SNEAKERS: A Concurrent Engineering Demonstration System

by

Robert E. Douglas, Jr.

October 20, 1998

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of
Master of Science
in
Computer Science

APPROVED:

Prof. David C. Brown (CS), Thesis Major Advisor

Prof. David C. Zenger (ME), Thesis Co-Advisor

Prof. Robert E. Kinicki, Head of CS Department

To my parents, Robert & Kathleen Douglas, I gratefully dedicate this thesis. Thank you for always supporting me. I think it was worth it.

Acknowledgments

I would like to thank, DEC and the Competitive Product Development Institute for their support of this work. I would also like to thank Professor Brown for all his help. It was absolutely invaluable. I would like to thank Professor Zenger for jumping in head first to help guide me, and for all those brain storming sessions we had together.

Thanks to Professor Selkow for reading this mess, and for being so kind about it. Special thanks to Jeff Choate for his help with LaTeX and matrices, and all sorts of other neat stuff. And thanks to Aaron Laznovsky, Duane Morin, and Stuart Wells, who took time out of their busy schedules to find bugs and offer suggestions for my work.

More thanks go to my evaluation team of Sundar Victor, Pete McCann, Eric Rasmussen, Jingwen Liu, and Kathy Urbanowicz. Thanks also go out to the WPI AI in Design Group and AI Research Group, and Professors Nabil Hachem, Mike Gennert, and Lee Becker, who provided inspiration.

Above all, many thanks to all my friends who have made the decision to stay in school the best decision I have ever made. They are, in no particular order, Doreen Burrell, Dawn Varacchi, Jim Moore, Ryan Smart, Kevin Dahm, Aaron Laznovsky, Jeff Choate, Stuart Wells, Kristi Henricksen, Scott Runstrom, Trish Gagnon, Duane Morin, Jenn Greenhalgh, Dominic Giampaolo, Brenda Yagmin, T. J. Mino, Eric Felton, Bill Katzman, Chad Council, Andrew Hansford, Brian Weissman, Craig Johnson, Pete Chestna, Taryn Schweitzer, Lydia MacHatton, Jon Stott, Mike “Gundy” Gunderman, Jonathan “Sparky” Davis, and Kristin Sullivan.

Abstract

Concurrent Engineering (CE) has already initiated a cultural change in the design and manufacturing of new products. It is expected to lead to better engineered and faster built products. But, in order for a company to take advantage of the power of CE, the members of product development teams have to be educated in the CE method of product development and how decisions made about one aspect of a design can affect other aspects. They also have to be educated in the usefulness of the tools that can be used for CE. Those tools include intelligent agents which can be used to offer design suggestions and criticisms.

The goal of this project is to build a computer system which will simulate a design environment and demonstrate the essential aspects of CE, in a way that they can be intuitively understood. It is supported by a grant from the Competitive Product Development Institute at the Digital Equipment Corporation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Introduction

This thesis describes **SNEAKERS**, a Concurrent Engineering Demonstration System. The purpose of the system is to act as an educational tool to highlight the importance of the Concurrent Engineering (CE) methodology as a part of modern design philosophy. The specific goals of this project are detailed in Section 1.2. CE itself is discussed in Section 1.3. The last two sections discuss the software developed for this thesis, and the organization of the rest of the thesis, respectively.

1.2 Goals of the Thesis

The main goal of this thesis is to define some of the key issues in CE, and then develop a simple tool to demonstrate several of these issues to managers and engineers. It is intended to be used as a training tool to help educate users in the power and necessity of CE. As such, the system was required to be easy to use, fairly intuitive and self-explanatory. As this system is to be used only in an educational manner, it is a single-user system, with the other members of the CE team being simulated by expert systems.

The first problem was defining what the key issues are in CE and which of those could be implemented in the demonstration system. The next problem was deciding how to design and implement those features in a single-user system within the timeframe of this thesis. Then, a suitable domain had to be established. Designs in the domain had to be simple enough for people from varied backgrounds to understand, but complex enough to offer several aspects from which to evaluate a design. **SNEAKERS** is the product developed to meet these goals.

