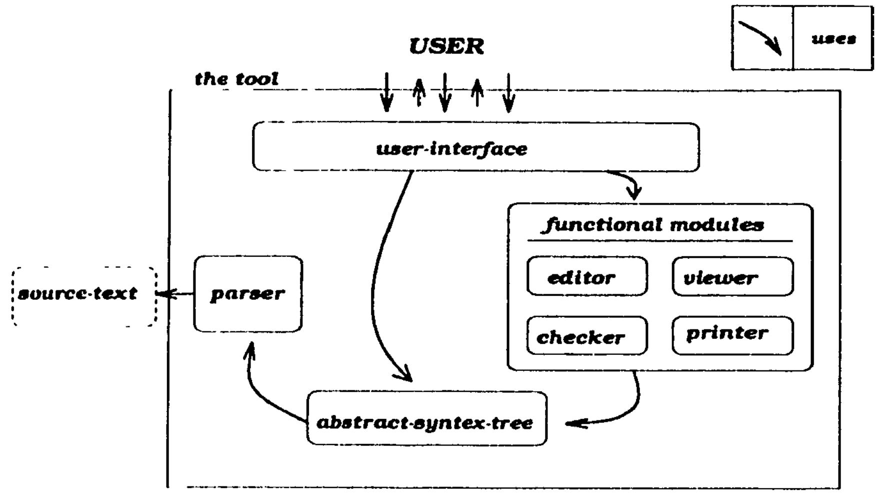


Figure 5: Organization of the tool

In our tool, the diagram form for display design is not provided in the first version, so we do not discuss it further in this thesis.

3.4 Design of the Tool

The issues discussed in this section include the functional structure of the tool; the most suitable data structure for the data repository (which turns out to be an abstract syntax tree); techniques for constructing and processing the abstract syntax tree; and the major services provided by the tool. In addition, we discuss the design of the user interface, including window layouts. We conclude with a brief discussion of the choice of implementation language and software libraries that we used to build the tool.

Due to the different aspects of the user-interface and its importance, user-interface will be described in a separate section later.

3.4.1 Organization of the Tool

Functionality and structure of the tool

As mentioned in Section 3.1, major functions of our tool are the ability to create, examine, check and modify the design, and to produce high-quality printable reports

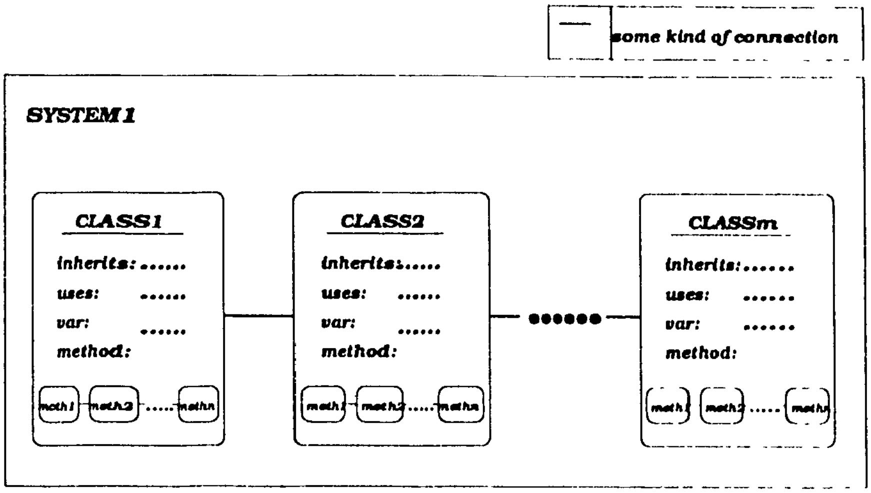


Figure 6: Abstract Syntax Tree of the tool

for the design. Based on this, we can divide the functionality of the tool into four service modules: a text-editor for creating and modifying the design, a viewer for examining and displaying the design, a checker for checking the consistency and completeness of the design, and a printer for producing high-quality output of the design. All these facilities are provided to users by a carefully designed, user friendly interface. The window-based interface is composed of different levels of windows which are equipped with various, but consistently formatted, pull-down menus and push buttons. All major functions can be activated by using the mouse only. However, all the functional modules can not be active without the data which are represented as the abstract syntax tree (AST). Every part of the tool has an intimate relationship with the AST. Whenever the user wants to modify, view, check or print a design, the tool needs to get data from the AST, then invokes the corresponding module for the user. The generation of the AST is the responsibility of the parser, which takes a file of source-text as input and constructs the AST. The parser is the first step in activating any functions in the tool. It reads source text, scans it, checks the syntax, and generates the AST.

Therefore the organization of the tool is composed of four parts: *data structure*, *parser*, *functional modules* and *user-interface*. These are shown in Figure 5.

Data Structure

The abstract syntax tree is required to store all the information about a design. Since a design is basically a list of class descriptions of a system, the AST should be able to hold arbitrary number of classes, and in turn, hold arbitrary number of variables and methods inside a class, as the design format specified in Section 3.2. Using a pictorial description, AST will look something like in Figure 6.

Parser

The parsing component will have three parts: one for scanning the input file and obtaining tokens for grammar checker, one for checking the syntax of input file and report syntax error, and the third one which is responsible for grabbing the data parsed and generating the AST. Once the syntax of the input file – source text – is determined, the scanner and the parser will be determined accordingly. The data grabber/clutter part will be fixed on the decision of AST.

Major functional modules

Text Editor

Text editor is the basic and conventional tool for creating and modifying text. The tool will provide a fully functional editor which allows the user to open and save files which contain design document, browse the design in text form, and modify it with convenient edit tool, like search, cut, paste, copy, etc., which are provided in the tool.

We assume that the design documents are files that with extension `.d`, so whenever the user wants to select a file for open or save, the tool will provide a file list, and only those files with extension `.d` appear on the list for selection.

Viewer

The viewer is a module which provides a tabular display of the system's design on a window. The tabular form is discussed in Section 3.3.3. The module will provide different tables on user's request. The user can choose to view system-level table by select a file which contains the design from a list of files provided. After the file has been parsed by the parser, and the AST has been generated, the module displays the system-level table like Table 3 in page 33. The user is allowed to choose to view a class-level table when a system-level table has been displayed - therefore, a system has been selected - otherwise, a warning message will be issued. If the class name selected does not belong to the system, a warning message will also be given.

The two kinds of tables will be displayed on the same window, therefore, only one table can be viewed at a time.

A *ScreenClear* function will be included in this module.

Checker

The ability to check the consistency and completeness of a design is a significant feature provided by our tool.

Consistency and *Completeness* mean every class and method used in the design is completely defined. Each class, whenever is inherited, is used, or is used as type of a variable, as return type of a method should be defined in the scope of the system, or in a standard library of the design whose source text is contained in a file named the same as the file containing the design, but with extension `.lib`.

Our checker will allow the user to select a file from a file list provided, and to invoke the checking by pressing a button. As in the case for Viewer, our parser will first parse the file and generate the AST which will be used by the Checker to perform checking. The check result will be given to the user which is either the information about undefined class names and method names or a succeed message.

For the convenience of the user, the consistency checker can be activated both as an independent action from a viewing window or from within the editor.

Printer

The printer is a module for generating output files of design in a form suitable for printing. We have found that LaTeX to be very suitable for this purpose: it is straightforward to generate LaTeX code, and the printed output is pleasing to the eyes. Although the development of a design is best done interactively, leisurely study of hard-copy often reveals area for improvement.

This module will allow the user to select a file from a file list provided by the tool, and start the process by a simple mouse click on a button. After the selected file has been parsed by our parser, and AST has been generated there after, the module creates a new file if there is no such file (or rewrite if there exists). The name of the new file is the same as the input file except that its extension is `.tex`. The printer then writes to this file with LaTeX form of the design. When successfully finishing the generating of the output file, the module will give a message to the user indicating the name of the generated file.

3.4.2 User-interface

Basic considerations

As mentioned in Section 3.1, the goal of our design tool is to provide a facility for designer to explore and modify the design as easily as possible. There are two parts that contribute to this goal: one is the functional modules as discussed before; the other is user-interface. Without interface, user and functional modules which are provided for achieving the goal are on two sides of a river. A carefully defined, and user friendly interface will sharply narrow the distance between what the user wants and what he/she will get. That is why we consider the interface part to be the significant and necessary part of our tool.

The most fundamental principle in user interface design is user compatibility, i.e. to know well about the user[18]. The users of our tool are the designers and maintainers of object-oriented software who have the knowledge about the design and the ability to create and modify the design. Meanwhile, some principles[18][24] are selected as guidelines in our design of the interface. They are listed as follow:

1. *Ease of use*: This characteristic makes the system acceptable and enjoyable to the user. Therefore, in our system, each window, button, menu and dialog and color or font used in the interface should be carefully organized so that the maximum of their potential self-explanation can be achieved, and the system is made explicit in both *what* and *how*. Conveniently obtained on-line help should be provided when it is necessary.

2. *Consistency*: Consistency allows people to reason by analogy and predict how to do things they have never done before, thus minimizing the need to consult a manual. It improves both performance and user satisfaction, as it imparts a sense of mastery to the user. We emphasize consistency within the system: the labeling and graphic conventions should be kept consistent as well as format in all displays; identical terminology should be used in prompts, menus and help screens, and consistent commands should be employed throughout.

3. *Simplicity*: One common mistake in interface design is to try to provide all the functionality in a complex interface which results in overwhelming and confusing details to the new user and tedious to navigate for the expert. One of our goals is to provide a rich complex functionality through a simple interface. The interface should not confuse the user by providing too much information on one screen or by presenting too many complicated functions in one step. This can be achieved through *layers* of the interface.

4. *Duration*: The system should tolerate common and unavoidable human errors. The system should offer simple error handling as much as possible. If an error is made, the system should detect the error and offer simple, comprehensible mechanisms for handling the error. The system should let the user feel confident that the system is robust enough to handle any kind of input, including errors.

5. *Familiarity*: Based on the natural human tendency to learn and reason by analogy, familiarity can greatly facilitate the learning of a new interface. Therefore, concepts, terminology and spatial arrangements that are familiar to the user will be used in the interface. Ambiguous words and description should be avoided

as much as possible.

Features of window layouts

According to the guidelines given above, we design the window layouts in the tool to contain the following features:

1. Each window only provides the necessary information for current/present actions, all other information which may be used later will be hidden from current appearance of the window, therefore the chance to make the user confused is largely eliminated. Advanced functions are presented to the user by multiple levels of window.

2. All the windows are carefully organized to give a simple appearance to the user. They are so straightforward that, with the exception of the MainWindow, no help windows are needed, although there are spaces for placing the *Help* button which may be needed when the system is extended in the future and more functions are added. Users can get information about the functions and usages of each window from the appearance of the window itself.

3. All the windows are consistent in their button placements: the *Quit* button on each working window is always on the left corner of the button or menu fields; the *Help* button on each window (if there exists) is always on the right corner of the button or menu fields; and the functional buttons on each working window are always on the middle of the button or menu fields.

4. All the *Information Dialogs* look the same: one message field and a button field; the button field only contains one button for the user to confirm the message and dismiss the dialog.

5. Whenever the user makes a mistake in input through either selection or typing, the system will give a message dialog corresponding to that error. Additionally, when the user want to exit the tool, a confirming message is provided. Protection is also made for the editor where designs are to be modified, and changes

should be saved before the updating information could be lost. Warning dialogs are used to remind the user a specific situation.

Structure of Window Layouts

ToolMainWindow

Since there are four major functional modules in the tool, and the user should be allowed to perform any of the four actions sequentially or in parallel, the tool interface should provide a top level window which incorporates the above actions and allow the users to select any one of them. This is the window we named *ToolMainWindow.* Each action is represented in this window by a push button. When the button is pressed, a corresponding new window is popped up for further action. As for Design, a new window named *DesignWindow* is popped up; for View, there is a *ViewWindow*; for Check there is *CheckWindow*; and for Output, there is an *OutputWindow*. Besides, inside the *ToolMainWindow*, a `Help` button is necessary and important which will pop up a *HelpWindow* and give descriptive information about the tool and its use. Also a `Quit` button for exiting the tool is provided.

HelpWindow

The *HelpWindow* should give the user all the information about the tool. The information that the window gives are displayed text, and the user is allowed to scroll the window when it is necessary. When the user finishes reading the text, the window can be dismissed by clicking on a button. The window should be retrieved by selecting the `Help` button in *ToolMainWindow* again.

DesignWindow

The *DesignWindow* is popped up to carry out the task of creating and modifying design upon user's request. There are two parts which form the layout of this window. On the top part, there is a message wrapped in a frame which gives description of this window, therefore the help message is provided directly; on the bottom part is the action field which consists of several buttons: a *Quit* button for exiting this window, a button named *Edit* for popping up a text editor, etc. When new functions, such as a *Query* window for a new way of creating and modifying the design, is to be added to the system, a new button with a name, such as *Query* (or another proper name) can be added to this part without difficulty.

ViewWindow

The *ViewWindow* is popped up to carry out the task of displaying the design upon user's request. Like the *DesignWindow*, this window is composed of a message field for the description of this window and an action field equipped with several buttons. As usual, a `Quit` button is provided. Tabular display is one of the major feature of our tool. Accordingly, a button named `Table` is presented for popping up a view-table window. When new functions of display of design are to be added to the tool, such as graphic display, new buttons, such as a `Graph` button, can be added to this field.

CheckWindow

The *CheckWindow* is popped up to carry out the task of checking the consistency and completeness of the design upon user's request. This window consists of three parts: on the top is the message field like the *DesignWindow* or the *ViewWindow*; in the middle there is a selection part which provides a file list for selection by mouse or by typing; in this field, the current directory is the default value, and switching to other directories is possible; on the bottom there is an action field which is composed of several buttons, such as a `Quit` button, a `ChangeDir` button, and a `Check` button for the check function on the system selected by the user. If an illegal file name is typed for selection or other error happens in the input file, a warning message is popped up to inform the user. After the information dialog has been dismissed, the user can continue with a new selection.

PrintWindow

The *PrintWindow* is popped up upon user's request to carry out the task of generating printable report of a design. Like the *CheckWindow*, it is composed of three parts: on the top is the message field for description of this window;

in the middle is the selection field which allows user to choose a system for printing by selecting a file from the file list provided or by typing in a file name on a given place; on the bottom is the action field with several buttons, such as a *Quit* button, a *ChangeDir* button and a *Print* button.

Implementation Language

We chose C as the implementation language and Motif as the toolkit for building user-interface. Although C has some drawbacks, it is the obvious language to use for software that will run under UNIX and will use X windows. There is a good debugging tool (**dbx**) available, and the interfaces to Motif and X are specified as C prototypes. It would have been possible to use another language, such as Pascal, but development within a mixed language system would probably have presented many problems.

Motif is a toolkit developed by the Open Software Foundation. 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)[14]. The Motif interface was intentionally modeled aftei IBM's Common User Access (CUA) specification, which defines the interface for OS/2 and Microsoft Windows. Its GUI was implemented by OSF using X as the window system and the X-toolkit(Xt) Intrinsics as the platform for the Application Programmer's Interface. Consequently, Motif is both portable and robust. 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, scollbars, and text-entry or display areas. We chose Motif because it is efficient, powerful, and popular, and because we like the pleasing layouts it provides.

Chapter 4

Tool Implementation

In this chapter, we discuss the design and implementation of the Design Tool. The tool is organized around a central data structure, the Abstract Syntax Tree or AST, that we describe in the first section. The AST is created as a side-effect of scanning and parsing the text of an object oriented design, as described in the second section. In the final section, we describe the user interface of the Design Tool: its overall structure, the windows and dialog boxes it uses, and the major functions used.

4.1 Abstract Syntax Tree

The Abstract Syntax Tree (AST) is the center part of our tool system. A proper definition of AST is crucial to all of the other parts of the system. Based on the design decision made in section 3.4.1, the AST can be built using C structures:

- A *system* is defined as a C structure which has several fields: a system name, its description, and a list of class definitions which belong to the system.

- A *class* is defined as a C structure which contains several fields: a class name, its description, the inherits class list, the uses class list, variable list, a list of method definitions and a pointer to the next class definition.

- A *method* definition is a C structure which consists of the method name, its description, its return type, a list of argument (name:type) pair, a list of uses (method::class) pair and a pointer to the next method definition.

The definitions of a system, a class and a method are given in Figure 7.

```
typedef struct TMethod {
     TLabel         label;      /* name and description */
     TTOPair        typeOrig;  /* type */
     TNTPairDict    *paraList; /* para:type pair */
     TNTPairDict    *usesList; /* method::class pair */
     struct TMethod *next;
   } TMethod;


typedef struct TClass {
     TLabel         label;       /* name and description */
     TInherits      *inherits;  /* inherits class list */
     TStringDict    *usesList;  /* uses class list */
     TVar           *varList;   /* var list */
     TMethod        *methodList;/* method list */
     struct TClass  *next;
   } TClass;


typedef struct TSystem {
     TLabel         label;       /* name and description */
     TClass         *classList; /* class list */
     struct TSystem *next;
   } TSystem;
```

Figure 7: AST of a system, a class and a method

Appendix B gives the complete declaration of the AST with explanations.

4.2 Parser

Our parser module is actually composed of three parts as mentioned in Section 3.4.1: a scanner for scanning the input file and generating 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 work inside the parser to generate the AST.

4.2.1 Scanner

```
switch(c) {
  case '-':   /*----handle comment----*/
    if (follow('-')) {  /* ``--'' are found */
      while (c != EOF) {
        c = ngetc();    /* continue to get next char */
        *tp++ = c;
        if (c == '\n')  /* until end of a line */
          break;
      }
      *--tp = '\0';     /* leave out the char '\n' */
      yylval.str = token;
      return COMMENT;
    }
```

Figure 8: C code of scanner for handling comment

Input streams in UNIX are byte streams. Converting the byte stream into a token stream is the task of a lexical analyzer, called a scanner. The scanner in our system is hand-coded and written in C (*lex.c*). Its major part is a procedure call *yylex* with `int` as its return type. The function of *yylex* is to examine the input stream and group individual characters into several distinct tokens, and return them through an external variable named *yylval* to pass the token value from the lexical analyzer to the parser. The names "yylex" and "yylval" are chosen for compatibility with Yacc, as described in the next section. We know from Section 3.3.2 that the keywords defined for the source text of design are *system, class, method, inherits, uses, var,* *end.* Besides, comments are defined as starting with double dashes and continuing until the end of the line. Therefore, *yylex* goes through each character in the input stream and compares each string obtained to the predefined keyword and also makes a copy of the word into *yylval.* If a match in the comparison is found, a corresponding keyword is returned, otherwise the string is returned as the semantic value of an identifier. Figure 8 gives the part of code for handling comments.

4.2.2 Parser

The function of the parser is to determine if a given source text is syntactically correct and to perform actions on correct text. The parser organizes the tokens it reads according to the rules of a grammar. When the parser has a sequence of tokens that corresponds to a rule, an associated action is executed. The actions can make use of the values of tokens to generate output or pass the values to other routines in the program.

Our parser is generated by *YACC*[17] — Yet Another Compiler-Compiler which is provided in UNIX. *Yacc* provides a general tool for imposing structure on the input to a computer program. The Yacc programmer prepares a specification of the input process, which includes rules describing the input structure, code to be invoked when these rules are recognized, and low-level routines to do the basic input. Yacc then generates a *parser*, which is a function named *yyparse*, and which calls the low-level input routine (the *yylex*) to pick up the basic items (the *tokens*) from the input stream. The tokens are organized according to the input structure rules; when one of these rules has been recognized, a corresponding action is invoked. Yacc generates its actions and output subroutines in C.

According to the syntax defined in Section 3.3.2, we prepared the specification of the input text. The specification is in the form of grammar rules with a set of actions. This is the file *parse.y.* The part of the rules for *system* written in Yacc is given in Figure 9.

This rule indicates that a system comes in the form that starts with keyword *system*, then a name for this system (the IDENTIFIER), then an optional comment,

```
system
    :  SYSTEM IDENTIFIER {
          settingSysName($2);
       }
       opt_comment {
          addComm(aSYSTEM);
        }
       opt_class_list
    ;
```

Figure 9: Grammar rule for *system*

and finally a list of classes.

4.2.3 Constructing the AST

The actions we specified in *parse.y* are a list of C routines which we defined separately in a file named *dataGrab.c*. When one of the grammar rules has been recognized by the parser, a corresponding action is invoked immediately. As in Figure 9, when in the source text something matches the rule *SYSTEM IDENTIFIER*, the action *settingSysName($2)* is invoked which saves the value of $2 (the semantic value passed by IDENTIFIER) to a proper C structure we defined for holding the system.

We declared several global variables in a file named *tempData.h* to hold the AST of a system being parsed. These variables are generated through the C routines which define the actions in the parser, and are used throughout the tool code. They will be initialized (by calling *initStruct*) and regenerated whenever a new source text is input for parsing. Actually, among these variables, only the `TSystem` typed variable *aSystem*, which contains all the semantic values of the system being parsed, is used after the parser phase. All other variables, like *aClass, aMethod* etc., are used only during the parse phase to help to catch information for generating the AST of the system.

Figure 10 gives the code of the operations *settingSysname* in the grammar rule of Figure 9. In some cases, one system depends on another. For example, many systems

```
/* create the TSystem typed pointer "aSystem"   */
/* and copy the given string to its "name"filed */
/*----------------------------------------------*/
void
settingSysName(str)
TString  str;
{
     TSystem  *tmp, *sysptr;

     tmp = (TSystem *)malloc(sizeof(TSystem));
     initSystem(tmp);
     tmp->label.name = (TString)strdup(str);
     tmp->next = NULL;
           /* always append the new created system to the tail */
           /* of the system list */
     sysptr = aSystem;
     if (sysptr) {   /* check to see if this is the first one */
        while (sysptr->next)  /* go to the tail */
            sysptr = sysptr->next;
        sysptr->next = tmp;
      }
     else {
        aSystem = tmp;
      }
   }
```

Figure 10: Operations of *settingSysName*

depend on a standard library. In these cases, the Design Tool parses the classes of the system first, and then the library classes, storing the entire system in a single AST. The system and its library are then checked for consistency as if they were a single unit. The newly created system of the library classes is always appended to the tail of the existing *aSystem* to indicating the close relationship of the two systems.

4.3 User Interface and Functional Modules

In our tool, the user-interface is closely related to the functional modules. For example, the *DesignWindow-Edit* is activated by calling the module of *editor*; the *ViewWindow-Table* is activated through calling the module of *viewer*, and so on. Therefore, we present the interface part and the functional module part together in this section. We will go through the functionality of major working windows provided by our tool, and describe the implementation issues in the following sequence:

1. *The structure of the block widgets* —- this gives the idea how we get the window layout in Motif, therefore the corresponding code for building the window is straightforward.

2. *The appearance of the window* —- this gives the actual layout of the window on screen, and therefore describes the appearance of the tool.

3. *Explanation of coding* —- this will include descriptions of some important routines in implementation, such as window creations, local variables and major computations, and so on.

Besides major working windows, a group of miscellaneous functions which play important roles in the functionality of the system will be described.

4.3.1 Major Windows

ToolMainWindow

The structure of widgets in the *ToolMainWindow* is portrayed in Figure 11, and Figure 12 gives the window layout.

This part also contains the main program of the system. It creates the toplevel shell and builds a number of different widgets on the top of the shell. The window is composed of two parts: on the top there is a message field, which is built with a label widget, that gives a welcome message; on the bottom, there is an action field, which consists of six buttons named `Quit`, `Design`, `View`, `Check`, `Output` and `Help`. The

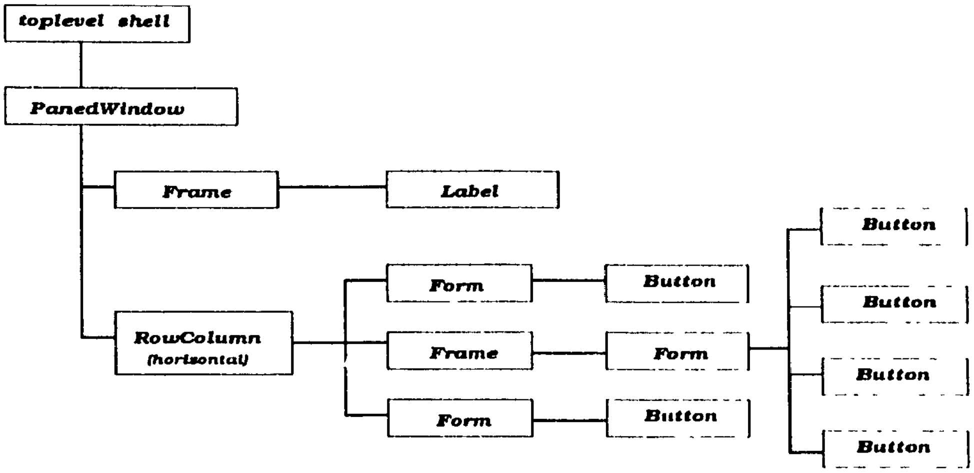


Figure 11: Structure of *ToolMainWindow*

Help button is made to be kept always on the right corner of the action field when the window is resized. When any of the buttons is pressed, a corresponding function, called the callback routine of this button, is invoked to carry out the action. The callback routine for Quit is *quitCb* which is defined in the same file scope; callback routines for Design, View, Check and Output are *designCb, viewCb, checkCb* and *outputCb* which are defined in files designw.c, vieww.c, checkw.c and outputw.c respectively; the callback routine for Help is *helpCb* which is defined in another file named helpw.c.

The function of *QuitCb* is to create a warning dialog when the Quit button is pressed. The warning dialog consists of two fields: a message field and an action field. It differs from other information dialog in that it has two buttons, Yes and No, instead of one. This warning dialog is created through calling convenient Motif function for creating warning dialog with specified button labels and warning message. When the user presses the Yes button, a further callback routine named *exitTool* which is defined in the same file scope is invoked to destroy the toplevel shell and exit the tool; if the user presses the No button, a corresponding callback routine named *notExitTool* which is defined in the same file scope is invoked to remove the warning dialog from the screen and leave all other windows unchanged. This warning dialog is defined as a static widget, therefore once it is created, it will be used throughout the life time of

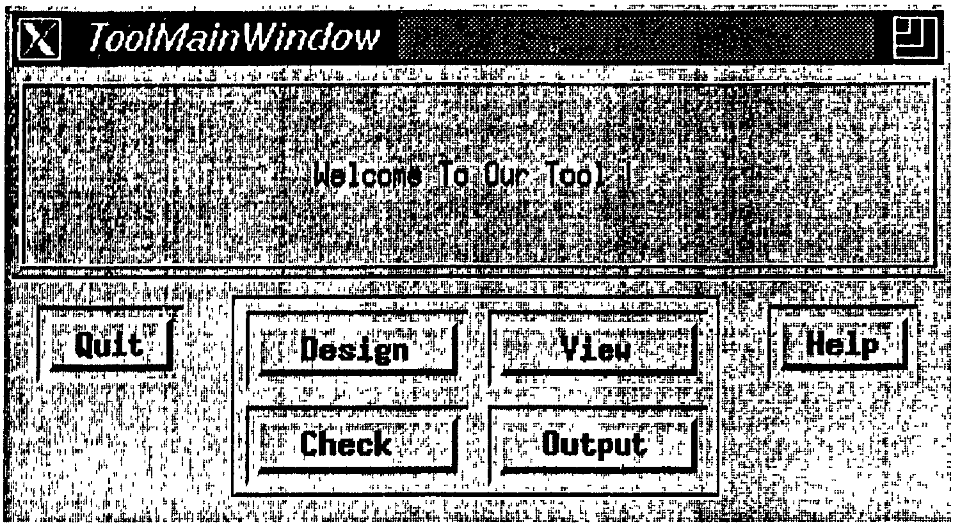


Figure 12: *ToolMainWindow*

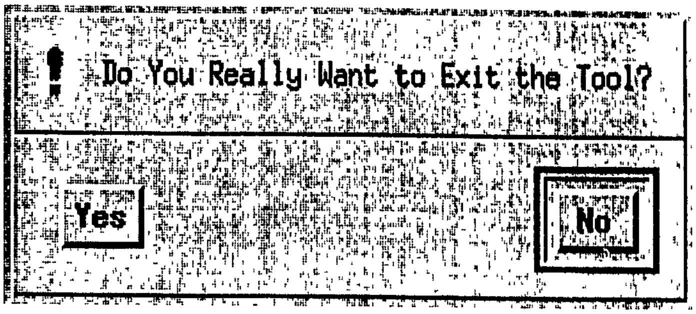
the system until the user exits the tool. It will be popped up on the screen whenever the `Quit` button is pressed in the *ToolMainWindow*, and will be removed from the screen when the `No` button is pressed. The warning dialog is shown in Figure 13.

HelpWindow

The structure of widgets in *HelpWindow* is given in Figure 14, and Figure 15 shows the layout of this window.

The shell of *HelpWindow* is created based on the parent of the `Help` button (by Motif convention), and is defined as static. Based on the shell, all other widgets of *HelpWindow* are defined. They are a *PanedWindow*, two *Forms*, a *Label* for display the image **i**, a *Text* widget for display help message, and two *Push Buttons*: one is `Ok` for dismissing the window and one is a nonactive button named `More`. The image **i** is contained in file `info.xbm`, and is loaded into the system during the creation of the *Label*. The help message is initially contained in a `String` array named *helpText*, and converted into a string array of type `char []`, then loaded into the system during the creation of the *Text* widget.

Like the warning dialog for exiting the tool, *HelpWindow* needs to be create only



Do You Really Want to Exit the Tool?

Figure 13: Warning Dialog for exiting *ToolMain Window*

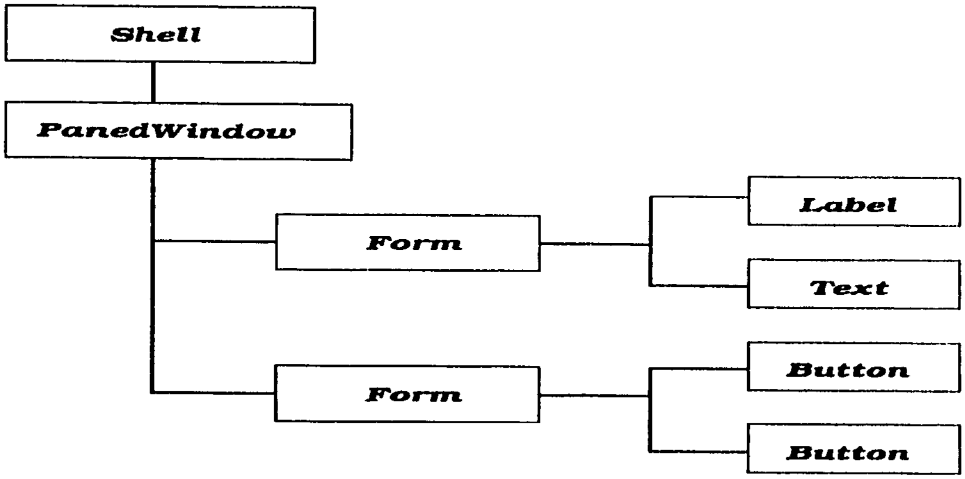
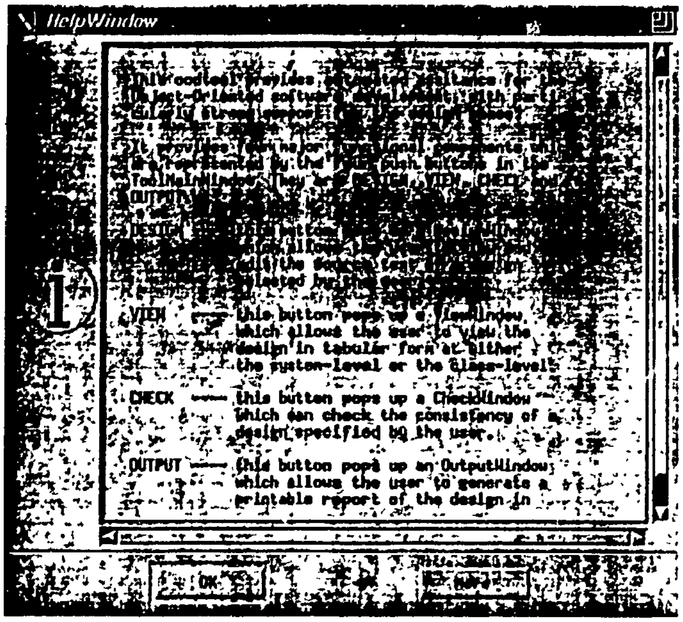


Figure 14: Structure of *HelpWindow*

at the first time when the user presses the `Help` button in *ToolMain Window*. Once it has been created, it will be directly popped up on the screen when the user presses the `Help` button. And it will be removed from the screen when the user presses the `Ok` button through the callback routine of the `Ok` button which is the miscellaneous function named *popdown Widget*.

DesignWindow

The structure of widgets in *Design Window* is given in Figure 16, and Figure 17 shows the layout of this window.



[illegible] provides [illegible] for the
Object-Oriented software [illegible]
[illegible]
It provides four major [illegible] components which
are represented by the four push buttons in the
ToolMainWindow. They are DESIGN, VIEW, CHECK and
OUTPUT [illegible]

DESIGN [illegible] this button [illegible]
which allows [illegible]
[illegible] the [illegible]
selected by the [illegible]

VIEW —— this button pops up a ViewWindow,
which allows the user to view the
design in tabular form at either
the system-level or the class-level.

CHECK —— this button pops up a CheckWindow
which can check the consistency of a
design specified by the user.

OUTPUT —— this button pops up an OutputWindow;
which allows the user to generate a
printable report of the design in

Figure 15: *HelpWindow*

As every popped up window must have a shell, the shell of the *DesignWindow* is created based on the parent ot the button `Design` of the *ToolMainWindow.* This *DesignWindow* consists of two parts: a message field for displaying the information of the design window, and an action field which contains buttons. The message is initially stored in a `String` array *designMsg*, and then loaded into the system during the creation of a *Label.* Since using a *Label* widget to display message is simpler and more direct than using a *Text* widget, a *Label* widget is used in this window for message display. A *Frame* widget is used with the *Label* widget to get a better appearance. There are four buttons in the action field, named `Quit, Edit, Query` and `Help`. The `Help` button is not necessary for the current version, therefore is made nonactive (say not *sensitive* in Motif) though it still appears in the window. It can be made active when necessary. It is made to be kept always on the right corner of the action field when the window is resized. Like *HelpWindow*, this window is declared as static, therefore once it is created, the only action to invoke it is to pop up its shell.