1.3 Concurrent Engineering

Concurrent Engineering (CE) is a comparatively new design methodology, which enhances productivity and leads to better overall designs. CE has been identified as a vital ingredient in America's attempts to modernize its design and manufacturing practices. In the development of a product, there are many "downstream" aspects to be considered. These include final cost, manufacturability, safety, packaging, and recyclability. These aspects represent different phases in the product's life-cycle. In traditional design methodologies, the product is evaluated after each phase is complete. However, the downstream aspects are affected by decisions made during the design phase. Consequently, these aspects should be taken into account during the design phase.

In the CE scheme, these aspects positively affect the design decisions during the design phase. A team, composed of experts on each aspect, is brought together to participate in the design. These people have information about how downstream issues are affected by design decisions. Having information about downstream issues at design time has several advantages. First, having all of this information minimizes the possibility of needing to redesign some or all of the product. Eliminating redesigns cuts product development time and cost. Next, decisions which take advantage of particular features of an aspect can be made, such as choosing a set of parts in the design for which the manufacturing equipment is already tooled. Stoll [1986] gives an overview of these issues as they apply to the manufacturing aspect of product development. The practice of considering manufacturing needs at design time is known as Design For Manufacturing (DFM). CE attempts to extend the DFM principle to other aspects of the product's life-cycle.

Another advantage of CE is that, while knowledge is being built up about the design of the product, additional knowledge is being acquired about the other aspects of its life-cycle. As the design progresses, the manufacturing expert will learn more about how to manufacture the product, and the packaging expert will know more about how to package it, etc. This accumulation of knowledge helps to speed the product through the development process and get it to the customer more quickly, i.e., time-to-market is reduced.

1.3.1 Ingredients of Concurrent Engineering

Based on research, twelve major issues have been identified in developing tools to support CE. These are the ingredients which have been found to be present in some form or other in many, if not all, of the systems studied.

1. **Design Agents** are the principal actors in developing a design. They make decisions, monitor progress, or analyze parts of the design. These agents can be humans, expert systems, or other computer tools. They interact in order to develop a design.

2. **Multi-disciplinary Goal and Specification Representation** is a means of expressing goals and design specifications in terms used by different disciplines, even though they use different terminologies. It is an attempt to unify the terminologies used by the Design Agents within the scope of a particular product design.

3. A **Catalog** contains descriptions of parts that can be used in the design. These parts can be components or partial designs. They are a means of relating past experience to the current design, and speeding the design process.

4. A **Design Database** contains the current representation of the object being designed. This is used by the agents when designing.

5. An **Accumulated Database** contains knowledge about downstream aspects accumulated along with the design. This accumulated knowledge is discussed

in Section 1.3.2 below.

6. **Shareability of Design Information** allows all Design Agents to use all design information, as they see fit. No information should remain the sole property of any agent, as there may be knowledge about a different aspect hidden in that information.

7. **Communication** allows the agents to communicate despite their different backgrounds. Support for this may come through the development of effective communication channels.

8. A **Manager** schedules agents, oversees negotiation, keeps track of resources used, and monitors and evaluates the progress of the design.

9. A **Design History** captures knowledge about alternatives considered and decisions made, and the rationale behind those decisions. This is useful in determining if the best design was chosen, for credit/blame assignment, for learning, and for use in restoring the design if a design decision has to be retracted.

10. A **Checklist** indicates important decisions that must still be made.

11. A **Standard Interface** to all tools keeps the user from having to learn too many different interfaces. This lends a feeling of working in a single environment, and the addition of new tools into the system does not require a long learning period.

12. **Virtual Collocation of People** makes all the agents appear to be in the same room. This promotes unity among the team members and encourages cooperation in determining a next course of action. It also encourages communication.