When the `Quit` button is pressed, a miscellaneous function named *popdownWidget*

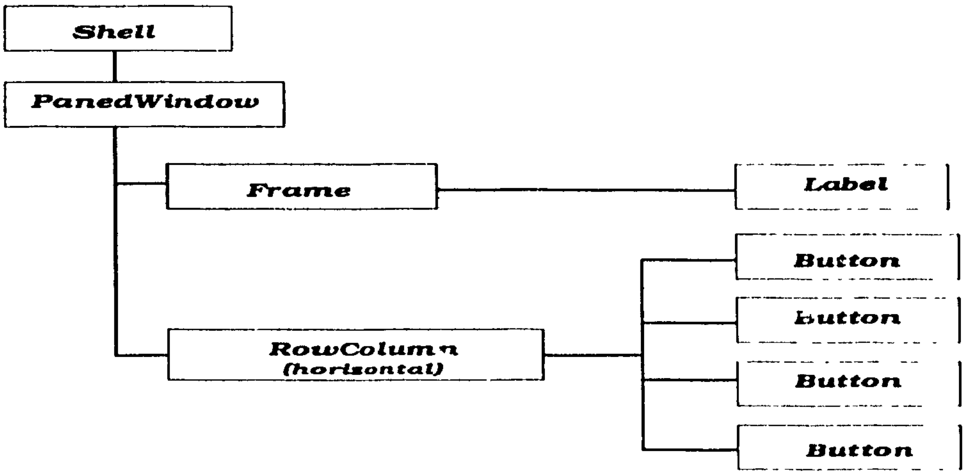


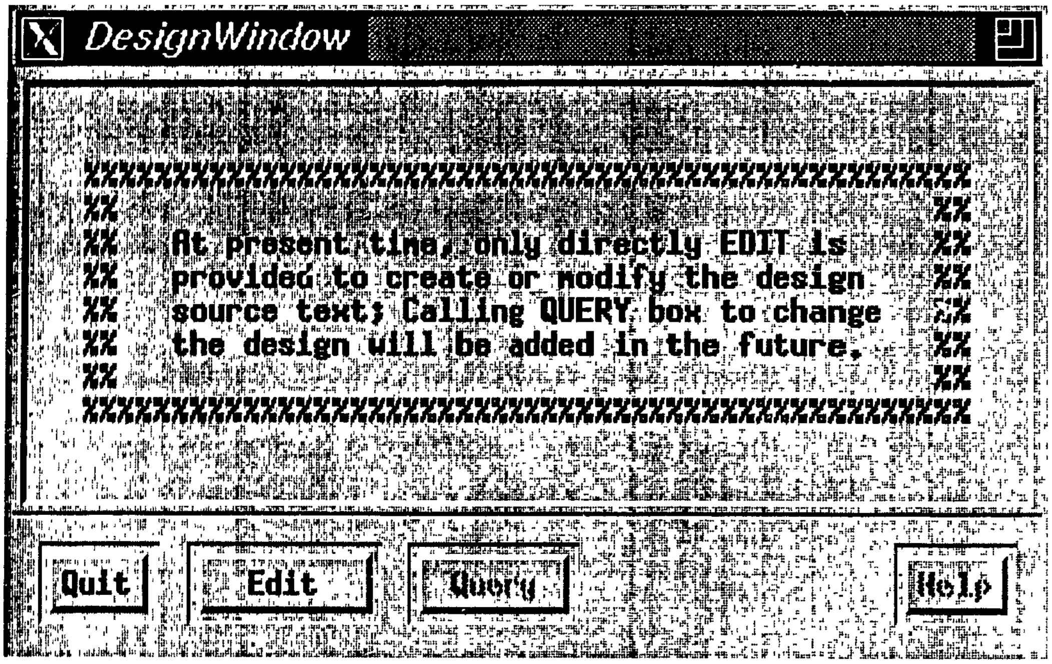
Figure 16: Structure of *DesignWindow*

is called to remove the shell, so the *DesignWindow* will disappear from the screen. The callback routine for the button `Edit` is *designEditCb* defined in file `dedit.c`. This function is responsible for creating the *DesignWindow—Edit* which provides the place for user to create or modify the design. The callback routine for the button `Query` is *designQueryCb* whose implementation is left to the future, and the button is made inaccessible from the window at present time.

DesignWindow—Edit

Figure 18 gives the structure of the widgets in the editor, and its window layout is shown in Figure 19.

The shell which is the base of all other widgets in the window is created as the child of the parent of the push button `Edit` in the *DesignWindow*. The *DesignWindow Edit* is composed of five parts which can be viewed from both the structure of widgets and the appearance of the window layout: on the top of the window there is a menu bar with six pull down menus which are labeled `Quit`, `File`, `Edit`, `Search`, `Option` and `Help` respectively. Each pull down menu has a list of one or more menus which can be selected and invoked to perform a specific action; following the menu bar is the place for input of searching and replacing strings. This part is created on a horizontal *RowColumn* with two pairs of *Label* and *TextField* widgets. The *Labels* are used for



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                           %%
%%    At present time, only directly EDIT is %%
%%    provided to create or modify the design %%
%%    source text; Calling QUERY box to change %%
%%    the design will be added in the future. %%
%%                                           %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 17: *DesignWindow*

displaying the string "Search Pattern" and "Replace Pattern". The *TextFields* are used by the user to input the patterns for searching and replacing; the next part in the *DesignWindow-Edit* is one of the message fields which is built on a horizontal *RowColumn* widget with a *Label* and *TextField.* This part is used for indicating the file name of a design whose source text is currently being loaded into the editor for editing; after that in *DesignWindow-Edit* is the working area of this window: the editing field, which is built with a *ScrolledText* widget. The size of the editing area is fixed at the time when the window is popped up on the screen, while the window can be resized and the user can scroll this editing area up and down and left and right with the scrollbar provided; finally, on the bottom is another message field which displays the information when one of the menu items in the pull down menus is invoked, such as it gives the number of bytes read during a file opening, or it displays the number of occurrences found when the user does a searching with a given pattern, etc.

Callback Routines

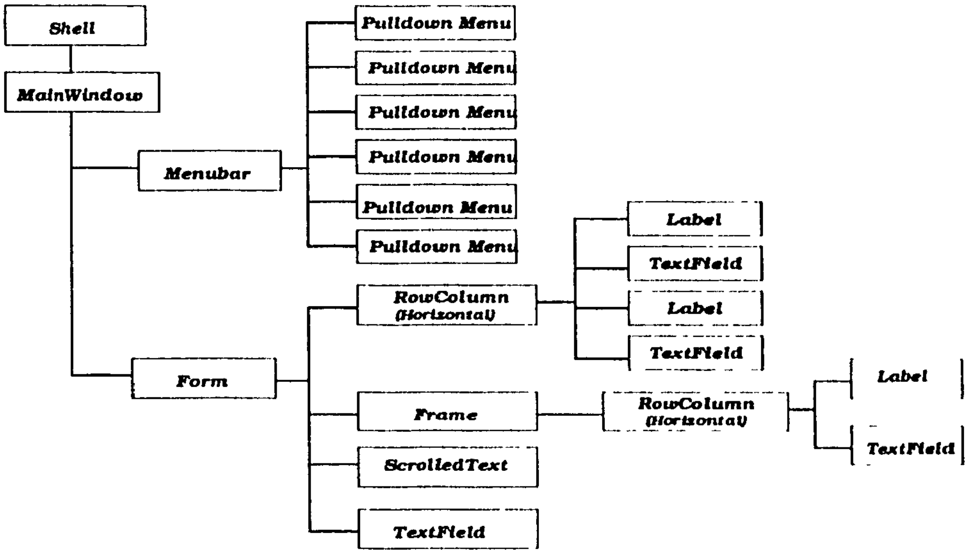


Figure 18: Structure of *DesignWindow - Edit*

Each pull down menus in the menu bar is associated with a callback routine which will be invoked when any of the menu items in the pull down menu is selected. There is one menu Quit in the pull down menu Quit with callback routine *editQuitCb* which is defined in the same file scope to pop down the shell of *DesignWindow-Edit.* There are two menus in pull down menu File: Open and Save; four menus in pull down menu Edit: Cut, Copy, Paste and Clear; four menu in Search: Find Next, Show All, Replace Text and Clear; and two menus in pull down menu Option: ScreenPrint and ConsistCheck. The callback routines for the pulldown menus File, Edit, Search and Option are the functions *fileCb, editCb, searchCb* and *editOptionCb* which are defined in the same file scope. The Help menu is placed always on the right corner of the menu bar as the Quit is placed on the left corner of the menu bar. The callback routine for Help is defined as *editHelpCb* and its implementation is left to future work when it is necessary.

fileCb for pulldown menu File

The callback routine *fileCb* is invoked when any of the two menu items in pulldown menu File is selected. The routine's major function is to create a corresponding file selection dialog for the menu item Open or Save. The file selection

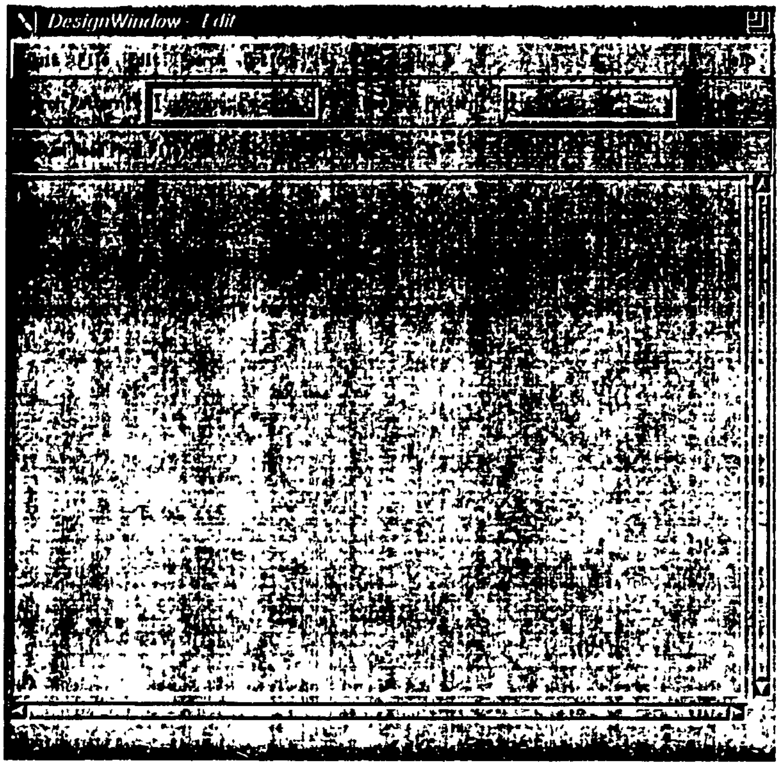


Figure 19: *DesignWindow—Edit*

dialog is created by calling a convenient Motif function with some modifications in file searching procedure and button labels. The new file searching procedure is a miscellaneous function named *mySearchProc* which searches the current directory for those files with extension `.d` or `.lib`. `Open` is the new button label for the default button *Ok* in the file selection dialog popped up when the user selects `Open` menu. Accordingly, `Save` is the new label in the dialog for menu `Save`. The label of original button *Filter* is changed to `ChangDir` for better understanding. The button `Cancel` is left unchanged.

There are callback routines for each buttons in the file selection dialog. The callback routine for `Cancel` is the function *popdownWidget* which is responsible for making the dialog disappear from the screen when the button `Cancel` is pressed. The callback routines for the button `Open` and `Save` in two dialogs are the same function *file_select_cb*. The responsibilities of this routine include the

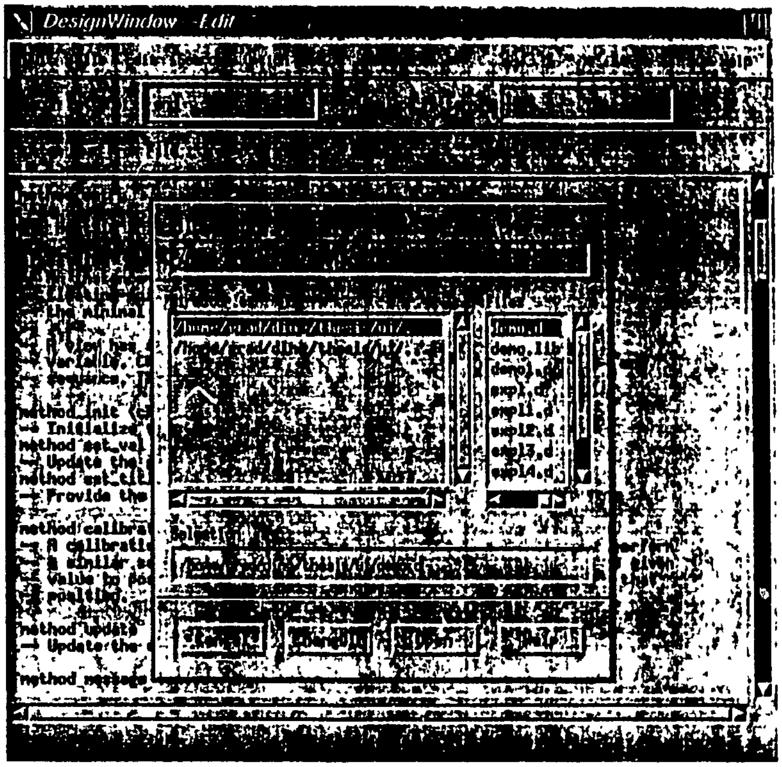


Figure 20: *DesignWindow—Edit* with **Open** selected in pulldown menu `File`

operations of obtaining the file name from the selection field in the dialog, and performing the action of file opening or file saving accordingly. If no file name is provided in the selection field, or a given file can not be opened, a corresponding message is displayed in the message field on the bottom of *DesignWindow Edit.* When the operation of reading text from a file (for menu `Open`) to the editing area, or writing the text in the *Text* widget (for `Save`) to a file is accomplished, a message for indicating the number of bytes read or written and the full name of the file is displayed in *DesignWindow-Edit.* Meanwhile, when a file is loaded, the file name appears on the message field up the editing area in *DesignWindow Edit.* An example with the file select dialog popped up is show in Figure 20.

editCb for pulldown menu `Edit`

The callback routine *editCb* is invoked when any of the four menu items in the pulldown menu `Edit` is selected. The procedure uses a variable *event* to capture the user's selection, which may be made either by clicking the mouse or by pressing a key. The selection is highlighted in the text. If the user selects the menu `Cut`, the procedure responses by cutting the selection from the text; if menu item `Copy` is selected, the procedure copys the user's selection into a clipboard preserved by Motif; if menu item `Paste` is chosen, the procedure pastes the content in the clipboard to the place where current cursor is. If no selection is made by the user before one of these options is selected, a message which says "There is no selection." is displayed on the message field of *DesignWindow-Edit*. If the user selects the menu item `Clear`, the procedure simply deletes the content in the clipboard and removes the highlight made by the user during the selection operation.

searchCb for pulldown menu `Search`

This routine is invoked when any of the four menu items in pulldown menu `Search` is selected. If the user selects the item `Clear`, the routine simply removes all the highlights in the text. For the other three menu items, the routine first checks if there is any text in the editing area. If there is none, a message which says "No text to search" is displayed on the message field on the bottom of *DesignWindow-Edit*, and the the procedure returns. Otherwise, the routine continues by checking if a search pattern is given in the window, if none, the routine gives a message and returns. When both text and a search pattern are available, and if the menu item `Find Next` is selected, the routine starts searching from current cursor position until a pattern is found in the text; if no matching pattern is found, a new search from position 0 of the text is invoked until a pattern is found in the text. Whenever the pattern is found or not found in the text, a corresponding message is given on the bottom of the *DesignWindow-Edit*. If the menu item `Show All` or `Replace Text` is selected, the procedure searches all the text, and highlights the found pattern(s) (for `Show All`), or replaces the found pattern(s) (for `Replace Text`), then displays a corresponding message for indicating the number of patterns found in the window of *DesignWindow-Edit*.

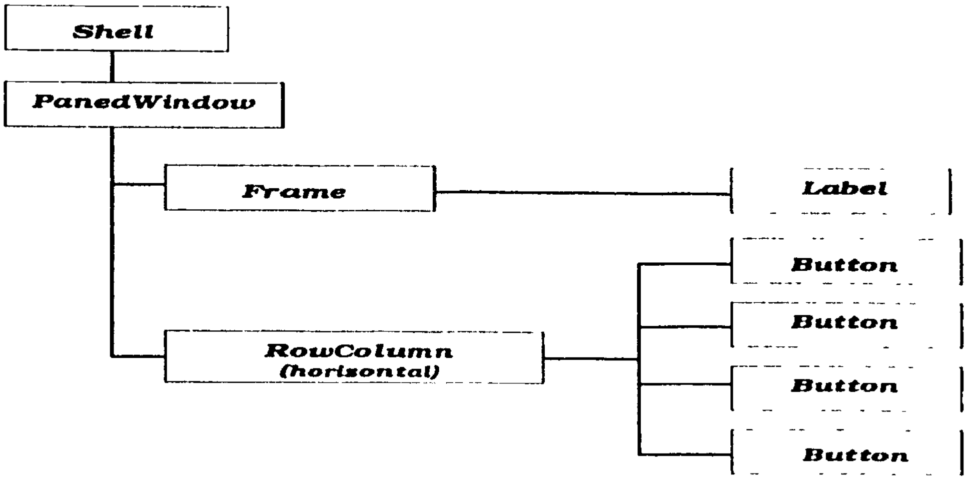


Figure 21: Structure of *ViewWindow*

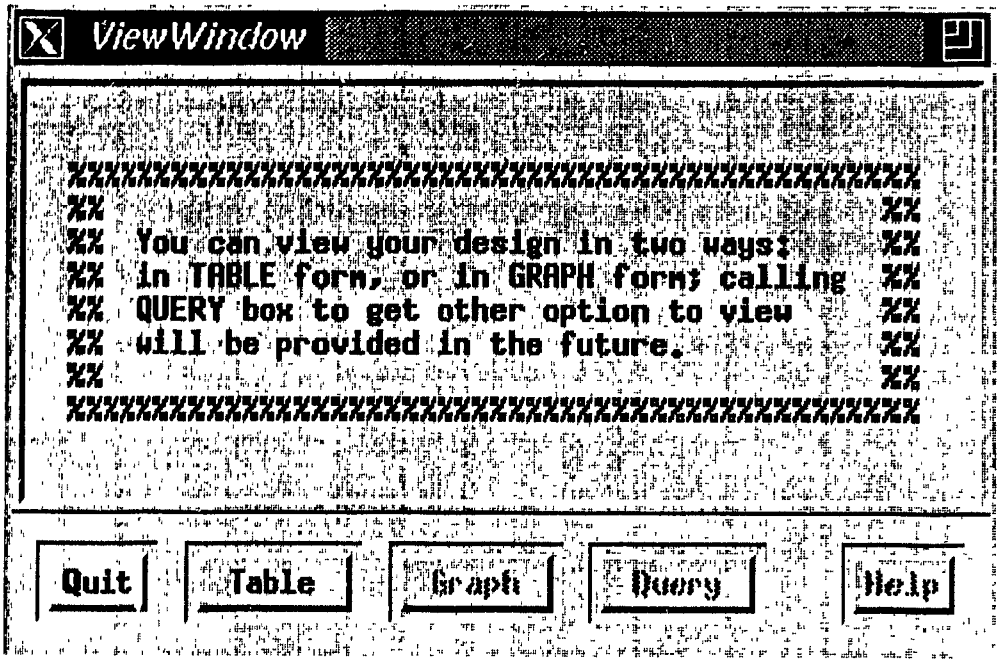
editOptionCb for pulldown menu `Option`

This routine is invoked when the user select the menu items in pulldown menu `Option`. There are two menu items in this pulldown menu: `ScreenPrint` and `ConsistCheck`, but only the latter one is implemented in the first version of the tool. When `ConsistCheck` is selected by the user, *editOptionCb* responds by first checking if a design source text has been currently loaded into the editing area. If there is current design, it continues by calling a function named *checkFunc*, which is defined in file `checkw.c`, and in the meantime passing the file name to this function. The function *checkFunc* is responsible for doing the consistency check of the design. If there is no file name available, a message which says "Choose a file" is given in the *DesignWindow Edit*.

ViewWindow

Figure 21 gives the structure of the widgets of the *ViewWindow*, and the window layout is shown in Figure 22.

Like the *DesignWindow*, the shell of this window is created as the child of the parent of the button `View` in *ToolMain Window*. There are two parts in the *ViewWindow*: a message field for displaying information about this window which is built with a *Frame* and a *Label*; and an action field equipped with several buttons. The buttons



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                             %%
%%  You can view your design in two ways:      %%
%%  in TABLE form, or in GRAPH form; calling   %%
%%  QUERY box to get other option to view      %%
%%  will be provided in the future.            %%
%%                                             %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 22: *ViewWindow*

in the window are `Quit`, `Table`, `Graph`, `Query` and `Help`. The callback routine of `Quit` button is the common function *popdownWidget* which is responsible for removing the shell of this window, thus making the window disappear from the screen. The callback routine for button `Table` is the function *viewTableCb* which is defined in file `vtable.c` and is to be described in the next subsection. As in all other windows, the `Help` button is kept always on the right corner of the action field, and `Quit` button is kept on the left corner. The implementation of the callback routines for `Graph` and `Query` are left to future work, and the buttons are made inaccessible from the window for the current version. The button `Help` is also made inaccessible because no help is needed and therefore is not provided for current version.

ViewWindow—Table

The structure of the widgets in this window is given in Figure 23, and Figure 24 shows the layout of the window.

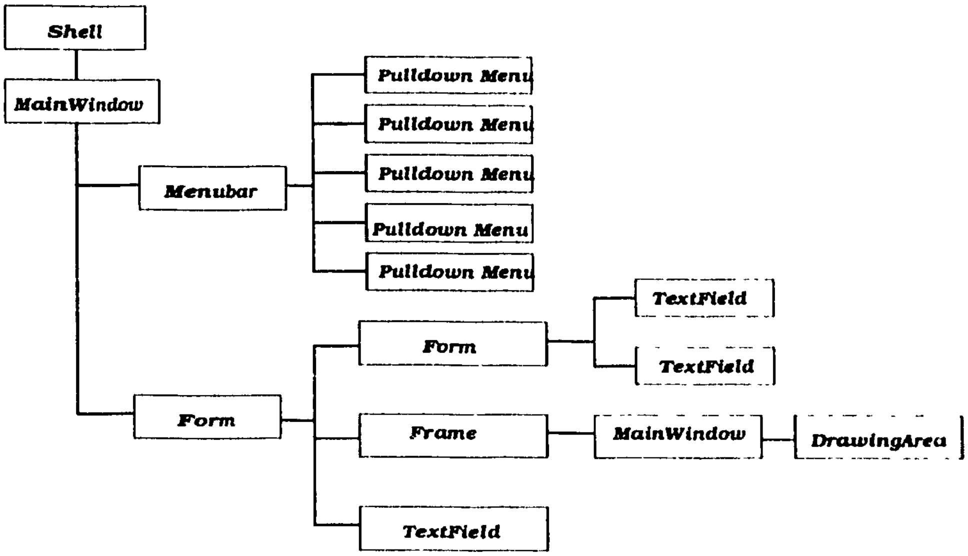


Figure 23: Structure of *ViewWindow– Table*

The shell of *ViewWindow-Table* is created as a child of the parent of the button `Table` in *ViewWindow.* The body of this *ViewWindow-Table* is composed of four parts as shown in the layout of the window. Like the *DesignWindow -Edit*, on the top of the window, there is a menu bar which consists of four pulldown menus. From left to right, these pulldown menus are **`Quit`**, **`TableView`**, **`Option`** and **`Help`**. Following the convention of the window layout in the system, the **`Help`** is kept far away on the right corner of the menu bar. Beneath the menu bar, there is a message field which is built with two *TextField* widgets. This message field (we call it *top message field* in order to distinguish it from another message field on the bottom of the window) is used for displaying the information about a table, such as a system name for a system-level table, or a class name for a class-level table. The next part in the window is the drawing area which is used for displaying tables. Finally, on the bottom of the window is another message field for indicating the name of the file which holds the source text of the design whose table is currently being displayed (for system-level table), or the name of the system of whom the class-level table is currently being displayed.

Callback routines

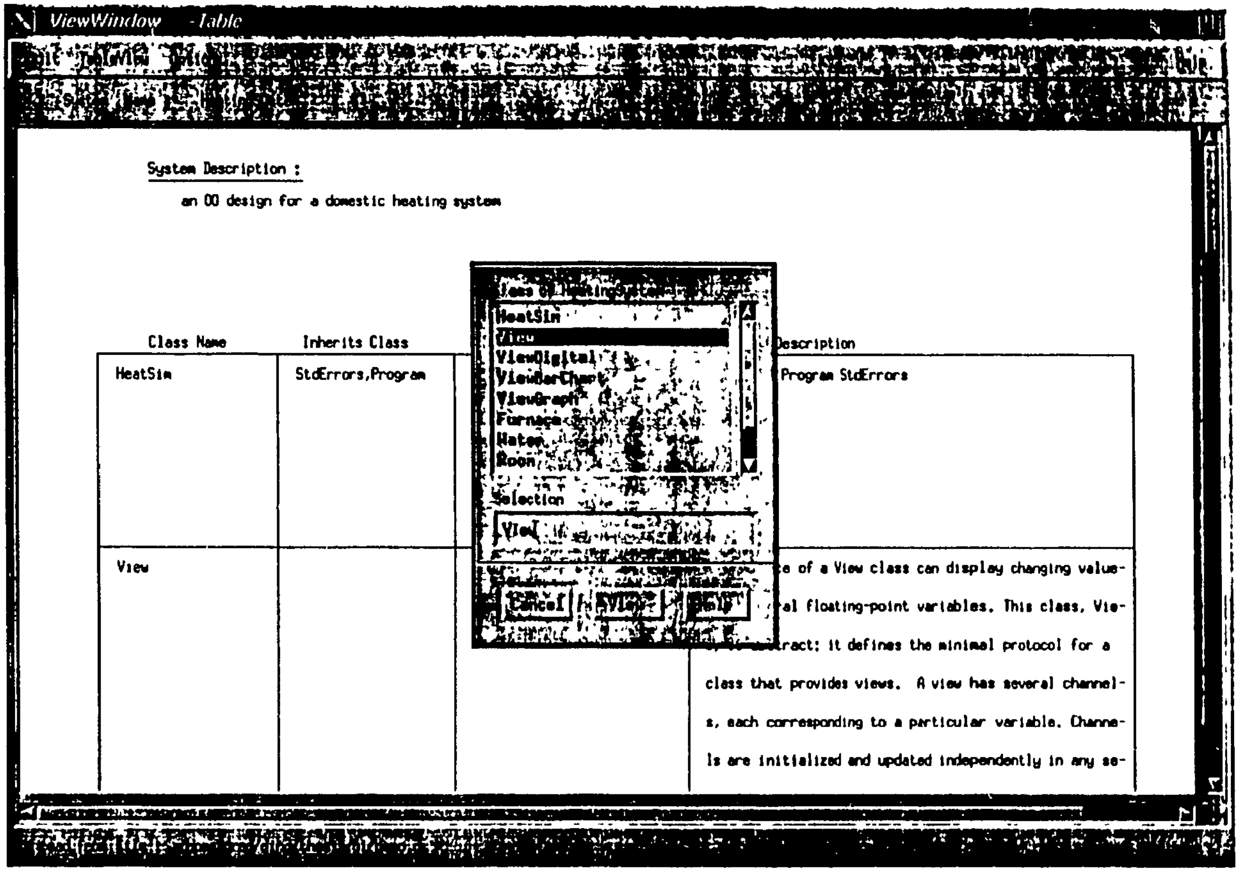
Each pulldown menu in the menu bar is associated with a callback routine

Figure 24: *ViewWindow—Table*

which will be invoked when any of the menu items in the pulldown menu is selected. As in *DesignWindow-Edit*, there is one menu items, `Quit`, in the pulldown menu `Quit` whose callback routine is *vtQuitCb*. This routine simply *pops down* (removes and can be retrieved later) the shell of *ViewWindow-Table* when `Quit` is selected. There are two menu items, `System` and `Class`, in the pulldown menu `TableView`. The callback routine for this pulldown menu is the function *tableCb* which is defined in the same file scope. There is one menu item, `ScreenClear`, in the pulldown menu `option` whose callback routine is *tableOptionCb*. The responsibility of `ScreenClear` is to clear the drawing area and the message fields. Therefore the callback routine *tableOptionCb* simply sets the text of all the *TextFields* to NULL, and calls the function *clearDrawArea* to remove anything in the drawing area.

Callback *tableCb* for pulldown menu `TableView`

The routine *tableCb* is invoked when any of the two menu items, `System` and `Class`, in the pulldown menu `TableView` is selected. If the user selects the item `System` which means a system-level table of a design needs to be displayed for



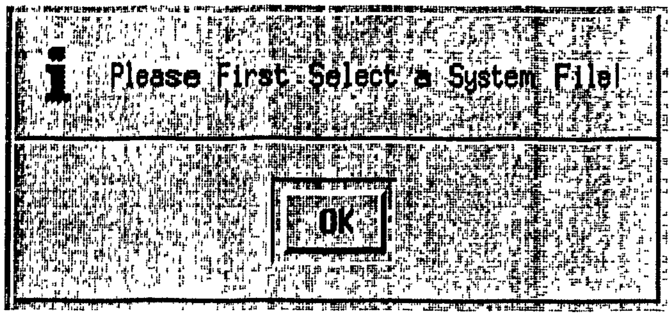
System Description :

an OO design for a domestic heating system

Class Name	Inherits Class		Description
HeatSim	StdErrors,Program		Program StdErrors
View			e of a View class can display changing value- al floating-point variables. This class, Vie- ract: it defines the minimal protocol for a class that provides views. A view has several channel- s, each corresponding to a particular variable. Channe- ls are initialized and updated independently in any se-

Figure 25: *ViewWindow—Table* with `Class` selected in pulldown menu `TableView`

the user, the procedure responses by popping up a file selection dialog for the user from which a file that contains the source text of the design can be selected. This dialog is created by calling a convenient Motif function with some changes in the file searching procedure and button labels. As all other file selection dialogs used in the tool, this dialog uses the function *mySearchProc* to search for displaying only those files with extension `.d` or `.lib` in the current working directory. Besides, the label of the *OK* button is changed to `View`, and the `Help` button is not necessary, therefore is made inaccessible. As usual, the label of the button *Filter* is changed to `ChangDir`, and `Cancel` button is left unchanged. The callback routine for `Cancel` is *popdownWidget*, which is responsible for



Please First Select a System File!

Figure 26: An information dialog

removing the dialog from the screen when Cancel is pressed. The callback routine for the button View in the dialog is the function *sysTableCb*. As usual, this file selection dialog is declared static.

If the user selects the menu item Class in the pulldown menu TableView to view a class-level table of a design, the routine *tableCb* first checks if there is a design available, i.e. if there has been a file selected from the file selection dialog which is popped up by choosing the menu item System. Before a class-level table can be displayed, a design must have been loaded into the tool. A variable "currentSys" is used to hold the abstract syntax tree of the design which is the most recently loaded into the tool. The variable is initialized to NULL when the system starts, and is checked before a class-level table is to be drawn. If the value of "currentSys" is NULL, i.e. no design loaded into the tool, *tableCb* is responsible for popping up an information dialog which informs the user to select a design through the menu item System, and then the routine *tableCb* returns. The information dialog is created by calling a miscellaneous function *createInfoDialog*. Otherwise, if a design is available, *tableCb* creates a selection dialog in which a list of class names of the design is displayed. This selection dialog is created by calling a convenient Motif function with some changes in button labels: the label of button *OK* is changed to View; the Cancel button is set on the left corner of the action field of the dialog; since further help text is not necessary, the Help button is made inaccessible. The list of class names is obtained from the AST of the design, and is loaded into the dialog before the dialog is managed and popped up to the screen. The dialog is declared static, therefore it needs to be created only once. But the list of class names of the design must be prepared each time when the selection dialog is to be popped up due to the difference of the classes in different designs. There are two buttons accessible in this selection dialog: `Cancel` and `View`. The callback routine for `Cancel` is *popdownWidget* which removes the dialog when `Cancel` is pressed. The callback routine for `View` is the function *clsTableCb*. Besides the callback *clsTableCb* for button `View`, another callback routine *noMatchClassCb* is also provided for the button `View`. This *noMatchClassCb* is to be invoked when the user presses the `View` button, but the selection made by the user does not match any one in the list displayed in the dialog. The no match callback function simply pops up an information dialog to inform the user that the selection is improper. This information dialog is also created by calling *creatIngoDialog* and setting a corresponding message in its message field before managing it and popping it up. An example with the menu item `Class` being selected is shown in Figure 25. Figure 26 gives the layout of an information dialog.

Callback *sysTableCb* of `View` in the file selection dialog

The callback routine *sysTableCb* is invoked when the user presses the `View` button in the file selection dialog which is popped up by the selection of item `System` in menu `TableView`. The responsibilities of *sysTableCb* include:

- obtaining the file name from the selection field of the dialog;
- calling the parser to parse the file and generate the Abstract Syntax Tree for the design;
- popping up an information dialog when the AST is not generated due to some reason such as file can not be opened or the parsing fails, or
- (when parsing succeeds) drawing the system-level table of the design as well as displaying the messages on the message fields in *ViewWindow Table*.