This list of issues was developed primarily from [Bedworth et al 1991], [Cunningham & Dixon 1988], [Jagannathan et al 1991], [Kott at al 1991], [Kroll et al 1988], [Lemke & Fischer 1990], [Lemon et al 1990], [Londoño et al 1989], [Stoll 1986], and [Subramanian 1990]

1.3.2 Views of Concurrent Engineering

Clearly, as CE systems require many ingredients, many views of what is “key” are possible. Most of these views concentrate on the needs of the agents in the system (human or computerized) to communicate, or to have the right information available. This is one of the primary concerns of the Concurrent Engineering Research Center (CERC) at West Virginia University. They have put their efforts into indexed databases and electronic conferencing, in order to increase the amount of information that can be communicated to the necessary people involved in a design. (See [Jagannathan et al 1991]).

In a somewhat different view, Douglas & Brown [1992] suggest that the key issue in CE is the accumulation of knowledge. Thus the focus of this view is on “what” is being decided or learned, as opposed to “who” is deciding or “how” it is being done. The primary purpose of CE is to produce a design. That is clearly a process of knowledge accumulation from requirements to a design.

In addition, a CE system should produce descriptions of all other aspects of the manufacturing process, the assembly process, the design for packaging, etc. Thus the goal is to accumulate knowledge about all of the aspects of the life-cycle. Consequently, this view of CE as Knowledge Accumulation can be seen to underlie all CE activities, and is independent of the processes used to generate the knowledge, and the strategies for controlling them.

Figure 1.1 presents a diagram that characterizes this knowledge accumulation process. Knowledge is accumulated during the design process about all relevant aspects of the life-cycle. In general, over time this knowledge moves from abstract to concrete, although in fact this transformation is more complex [Brown 1992b]. The design is complete when all relevant knowledge has been accumulated, and not merely when the component or product design is complete.

1.4 SNEAKERS

SNEAKERS is a single-user demonstration system. Its task is to design towers composed of pieces similar to $TinkerToys^{TM}$. The screen is set up in a windowed