The parsing is invoked by calling the function *ast*, and the generated Abstract

Syntax Tree is held in a global variable *aSystem* in file *tempData.h.* As before, the information dialog is created with *createInfoDialog.* The drawing of the table is accomplished by calling a function *drawSystemTable.*

In *drawSystemTable*, some local variables are defined for representing the positions of lines and text of a table, and a new C structure *TMultiStr* is defined for holding multiple lines of a string. There are two fields in this C structure: a `char` array for holding a string and an `int` for holding the length of the string. Variables which are declared as arrays of such type are used frequently in building a table. Several library routines in Xlib[3] are called to accomplish the task of drawing, such as XDrawLine, XDrawRectangle and XDrawString. Since the window is allowed to be resized, a pixel map is created to store a copy of the table in the drawing area. When the window is resized or when the user scrolls the window by the provided scrollbar, the *drawSystemTable* is responsible to handle the expose event by redrawing the table. This is accomplished by adding to the window an "exposeCallback", the function *redraw*, which copys the table from the pixel map to the drawing area.

In a system-level table, the system name of a design is displayed in the top message field of the window. If there is a description (comments) about the system in the design, it is to be displayed on the top of the drawing area. The length of one line for displaying the system description is prefixed by the program as indicated by the variable *scwd* (the value of this variable represents the number of pixels). The description is actually a group of characters including double dashes ("--") and carriage returns ("\n"). Before this string is to be displayed, all the double dashes and carriage returns are removed by calling a miscellaneous function *rmRnDash.* The length of the new string is compared with the value of *scwd* to see if the string can be held completely in one line. In our tool, one character takes about 6 pixels based on the font we chosen. If one line is not long enough to hold the string of system description, new lines are added and carriage returns are inserted into the string accordingly, by calling the function *addRetn*, until the string of the system description is completely drawn

From Section 3.3.3, we know there are four columns in a system-level table. From left to right, the columns are labeled **Class Name, Inherits Class, Uses Class** and **Description**. The width of each column is prefixed in the program, but leaving the height (number of lines) of a row adjustable for holding different length of strings in a row. The strings for labeling each column are drawn in a line under the system description, and their positions are calculated according to the width of each column. The content of the table is drawn from row to row, and is done by first drawing a rectangle and then drawing a string inside the rectangle. As mentioned before, the width of each column is prefixed, while the number of lines in a row depends on the length of the strings which are to be displayed in this row. The number of lines in a row is initialized to 1 before drawing, and the actual number of lines in a row is computed as follow:

- starting from the class name, the procedure compares the length of the string (the class name) to the prefixed width of first column. If one line is not long enough to hold the string, new lines are added and carriage returns are inserted into the string accordingly until the string of the class name can be held completely in first column. A variable *cnhi* is used to store the minimal number of lines in a row needed for the class name;
- similar calculations are made for "Inherits Class". All the names of "inherits class" are concatenated together into one string, and commas are inserted between any two names. The length of this new string is compared to the width of the second column. If one line is not long enough to hold the string, new lines are added, and carriage returns are inserted into the string accordingly, until the string can be held completely in the second column. The variable used for holding the minimal number of lines needed in a row is *cihi*;
- similar computations are made for "Uses Class" and "Description" of the class, and the number of lines needed to hold the strings are stored in variables *cuhi* and *cdhi* respectively;
- then the four integers are compared to find the biggest one among them

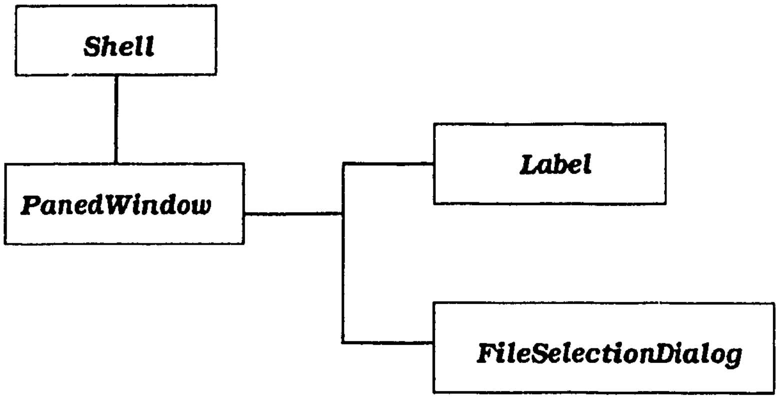


Figure 27: Structure of *CheckWindow*

which is to be the actual number of lines used in drawing this row.

Finally, four rectangles and four strings are drawn at their calculated positions.

Callback *clsTableCb* of View in the class selection dialog

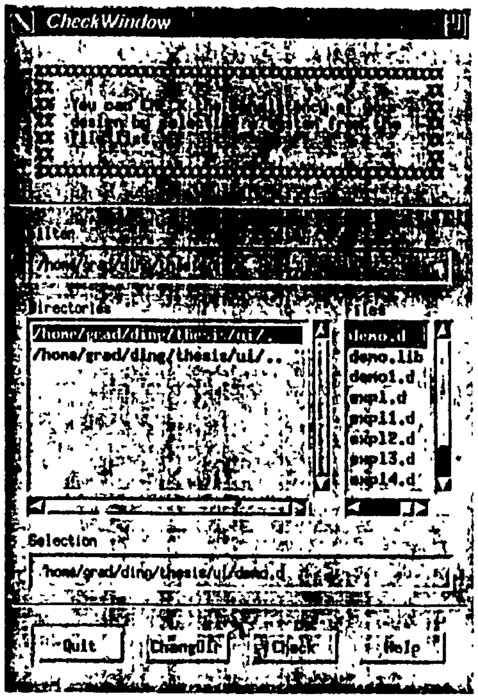
The callback *clsTableCb* is invoked when the user presses the View button in the selection dialog popped up by choosing the item Class in pulldown menu TableView. This routine is responsible for filling the message fields of the window and draw a class-level table on the drawing area by calling function *drawClassTable*. The name of the class whose class-level table is to be drawn is displayed on the top message field, and the system name of a design to which the class belongs is displayed in the message field on the bottom of the window.

The function *drawClassTable* is similar to the function *drawSystemTable*, except that there are five instead of four columns in the class-level table. These columns are labeled **Method Name, Return Type, Parameter, Uses Methods** and **Description** (comments for the method).

CheckWindow

The structure of the widgets in the *CheckWindow* is given in Figure 27, and Figure 28 shows the layout of the window.

This window is popped up when the button Check is pressed in the *ToolMainWindow*. From the layout of the window (Figure 28), we can see the body of this



You can [illegible]
defined by [illegible]
[illegible]

/home/grad/ding/thesis/ui/.
/home/grad/ding/thesis/ui/..

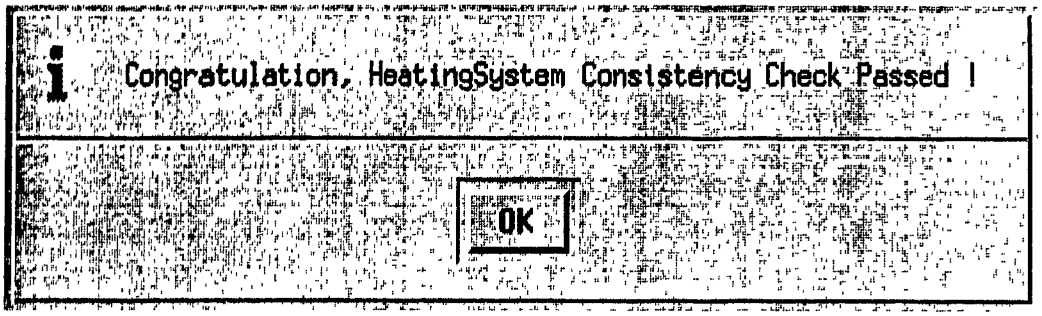
demo.d
demo.lib
demo1.d
expl.d
expl1.d
expl2.d
expl3.d
expl4.d

/home/grad/ding/thesis/ui/demo.d

Figure 28: *CheckWindow*

window is composed of three parts: on the top there is a message part which is used for displaying information about this window; in the middle there is a file selection part where a list of files with extension `.d` and `.lib` are displayed and a selection field is provided under the file list; on the bottom there is an action part equipped with four buttons. The message part is created with a *Label* widget, and the text for display is loaded during the creation of the Label. The file selection part and the action part are created together by calling a convenient Motif function for creating file selection dialog. Function *mySearchProc* is used for file searching, and some changes in button labels are made. The label of the button *OK* is changed to **`Check`**, and **`Cancel`** button is set to the left corner of the action part, and **`Help`** button is made inaccessible since no further help text is needed for the current version. As usual, the callback for `Cancel` is *popdownWidget*. The callback for button **`Check`** is the function *doCheckCb*.

The routine *doCheckCb* is responsible for obtaining the file name from the selection



Congratulation, HeatingSystem Consistency Check Passed !

Figure 29: The information dialog which is popped up when the checking succeeds

field in the *CheckWindow*, then passing the file name to a function named *checkFunc*. The function *checkFunc* calls the function *ast* to parse the given file. If parsing fails, a corresponding information dialog is popped up on the screen to inform the user, and the checking process is terminated. If the parsing succeeds, *checkFunc* assumes there is a standard library for the selected design. The AST for the library is generated inside the function *checkFunc* and the AST is stored together with the AST of the design in the global variable *aSystem*. The design selected by the user and the library are checked together as one unit by calling the function *doCheckStuff*.

The function *doCheckStuff* is a routine that prepares for the actual check by initializing the data structure for holding the undefined class and method names, calls the function *consistCheck* to check the consistency and completeness of the selected design, and finally gives the check result on a popup window. If the design passes the consistency check, an information dialog is popped up to inform the user of the success of the checking (an example of this is given in Figure 29). If the consistency check of the selected design fails, *doCheckStuff* is responsible for outputting the checking result in a popup window. The unsuccessful checking result is a list of class names and method names which are not defined but are used in the design. The list of class and method names is generated by the function *consistCheck*, and stored in a global variable *nonDefine* which is defined in file `nonDefineData.h`. The popup window is composed of three parts: a message part on the top of the window which is built with a *Label* widget and is used for displaying the message of "Consistency

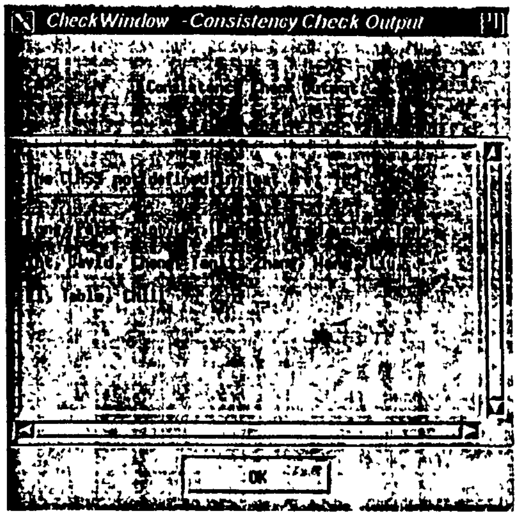


Figure 30: The output window which is popped up when the checking fails

Check Output"; a *Text* in the middle of the window which is used to display the list of undefined names; and on the bottom, there is an action part with a button `OK` in it. The callback routine for `OK` is the function *popdownWidget* which is responsible for removing the result window from the screen. The function *consistCheck* is defined in file `consistCheck.c`.

Inside the file `consistCheck.c`, a function *initNonDefine* is also defined which simply sets the variable *nonDefine* to NULL. This function is always called before checking to prepare the data structure for holding the new undefined names which are to be generated during the coming checking.

As we have seen, the global variable *aSystem* defined in file `tempData.h` is used to hold the AST of a design. The standard library of the design is stored in the place pointed by the `next` field of the variable *aSystem*. The two systems (the design and the library) are to be used together in consistency checking. The function *consistCheck* checks the following parts of a class C in the system of a design:

- each class that C inherits must be defined in the design or in the library of the design;

- each class used by C must be defined in the design or in the library;
- each type used in the variable declarations of C must be defined in the design or in the library;
- the return type as well as each parameter type in a method protocol in the class C must be defined in the design or in the library; and
- for each "method::name" pair inside a method of the class C, the "method" must be defined in the class "name", and obviously the class "name" must be defined in the design or in the library.

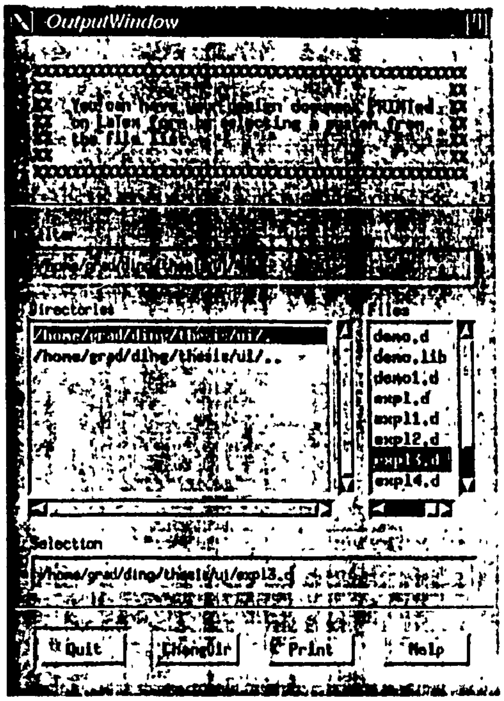
The undefined class names and "method::name" pairs are stored in a global variable *nonDefine* and duplicated names and pairs are eliminated from *nonDefine*. As an example, the part of code for checking the inherited class name is given in Figure 33 at the end of this Chapter.

OutputWindow

The structure of the widgets in this window is the same as that of the *CheckWindow*, and is shown in Figure 26, and Figure 31 shows the layout of the window.

This window is popped up when the button `Output` in the *ToolMainWindow* is selected by the user. The layout of this window is very similar to that of the *CheckWindow* except that the message on the top of the window is different and the label of the button *OK* is `Print` instead of `Check`. The procedure of creating *OutoutWindow* is exactly the same as that of creating *CheckWindow*. The callback routine for the button `Print` is the function *printCb* defined in file `print.c`.

The purpose of *printCb* is to generate for the user a file in LaTeX form which is the printable report of a design. The routine *printCb* gets the file name from the selection field of *OutputWindow*, then passes the file name to the function *ast* for parsing. If the parsing fails, an information dialog is popped up to inform the user and *printCb* returns. Otherwise, the output process moves on to get an output file name which is the same as the name of the selected file containing source text of the design but with extension `.tex`. The file name obtained is passed to the function *thereIs*, which



You can have your design document PRINTed by LaTex form by selecting a [illegible] from the file list [illegible]

/home/grad/ding/thesis/ui/.
/home/grad/ding/thesis/ui/..

/home/grad/ding/thesis/ui/expl3.d

Figure 31: *OutputWindow*

looks for a file of the same name in the current working directory. If such a file exists, it is deleted. In either case, the function ensures that an empty file with the given name is available for writing. The following contents are written to the file:

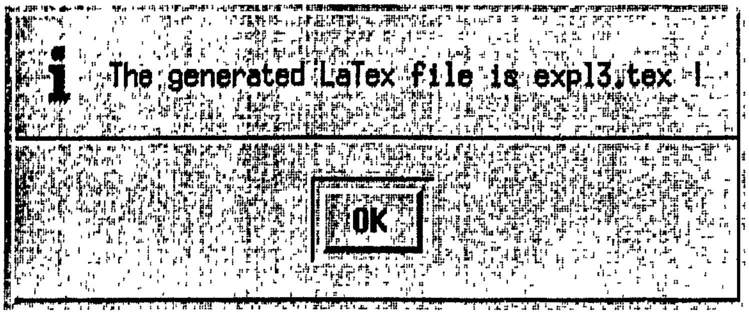
- The function *printCb* first writes to the file a text which is the header of a LaTeX file and contains the information of textwidth, textheight and topmargin, and so on. The text is stored in a local variable *textBegin*.

- Then *printCb* writes to the file the name of the system of the design and the description of the system, in LaTeX form.

- After that, it writes to the file in LaTeX form the information about each class in the system of the design. The information written includes:

  ▷ the class name, its description;

  ▷ the list of names of inherited class;



The generated LaTex file is expl3.tex !

Figure 32: The information dialog which is popped up to indicate the generated file

  - ▷ the list of class names used by the class;
  - ▷ variables defined in the class, and
  - ▷ the information about each method defined in the class, i.e. the method protocol and its description.

- Finally, *printCb* writes to the file a text stored in a local variable *textEnd* which is the end part of a LaTeX file containing "\end{*document*}".

After all the content has been written to the file, *printCb* closes the file and pops up an information dialog to inform the user the generated file. One such example is given in Figure 32.

4.3.2 Miscellaneous Functions

The functions described here are routines called miscellaneous functions which are frequently used for the creations of different windows, and are grouped into a file named `miscelFunc.c`.

popdownWidget

The most frequently used routine is the *popdownWidget* which simply calls **XToolkit** library *XtPopdown* to pop down the given widget.

isGood

The purpose of the function *isGood* is to check if the given file has extension `.d` (files containing the source texts of the designs) or `.lib` (the file containing the source text of the standard library). It obtains the extension of the given file (i.e. the string after the first dot in the file name), and compares the string to `d` and `lib`. If a match is found, an integer value 1 is returned to the calling routine, otherwise 0 is returned.

mySearchProc

The funtion *mySearchProc* is the new file searching procedure used during the creation of all the file selection dialogs in the tool. This function obtains the path of the current working directory from the *search_data* (provided by Motif) of the file selection dialog, reads all the files in the directory to a buffer, then calls the function *isGood* to select only those files with extension `.d` or `.lib` for display.

bigger

The function *bigger* compares two given integers and returns the bigger one to the calling routine.

rmRnDash

The purpose of the this function is to remove the carriage returns ("\n") and double dashes ("`--`") in the given string. It goes through each character in the given string, and whenever a carriage return or a pair of double dashes is found, it simply removes them.

addRetn

This routine is used frequently in drawing tables of a design. The purpose of *addRetn* is to split the given string into multi-line strings according to some calculation, then store the resulted strings in a variable *multiStr* which is an array of the type `TMultiStr`. Meanwhile, the number of lines needed for holding this string in a row is calculated accordingly and stored in a variable *norow*. The number of characters that will fit on a line is obtained by dividing the length of the line by the width of each character. The length of the line, in pixels, is known, and the Design Tool uses a font in which each character is 6 pixels wide. The given string, which may be a long string and has no carriage returns, is split into multiple lines and each split one-line string and its length are stored in the variable *multiStr*. In case one word is split into two lines, a dash ("-") is inserted at the end of the first part of the word. Finally, the split string and the number of lines it takes are returned to the calling routine by the variables *multiStr* and *norow*.

thereIs

The purpose of the function *thereIs* is to check if there is a file existing in the current working directory which matches the given file name. The function obtains the path of the current working directory by calling a UNIX system call[10] *getwd*, and reads all the files in the directory to a buffer, then compares each file name with the given string until a match is found (1 is returned) or the end of the file list is reached (0 is returned).

createInfoDialog

Information dialogs are used very often in the tool which are popped up to inform the user a specific status of a process, such as when a design has been checked successfully. As mentioned in the previous subsection, all the information dialogs are created by calling this *createInfoDialog*. This function calls the convenient Motif function to create a information dialog as a child of the given widget. During the creation, the *OK* button and *Help* button are not managed and therefore disappear from the action field of the dialog. The label of the *Cancel* button is changed to OK and the callback routine for this button is *popdownWidget* which is responsible for removing the dialog from the screen when OK is pressed. Finally, the shell of the created dialog is passed back to the calling routine.

writetofile

This function is used by *printCb* during the generation of the LaTeX file for a design. It simply call the C library *fwrite* to write a given string to a opened file.

checkspechar

This function is used frequently by *printCb* during the generation of the LaTeX file for a design. The purpose of this file is to preprocess all the special characters in the given string. These characters, which need special handling in LaTeX are "_", "%", "&", "$", "#", "{", "}", "~", "^" and "\". If any of the first seven characters is found, a backslash is inserted before it. Otherwise, a special handling is needed, such as for "\", a string "$\backslash$" is used to replace it, for "~", a string "\~{ }" is used instead, and for "^", a string "\^{ }" is used for replacing. Finally, the new string is returned to the calling routine.

isExist

The purpose of this function is to check if the given name already exists in the given list. This function is used by *consistCheck* to eliminate the duplicated names found in the variable *nonDefine* which is used to hold the undefined class and method names. The "name" is a string, and the "list" is a pointer to the type `TStringDict`. If the "name" is found in the "list", 1 is returned, otherwise, 0 is returned.

```
   /* check inherits class names */
inhs = currCls->inherits;
while (inhs) {
    tmpSys = aSystem;
    while (tmpSys) {
       otherCls = tmpSys->classList;
       name = (char *)strdup(inhs->name);
       found = FALSE;
       while (otherCls) {
           if (strcmp(name, otherCls->label.name) == 0) {
              found = TRUE;
              break;
            }
           otherCls = otherCls->next;
         }
       if (found)
          break;
       tmpSys = tmpSys->next;
     }
    if (!found) { /* the name not defined */
       if (!isExist(name, nonDefine)) {
          tmp = (TStringDict *)malloc(sizeof(TStringDict));
          tmp->name = (char *)strdup(name);
          tmp->next = nonDefine;
          nonDefine = tmp;
        }
      }
    inhs = inhs->next;
  }
```

Figure 33: Code for checking inherited class name

Chapter 5

A Walk-Through of The Design Tool

In this chapter, we describe a walk-through of the Design Tool with the example which we described in Section 3.3.1, the Heating System. Therefore the tables in the View Window and the text in the Edit Window can be used to compare with the tables and text which we developed in Section 3.3. Major functions of the tool are described with corresponding pictures of the tool we dumped from X window environment. An example of a printable report which is generated by the Design Tool is also given at the end of this chapter.

Common Features of Windows and Dialog Boxes

The following are features common to all the windows and dialog boxes except the main window of the Design Tool (with the exception of the first one):

- Every window can be resized by placing the mouse pointer on the small square which is on the right top of the window frame, pressing the left button of the mouse, and dragging the mouse to the desired size then releasing the mouse button (The *ToolMain Window* can be resized in the same way).

- The selection of an item in every selection dialog box (including file selection dialog and general selection dialog) can be made either by mouse or by keykoard input. Mouse selection is done by placing the mouse pointer to the desired name in the list provided by the dialog, clicking the left button of the mouse, and then using the mouse to click on the action button, such as `View` or `Open` and so on, to finish the action of selection. The keyboard input can be done by typing the desired name on the selection field which is usually on the bottom part of the dialog, then pressing the *Return* key to invoke the selection action.

- The newly popped up window is always appears on the top of the screen. When a window is popped up on the screen, it always appears on the place where it is removed (popped down) last time from the screen, except on the first time that it is created, at what time it appears on the left up corner of the screen.

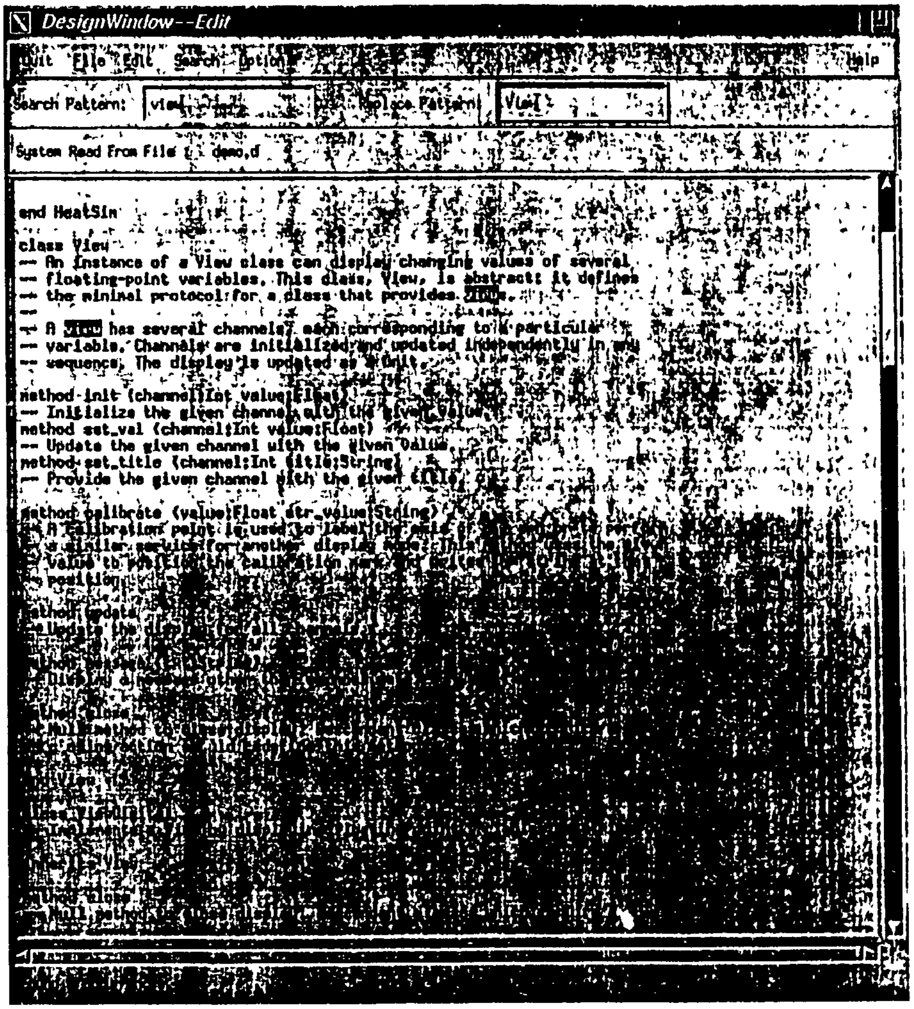
- There is at most one window of each type exists in the tool, and when the user presses some button to invoke the popping up of the corresponding window, the popped up window will appear on the top of the screen therefore it can be seen as a whole by the user, no matter whether it has been created and hidden by other windows, or it has been removed from the screen after it has been created, or it has not been created.

ToolMainWindow

The tool is activated by typing the command `oodtool` inside the X window or the Motif window environment, and the main window of the tool, the *ToolMainWindow* (Figure 12), is popped up on the top of the screen. The *ToolMainWindow* is the only place to get access to the help, in the *HelpWindow* which provides the on-line help to the tool, and the working windows which provide the four major functions, i.e. *editing, viewing, checking* and *outputting.* It is also the only place to properly exit from the Design Tool. The *HelpWindow*, which is shown in Figure 15, is invoked by pressing the `Help` button on the right corner of the action field of the main window. It can be dismissed by selecting the `OK` button in the window, and can be retrieved later by selecting the `Help` button in the *ToolMainWindow* again. If we want to exit the tool, the `Quit` button should be pressed, and when a warning dialog (Figure 13) is popped up, the `Yes` button in the dialog should be selected. Once we select the button `Yes` in the warning dialog, all the windows of the tool are deleted from the screen. Otherwise, if the button `No` is selected, only the warning dialog is removed from the screen, and everything else in the tool are left unchanged.

DesignWindow-Edit

Suppose we want to activate the editing function to view the source text of a design or to modify a design, the button `Design` in the *ToolMainWindow* should be selected by a mouse clicking on the button. As the result, the *DesignWindow* (Figure 17) is popped up on the top of the screen. We can quit the *DesignWindow* by selecting the `Quit` button in the *DesignWindow*, and can retrieve it by selecting the `Design` button again. The access to the Edit Window of the Design Tool can be achieved by selecting the `Edit` button in the *DesignWindow*. The Edit Window, named *DesignWindow-Edit*, is popped up on the top of the screen with an empty editing area in the middle of the window (Figure 19). Exiting from this window can be done by selecting the `Quit` menu in the pull down menu `Quit` on the left corner of the menu bar. Suppose we want to edit the source text of the design of the Heating System, we can do so by selecting the `Open` menu in the second pull down menu `File` on the menu bar, and when a file selection dialog is popped up on the screen, selecting the file *demo.d* in the file list (Figure 20). As the result of such selection, the source text of the design of the Heating System is loaded into the editing area of the window and ready for viewing and editing. Meanwhile, the file name *demo.d* is displayed in the message field on the top of the editing area. We can scroll the scrollbars up or down, and right or left to view the different part of the text in the editing area, as well as resize the the whole window to make the editing area larger or smaller. For editing, we can make some selection of the text by mouse dragging, and the selected text is highlighted in the editing area. Then we can do *Cut*, *Copy* and *Paste* to the selected text by choosing the corresponding menu in the pull down menu `Edit` of the menu bar. The *Cut* action simply cuts the selected text, which is also highlighted, from the edting area. *Copy* will copy the selected text into a clipboard which is preserved by the tool and used as



Search Pattern: view

Replace Pattern: View

System Read From File : demo.d

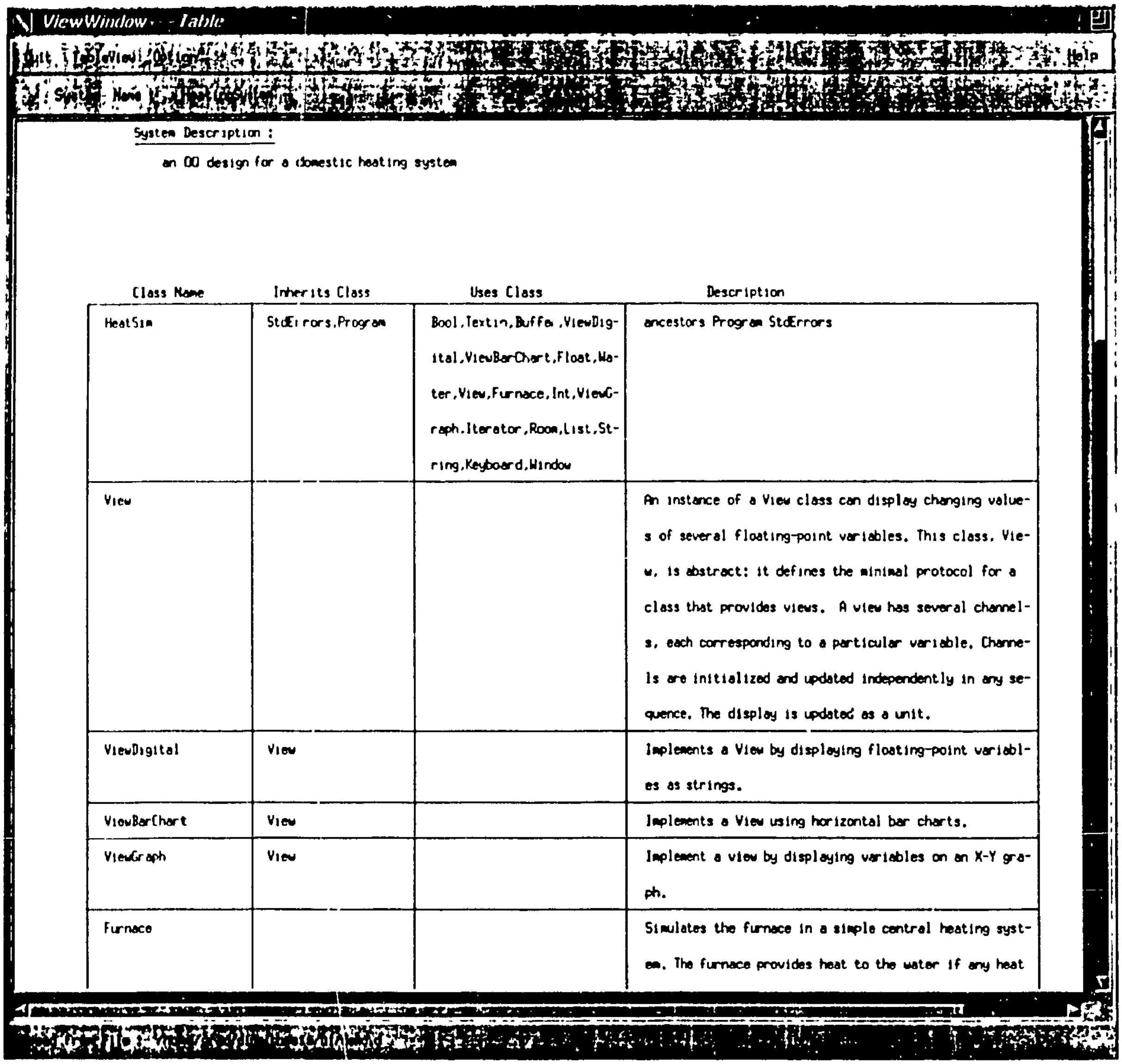
```
end HeatSin

class View
-- An instance of a View class can display changing values of several
-- floating-point variables. This class, View, is abstract: it defines
-- the minimal protocol for a class that provides views.
--
-- A view has several channels, each corresponding to a particular
-- variable. Channels are initialized and updated independently in any
-- sequence. The display is updated as a unit.

method init (channel:Int value:Float)
-- Initialize the given channel with the given value.
method set_val (channel:Int value:Float)
-- Update the given channel with the given value.
method set_title (channel:Int title:String)
-- Provide the given channel with the given title.

method calibrate (value:Float str_value:String)
-- A calibration point is used to label the axis of [illegible]
-- a similar service for another display [illegible]
-- value to position the calibration mark [illegible]
-- position [illegible]
method update
-- Update the display [illegible]
[illegible]
method close
[illegible]
```

Figure 34: *DesignWindow Edit* with Search invoked

a temporary buffer. The *Paste* action pastes the content in the clipboard to the place where the current cursor is. The menu `Clear` of the pull down menu `Edit` deletes the content of the clipboard. This window also allows the user to do searching and replacing on the text. Searching is done by input the search pattern on the left text field directly beneath the menu bar. If we want to search for the string *view* in the text, we can do so by typing the *view* in the searching place, and then selecting the menu item `Find Next` in the pull down menu `Search` to search the string one by one in the text, or selecting the `Show All` to get all the string *views* highlighted in the text (an example of this is shown in Figure 34). The number of occurrences which match the search pattern is displayed on the message field on the bottom of the window. The Replacing can be done by first typing the searching pattern, such as *view*, in the left text field and replace pattern, such as *View*, in the right text field directly beneath the menu bar, then selecting the menu item `Replace Φ Text`. As the result, all the string *views* will be replaced by the string *Views*. The highlights of text which are made during the selection can be removed by selecting the menu item `Clear` in the pull down menu `Search`. After some changes have been made to the source text of the design, we can save the changes in the design by selecting the `Save` menu in the pull down menu `File`, and after a file selection dialog is popped up on the screen, selecting the right file name by mouse click or typing the desired file name through the keyboard. Then the updated content in the editing area is saved to the file which we chose. If some changes are made to the design, and the changes are not saved to a file when we try to exit the *DesignWindow-Edit*, a warning dialog is popped up on the screen to remind that the changes have not been saved, and could be lost if we exit the window. Similar things happen when we make some modifications to the design, then try to open a new file before the modifications we made are saved to a file. Another function which does not belong to the scope of editing but presents the convenience of checking to the designers is also provided in this window. This is the function `ConsistCheck` in the pull down menu `Option`. Once we have loaded a file into the editing area, or when we have finished the updating of the design in the editing area, we can check the consistency of the design by selecting the menu `ConsistCheck`. There is one thing needs to be mentioned here: when the source text of a design has been changed in the editing area, only when we save the source text of the design into the same file which previously contains the old source text of the design, the consistency check will be done on the updated design, otherwise, the check is done on the old one. Therefore, the Design Tool provides protection by checking the editor to see if some modifications have been made, but have not been saved. If this happens, the tool pops up a warning dialog to remind us about this situation.

ViewWindow–Table

If we want to view the tables of a design, we can do so by selecting the `View` button in the *ToolMainWindow*, and when the *ViewWindow* (Figure 22) is popped up on the screen, choosing the `Table` button. Then the view table window, *ViewWindow Table*, is popped up on the screen with an empty and white drawing area in the middle of the window (Figure 24). Since the default window size is too small to view any particular tables, we can enlarge the window size by mouse to get a bigger drawing area. Suppose we want to view the tables of the design of the Heating System, we then need to do the selection by choosing the menu `System` in the pull down menu `TableView`, and when a file selection dialog is popped up on the screen, selecting the file name *demo.d* from the file list. As the result, the system-level table of the design of the Heating System is displayed in the drawing area of the window, with the system name of the design *HeatingSystem* displayed on the message field on the top, and the file name *demo.d* displayed on the message field on the bottom of the window. We can scroll the scrollbars up or down, and right or left to view different part of the table in the window. The complete system-level table of the Heating System is given in Figure 35 and Figure 36. If we want to view a class-level table of one of the classes in the design of the Heating System, we can do so by choosing the menu `Class` in the pull down menu `TableView`. A selection dialog is then popped up on the screen with a list of class names which belong to the design of the Heating System (Figure 25). Suppose we want to view the table for the class *View*, a selection of the name *View* in the class list of the selection dialog should be made by mouse or by keyoard input. The class-level table of the class *View* is then displayed in the drawing area of the window, with the class name *View* displayed on the message field

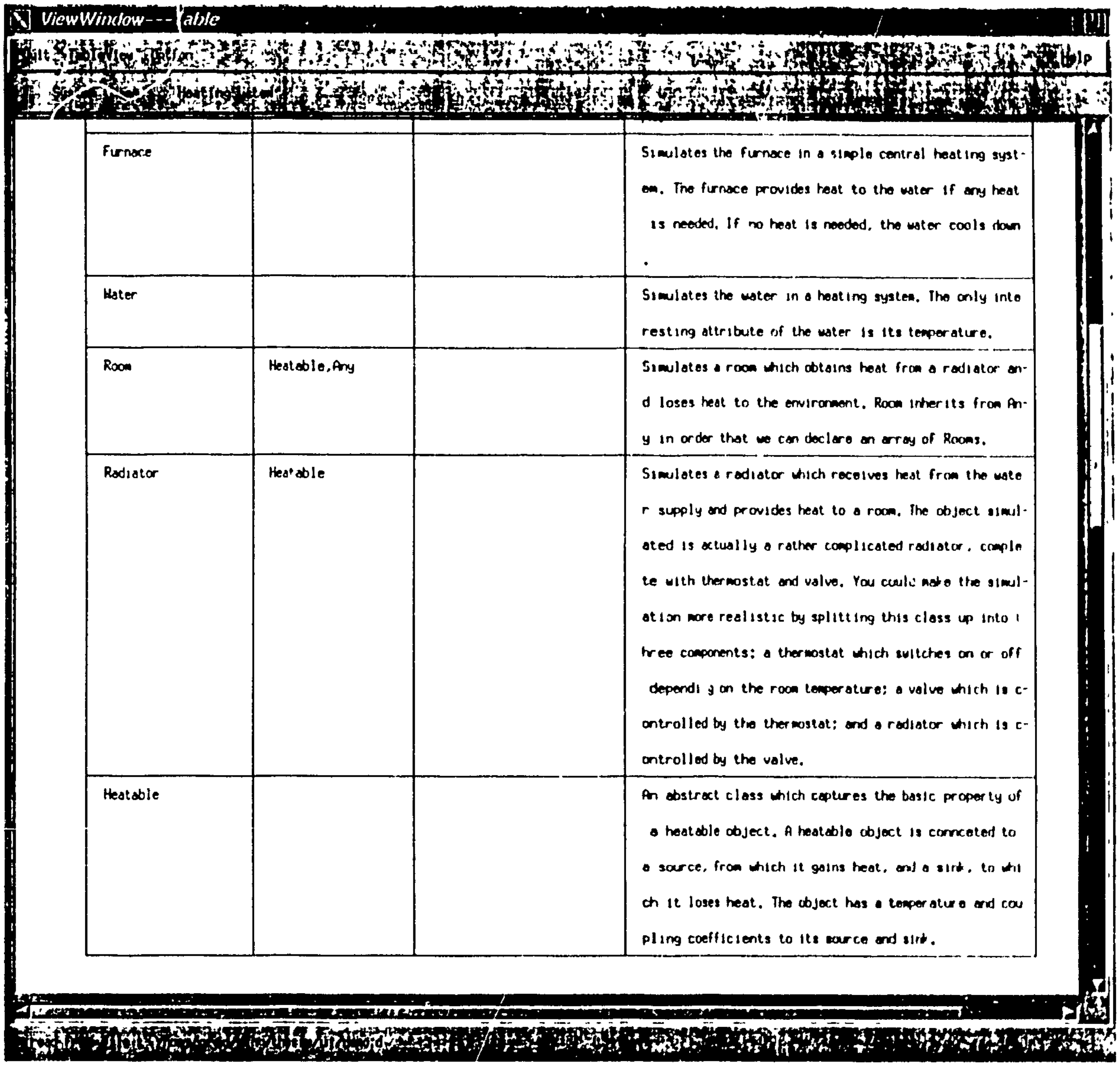


System Description :

an OO design for a domestic heating system

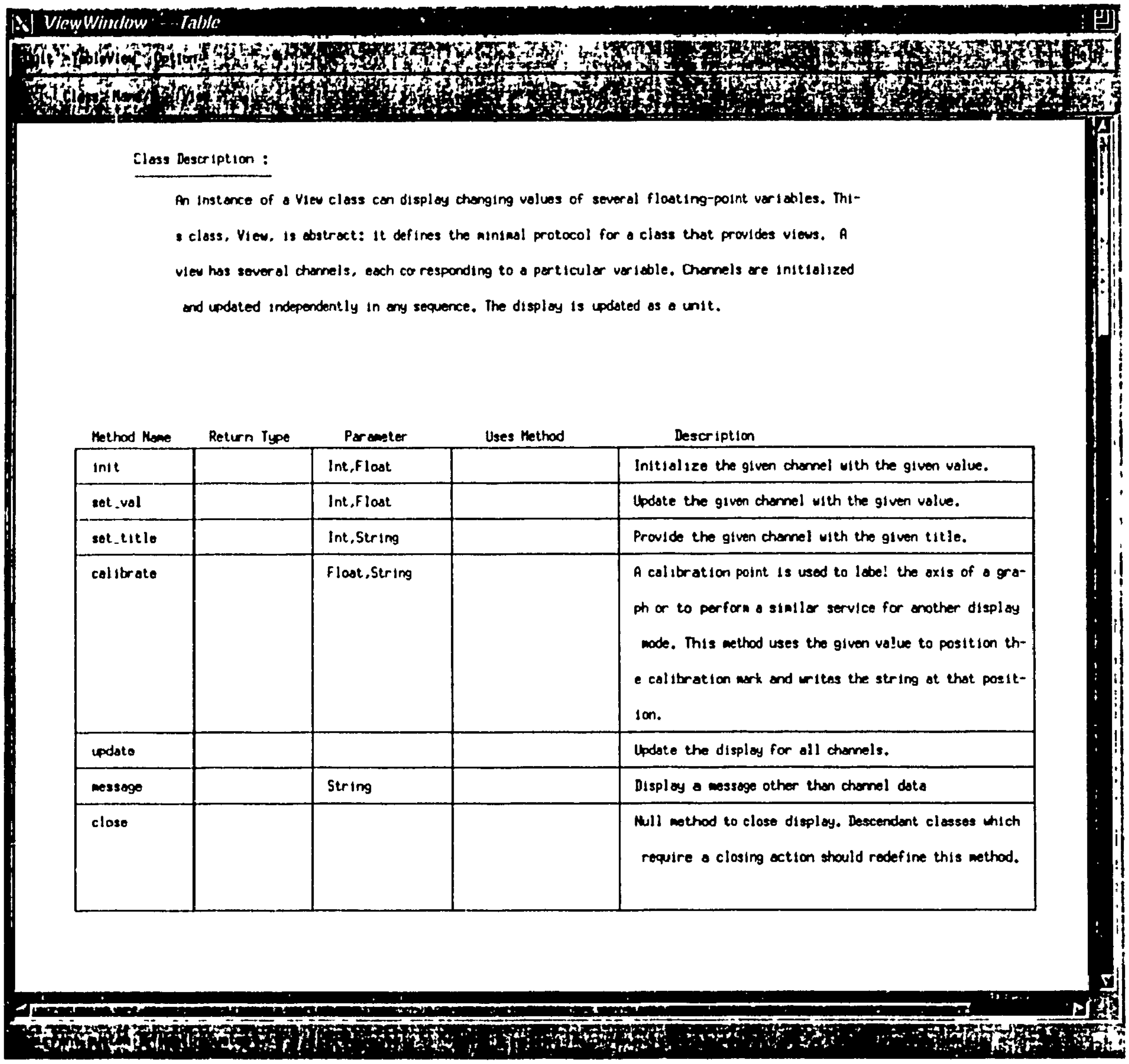
Class Name	Inherits Class	Uses Class	Description
HeatSim	StdErrors,Program	Bool,Textin,Buffer ,ViewDigital,ViewBarChart,Float,Water,View,Furnace,Int,ViewGraph.Iterator,Room,List,String,Keyboard,Window	ancestors Program StdErrors
View			An instance of a View class can display changing values of several floating-point variables. This class, View, is abstract; it defines the minimal protocol for a class that provides views. A view has several channels, each corresponding to a particular variable. Channels are initialized and updated independently in any sequence. The display is updated as a unit.
ViewDigital	View		Implements a View by displaying floating-point variables as strings.
ViewBarChart	View		Implements a View using horizontal bar charts.
ViewGraph	View		Implement a view by displaying variables on an X-Y graph.
Furnace			Simulates the furnace in a simple central heating system. The furnace provides heat to the water if any heat

Figure 35: The system-level table of the Heating System



Furnace			Simulates the furnace in a simple central heating system. The furnace provides heat to the water if any heat is needed. If no heat is needed, the water cools down .
Water			Simulates the water in a heating system. The only interesting attribute of the water is its temperature.
Room	Heatable,Any		Simulates a room which obtains heat from a radiator and loses heat to the environment. Room inherits from Any in order that we can declare an array of Rooms.
Radiator	Heatable		Simulates a radiator which receives heat from the water supply and provides heat to a room. The object simulated is actually a rather complicated radiator , complete with thermostat and valve. You could make the simulation more realistic by splitting this class up into three components; a thermostat which switches on or off dependi g on the room temperature; a valve which is controlled by the thermostat; and a radiator which is controlled by the valve.
Heatable			An abstract class which captures the basic property of a heatable object. A heatable object is connceted to a source, from which it gains heat, and a sink, to which it loses heat. The object has a temperature and coupling coefficients to its source and sink.

Figure 36: The system-level table of the Heating System (*continued*)

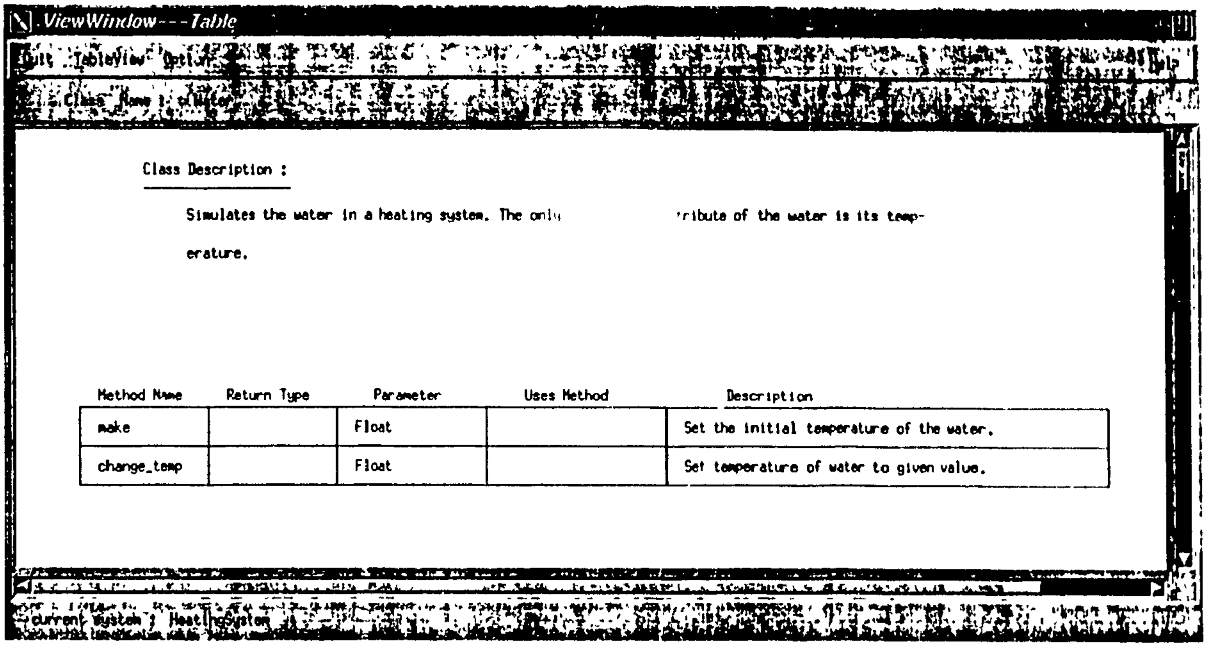


Class Description :

An instance of a View class can display changing values of several floating-point variables. This class, View, is abstract: it defines the minimal protocol for a class that provides views. A view has several channels, each corresponding to a particular variable. Channels are initialized and updated independently in any sequence. The display is updated as a unit.

Method Name	Return Type	Parameter	Uses Method	Description
init		Int,Float		Initialize the given channel with the given value.
set_val		Int,Float		Update the given channel with the given value.
set_title		Int,String		Provide the given channel with the given title.
calibrate		Float,String		A calibration point is used to label the axis of a graph or to perform a similar service for another display mode. This method uses the given value to position the calibration mark and writes the string at that position.
update				Update the display for all channels.
message		String		Display a message other than channel data
close				Null method to close display. Descendant classes which require a closing action should redefine this method.

Figure 37: The class-level table for the class *View*



Class Description :

Simulates the water in a heating system. The only 'ribute of the water is its temp-erature.

Method Name	Return Type	Parameter	Uses Method	Description
make		Float		Set the initial temperature of the water.
change_temp		Float		Set temperature of water to given value.

Figure 38: The class-level table for the class *Water*

on the top of the drawing area and the system name of the design *HeatingSystem* displayed on the message field beneath the drawing area. This is shown in Figure 37. Another class-level table for the class *Water* is also given in Figure 38, and these two figures (37, 38) can be compared with Table 6 and 7 in Section 3.3.1. We can clear the drawing area by selecting the menu `ScreenClear` in the pull down menu `Option`. Exiting from this window can be done by selecting the menu `Quit` in the pull down menu `Quit`.

CheckWindow

As mentioned before, we can do the consistency check of a design either inside the *DesignWindow-Edit*, or through the check window. The *CheckWindow* (Figure 28) can be loaded by selecting the `Check` button in the *ToolMainWindow*. Suppose we want to check the consistency of the design of the Heating System, we need to select the file which contains the source text of the design, i.e. select the file *demo.d* in the file list. The check action is accomplished by mouse clicking on the button Check. On the accomplishment of the checking, the *CheckWindow* is removed from the screen, and the result of the check is given by either an information dialog for indicating the success of the checking, or an output window (Figure 30) for listing the undefined names of classes and methods in the design. For the Heating System, an information dialog (Figure 29) is popped up on the screen. The files displayed in the file list are based on the current working directory, and we can change to another directory by either typing the name of that directory on the selection field, or using mouse to click on the directory. One thing needs to be mentioned here is that all the file selection dialogs in the tool only display the files with extension .d or .lib, therefore, in some case, the file list part in the dialog may not contain any file names. The exiting of the *CheckWindow* can be done by selecting the button Quit.

OutoutWindow

The printable report of a design can be generated by invoking the output function of the tool. This is done by selecting the Output button in the *ToolMainWindow*. Then the *OutputWindow* (Figure 31) is popped up on the screen. Suppose we want to generate the printable report of the design of the Heating System, we begin by selecting the file *demo.d* in the file list, then mouse clicking on the button Print. The generated LaTeX file of the design is given the name *demo.tex* on the current working directory. Part of this generated file is shown in the following, and the report of the design, which is obtained by compiling the generated LaTeX file and converting it to the *PostScript* form, is shown next.

```
% The first command specifies the style (article) and base font size (12pt).
\documentstyle[12pt]{article}
% Page size and margins: set for 8.5in x 11in paper with 1in margins.
\textwidth    6.5in
\textheight     9in
\topmargin   -0.5in
\oddsidemargin  0in
```

```
\evensidemargin 0in
\parskip        1ex
\parindent      0em

% Insert any additional macros etc here.
% Then start the 'document'.
\begin{document}

{\Large {\bf system }{\sl HeatingSystem}

{\leftskip 24pt \small
an OO design for a domestic heating system\par}

\vskip 30pt plus 12pt minus 6pt
{\large {\bf class }{\sl HeatSim}

{\leftskip 24pt \small \par}
{\leftskip 24pt \small
ancestors Program StdErrors\par}

\vskip 6pt plus 3pt minus 1pt
{\bf inherits}
{\sl StdErrors\/}
{\sl Program\/}

{\bf uses}
{\sl Bool\/}
{\sl Textin\/}
{\sl Buffer\/}
{\sl ViewDigital\/}
{\sl ViewBarChart\/}
```

```
{\sl Float\/}
{\sl Water\/}
{\sl View\/}
{\sl Furnace\/}
{\sl Int\/}
{\sl ViewGraph\/}
{\sl Iterator\/}
{\sl Room\/}
{\sl List\/}
{\sl String\/}
{\sl Keyboard\/}
{\sl Window\/}

\vspace{6pt}
{\bf method }{\sl entry\/}\par
\vskip 6pt plus 3pt minus 1pt

\vskip 36pt plus 12pt minus 6pt
{\large {\bf class }{\sl View}

{\leftskip 24pt \small \par}
{\leftskip 24pt \small
An instance of a View class can display changing values of
several floating-point variables. This class, View, is abstract:
it defines the minimal protocol for a class that provides views.
A view has several channels, each corresponding to a particular
variable. Channels are initialized and updated independently in
any sequence. The display is updated as a unit. \par}

\vskip 6pt plus 3pt minus 1pt
\vspace{6pt}
```

```
{\bf method }{\sl init\/}({\sl channel\/}\,:\,{\sl Int\/}, {\sl
value\/}\,:\,{\sl Float\/})\, \par
{\leftskip 24pt \small
Initialize the given channel with the given value. \par}

\vskip 6pt plus 3pt minus 1pt
{\bf method }{\sl set\_val\/}({\sl channel\/}\,:\,{\sl Int\/},
{\sl value\/}\,:\,{\sl Float\/})\, \par
{\leftskip 24pt \small
Update the given channel with the given value. \par}

\vskip 6pt plus 3pt minus 1pt
{\bf method }{\sl set\_title\/}({\sl channel\/}\,:\,{\sl Int\/},
{\sl title\/}\,:\,{\sl String\/})\, \par
{\leftskip 24pt \small
Provide the given channel with the given title. \par}

\vskip 6pt plus 3pt minus 1pt
{\bf method }{\sl calibrate\/}({\sl value\/}\,:\,{\sl Float\/},
{\sl str\_value\/}\,:\,{\sl String\/})\, \par
{\leftskip 24pt \small
A calibration point is used to label the axis of a graph or to
perform a similar service for another display mode. This method
uses the given value to position the calibration mark and writes
the string at that position. \par}

\vskip 6pt plus 3pt minus 1pt
{\bf method }{\sl update\/}\par
{\leftskip 24pt \small
Update the display for all channels. \par}
{\bf inherits}
```

```
{\sl View\/}

\vskip 6pt plus 3pt minus 1pt
{\bf method }{\sl message\/}({\sl txt\/}\,:\,{\sl String\/})\, \par
{\leftskip 24pt \small
Display a message other than channel data \par}

\vskip 6pt plus 3pt minus 1pt
{\bf method }{\sl close\/}\par
{\leftskip 24pt \small
Null method to close display. Descendant classes which require a
closing action should redefine this method. \par}

\vskip 36pt plus 12pt minus 6pt
% Finally, end the file with:
\end{document}
```

The following are the actual printable report of the part of the design, the Heating System, obtained by compiling the above LaTeX file.

system *HeatingSystem*

an OO design for a domestic heating system

class *HeatSim*

ancestors Program StdErrors

inherits *StdErrors Program*

uses *Bool Textin Buffer ViewDigital ViewBarChart Float Water View Furnace Int ViewGraph Iterator Room List String Keyboard Window*

method *entry*

class *View*

An instance of a View class can display changing values of several floating-point variables. This class, View, is abstract: it defines the minimal protocol for a class that provides views. A view has several channels, each corresponding to a particular variable. Channels are initialized and updated independently in any sequence. The display is updated as a unit.

method *init(channel: Int, value: Float)*

Initialize the given channel with the given value.

method *set_val(channel: Int, value: Float)*

Update the given channel with the given value.

method *set_title(channel: Int, title: String)*

Provide the given channel with the given title.

method *calibrate(value: Float, str_value: String)*

A calibration point is used to label the axis of a graph or to perform a similar service for another display mode. This method uses the given value to position the calibration mark and writes the string at that position.

method *update*

Update the display for all channels.

inherits *View*

method *message(txt: String)*

Display a message other than channel data

method *close*

Null method to close display. Descendant classes which require a closing action should redefine this method.

Chapter 6

Conclusion

In this thesis, we presented a supporting tool for object-oriented design and system evolution. As design is the most important and effort consuming phase in object-oriented software development, our tool aims at providing automated support to empower the designers as much as possible during the design process. Maintenance is the most costly phase in system evolution, our tool provides direct assistances to the maintainers during system evolution by supporting automation in capturing and altering the design. Important features of the tool are editing, viewing, consistency checking and the generating of printable report of the design.

The design and implementation of the tool were presented in the thesis. Design format which forms the theoretical basis for the tool was discussed; design display in textual and tabular forms were exploited and illustrated with an example; the abstract syntax tree(AST) which is the base of data in the tool were generated, and a parser module for constructing the AST of a design were presented; functional modules of editor, viewer, checker and printer were built based on the analysis of the tool and theoretical results.

A carefully defined, consistently formatted user interface was developed as part of the tool, and as the media for human interaction with the function modules of the tool. We considered the interface part to be the significant and necessary part of the tool design, and emphasis was put on it to achieve the ease of the use of the tool.

As the first version of the design tool, limitation of the functionality of the tool is unavoidable, and several directions for future work can be suggested to contribute to the improvement of the tool. The following are some suggestions for future work:

- Some new functions can be added to the tool to enlarge the tool's functionality. This may include the ability to provide graphical display (diagram) of the system of a design and a class in the system; the ability to respond to the user's queries, entered by means of menu and simple dialog boxes, some typical queries are: which classes use service C::S? Which services does class C use? If I change C::S, what classes will be affected? etc.; the ability to provide a faster and more convenient way than on-line editing to modify the design, and once a change has been made to a design, the user would soon be informed in a proper way.

- More advanced features can been added to the tool. For example, the tool could suggest a sequence for implementing the design, and could propose test plans for each stage of the implementation.

- The future version of the tool can take into account the dynamic model of a system as a part of the design of the system. This could be done by portraying the dynamic behaviors and causality flows of the system through the notations of *use cases* and *timethreads*.

- The tool could be added to with a *system interface*, therefore it could be incorporated with other automated tools, like a code generator, to form a unified development and maintenance environment for object-oriented software.

- The tool could be extended with generators for various target languages. For example, it could generate class specifications in C++ or Eiffel, leaving only the bodies of the methods to be completed by the implementors.

- An experiment could be conducted to compare this tool with other tools.

Some people, including myself, like the implementation phase of software development. Implementation is enjoyable because progress is visible during the process. Our tool makes the design enjoyable by allowing the designer to see progress immediately and to monitor it. In this way, the tool supports and encourage design evolution.

References

[1] Mehmet Aksit and Lodewijk Bergmans. Obstacles in object-oriented software development. *OOPSLA '92*, pages 341-358, 1992.

[2] Robert S Arnold, Malcolm Slovin, and Norman Wilde. Do design records really benefit software maintenance? *IEEE Conference on Software Maintenance*, pages 234-243, 1993.

[3] Nabajyoti Barkakati. *X Window System Programming*. SAMS, 1991.

[4] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *OOPSLA 89 Proceedings*, pages 1-6, 1989.

[5] Grady Booch. Object-oriented development. *IEEE Trans. Software Engineering*, SE-12(2):211-221, February 1986.

[6] Grady Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.

[7] R. J. A. Buhr and R. S. Casselman. Designing with timethreads. *SCE-93-05, Department of System & Computer Engineering, Carleton Univ., Ottawa Canada*, 1993.

[8] Raymond J. A. Buhr and Ronald S. Casselman. Architectures with pictures. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages an*, pages 466-483, 1992.

[9] Derek Coleman, Fiona Hayes, and Stephen Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Trans. Software Engineering*, SE–18(1):9–18, January 1992.

[10] David A. Curry. *Using C on the UNIX system*. O'Reilly & Associates, Inc, 1989.

[11] Dennis de Champeaux, Douglas Lea, and Penelope Faure. The process of object-oriented design. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages an*, pages 45–62, 1992.

[12] Peter Grogono. Designing for change. *Department of Computer Science, Concordia University, Montreal, Canada*, 1994.

[13] Wayne Haythorn. What is object-oriented design? *J. Object-Oriented Programming*, 7(1):67–78, March-April 1994.

[14] Dan Heller and Paula M. Ferguson. *Motif Programming Manual*. O'Reilly & Associates, Inc, 1994.

[15] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[16] Wilf LaLonde, John Pugh, Paul White, and Jean-Pierre Corriveau. Towards unifying analysis, design, and implementation in object-oriented environments. *The center for Object-Oriented Programming, School of Computer Science, Carleton University, Ottawa, Canada*, pages 563–569, 1993.

[17] Tony Mason and Doug Brown. *Lex & Yacc*. O'Reilly & Associates, Inc, 1990.

[18] Deborah J. Mayhew. *Principles and Guidelines in Software User Interface Design*. Prentice Hall, 1992.

[19] Bertrand Meyer. Reusability: The case for object-oriented design. *IEEE Software*, pages 50–64, 1987.

[20] David E. Monarchi and Gretchen I. Puhr. A research typology for object-oriented analysis and design. *Comm. ACM*, 35(9):35–47, October 1992.

[21] K. Nygaard and O-J. Dahl. The development of the SIMULA language. In R. Wexelblat, editor, *History of Programming Languages*, pages 439–493. Academic Press, 1981.

[22] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, December 1972.

[23] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[24] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interation.* Addison-Wesley Publishing Company, 1987.

[25] Anthony I. Wasserman, Peter A. Pircher, and Robert J. Muller. The object-oriented structured design notation for software design representation. *IEEE Computer*, pages 50-63, March 1990.

[26] Rebecca Wirfs-Brock and Ralph E. Johnson. Surveying current research in object oriented design. *Comm. ACM*, 33(9):104–124, September 1990.

[27] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, 1990.

Appendix A Class Specifications of the Heating System

class	HeatSim
superclass	none
subclass	none
description	the main class used to invoke the simulation
contract	1. entry
description	one method for starting main loop

class	Furnace
superclass	none
subclass	none
description	Simulates the furnace in a simple central heating system. The furnace provides heat to the water if any heat is needed. Simulates the furnace in a simple central heating system.
contract	1. make(furnace_power:Float sys_water:Water max_water_temp:Float)
description	Construct a furnace object, given the water it has to heat, and the maximum temperature of the water.
contract	2. update(heat_needed:Bool env_temp:Float)
description	Use the furnace to increase the water temperature if heat is needed and the water is not above its maximum temperature. The water loses a little heat to the environment.

class	Room
superclass	Heatable, Any
subclass	none
description	Simulates a room which obtains heat from a radiator and loses heat to the environment. Room inherits from Any in order that we can declare an array of Rooms.
contract	1. same(other:Room): Bool
description	Return true if self and other refer to the same object.
contract	2. show : String
description	Return a string representation of the receiver.
contract	3. change_temp(source_temp:Float sink_temp:Float)
description	Change the temperature of the heatable object given the temperatures of the associated source and sink. The new temperature will depend on the relative temperatures of the object, the source, and the sink as well as the thermal coupling coefficients between the objects.
contract	4. make (room_name:String room_temp:Float room_from_rad:Float room_to_env:Float rad_temp:Float rad_from_water:Float rad_to_room:Float rad_setting:Float rad_tol:Float sys_water:Water)
description	Constructing a room requires that several important parameters be set. room_from_rad is the thermal coupling for heat flowing from the radiator to the room. room_to_env, rad_from_water, and rad_to_room have similar meanings. The rad_setting is the temperature at which the radiator comes on, but the actual transitions are determined by rad_setting +/- rad_tol.
contract	5. update (env_temp:Float)
description	Update a room given the temperature of the environment.
contract	6. rad_on : Bool
description	Return true if the radiator belonging to this room is on.

class	View
superclass	none
subclass	ViewDigital, ViewBarChart, ViewGraph
description	An instance of a View class can display changing values of several floating-point variables. This class, View, is abstract: it defines the minimal protocol for a class that provides views. A view has several channels, each corresponding to a particular variable. Channels are initialized and updated independently in any sequence. The display is updated as a unit.
contract	1. init(channel:Int value:Float)
description	Initialize the given channel with the given value.
contract	2. set_val(channel:Int value:Float)
description	Update the given channel with the given value.
contract	3. set_title(channel:Int title:String)
description	Provide the given channel with the given title.
contract	4. calibrate(value:Float str_value:String)
description	A calibration point is used to label the axis of a graph or to perform a similar service for another display mode. This method uses the given value to position the calibration mark and writes the string at that position.
contract	5. update
description	Update the display for all channels.
contract	6. message(txt:String)
description	Display a message other than channel data .
contract	7. close
description	Null method to close display. Descendant classes which require a closing action should redefine this method.

class	Water
superclass	none
subclass	none
description	Simulates the water in a heating system. The only interesting attribute of the water is its temperature.
contract	1. make (water_temp:Float)
description	Set the initial temperature of the water.
contract	2. change_temp(new_temp:Float)
description	Set temperature of water to given value.

Appendix B Abstract Syntax Tree

The AST is declared in C as a union of "**struct**" with each "struct" corresponding to a non-terminal of the abstract syntax of the design input language.

```
#ifndef TOOLSTRUCT_H
#define TOOLSTRUCT_H

#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#define NIL   NULL
#define TRUE     1
#define FALSE    0
#define NOFILENAME   0
#define CANNOTOPEN   1
#define PARSEFAIL    2
#define PARSESUCCEED 3
typedef char* TString;

  /* data structure used for holding multi-line string in display */
typedef  struct TMultiStr {
         char  *str;
         int   len;
 } TMultiStr;
```

```
typedef enum
{ aSYSTEM, aCLASS, aMETHOD, aVAR, aUSES, anINHERITS } TAttrType;

typedef struct TLabel {
     TString   name;
     TString   comment;
   } TLabel;

typedef struct TStringDict {
     TString            name;
     struct TStringDict *next;
   } TStringDict;

typedef struct TNTPairDict {
     TString       name;
     TString       type;
     struct TNTPairDict *next;
   } TNTPairDict;

typedef struct TTOPair {
     TString   type;
     TString   origin;  /* indicating where the type is declared */
   } TTOPair;

typedef struct TVar {
     TLabel      label;    /* var name and comment */
     TTOPair     typeOrig; /* only the ''type'' field is used */
     struct TVar *next;
   } TVar;

typedef struct TMethod {
    TLabel         label;     /* method name and comment */
    TTOPair        typeOrig;  /* the return type */
    TNTPairDict    *paraList; /* para:type pair */
    TNTPairDict    *usesList; /* the ``method::class'' pair */
    struct TMethod *next;
  } TMethod;

struct  TClass;
typedef struct TInherits {
    TString         name;  /* only the ``name'' field is used */
    struct TClass   *inhNodePtr; /* pointing the class definition */
                          /* where the inherited class is defined */
    struct TInherits *next;
  } TInherits;

typedef struct TClass {
    TLabel         label;  /* class name and comment */
    TInherits      *inherits; /* inherits class list */
    TStringDict    *usesList; /* uses class list */
    TVar           *varList;  /* var list */
    TMethod        *methodList; /* method list */
    struct TClass  *next;
  } TClass;

typedef struct TSystem {
    TLabel         label; /* name and description */
    TClass         *classList; /* class list */
    struct TSystem *next;
  } TSystem;

#endif
```