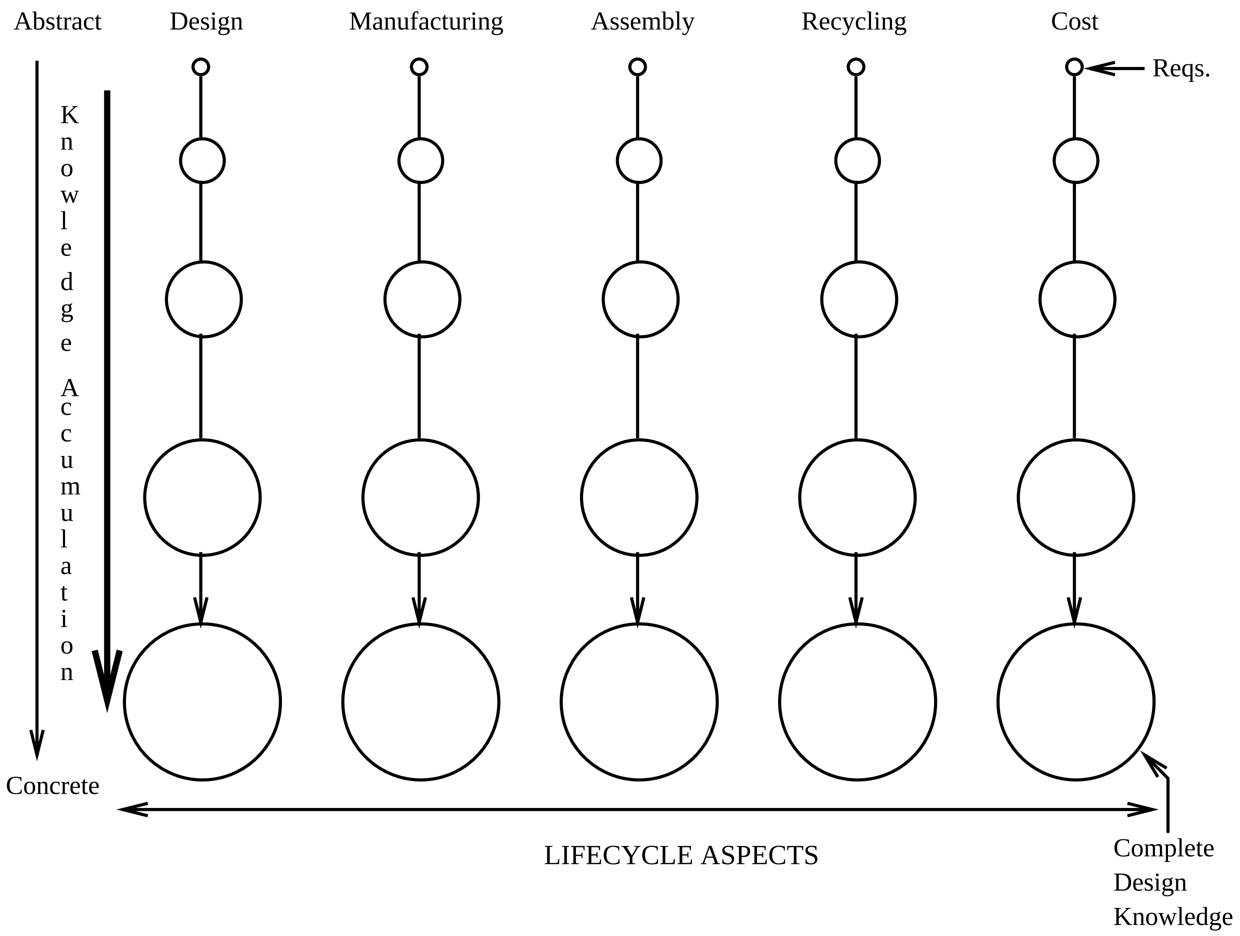


Figure 1.1: Knowledge Accumulation along Multiple Paths

environment, with which the user interacts using a mouse. Various expert systems run in the background and offer criticisms and suggestions to the user concerning recent design decisions. **SNEAKERS** is easy to use, and helpful in demonstrating the value of CE.

1.5 The Thesis

The chapters that follow show the development of **SNEAKERS**. Chapter 2 is a look at other work which is relevant to this project, covering CE, work in Artificial Intelligence, or work in User Interface development. Chapter 3 discusses the tools and methods used in developing **SNEAKERS**. Chapter 4 highlights the choice of the domain in which **SNEAKERS** operates, and the reasons behind the choice. Chapter 5 shows a logical view of the system design, while Chapter 6 details the implementational issues which affected the final project, and shows how the project was developed. The last chapter lists results, conclusions, and suggestions for future work in this area. The appendices at the end include the user's guide and information intended for those who have an interest in the programming details of this project.

Chapter 2

Literature Review

2.1 Introduction

The following chapter is an overview of the current research in areas relevant to this thesis. There are three areas from which this thesis could be viewed, and literature relevant to each view was examined. The first is the Concurrent Engineering (CE) view. The second is the Artificial Intelligence (AI) view. Finally, there is the Human-Computer Interaction (HCI) view. There were several pieces of literature from which no specific ideas were drawn for this thesis, but which were necessary to gain the background needed to understand other work being done. These are referenced in the Bibliography, but no formal summary of that work is included.

Section 2.2 describes some of the work being done in CE, as well as related work in Design For Manufacture (DFM), and tools used in design software. Section 2.3 is an overview of some of the issues facing the choice of an implementation for the intelligent agents that were to be used. Section 2.4 discusses some of the work being done on building user-interfaces, especially for design systems.

2.2 Concurrent Engineering

There are several research efforts into computer support for CE currently underway. The best known of these is the DARPA Initiative in Concurrent Engineering (DICE) at the Concurrent Engineering Research Center (CERC) at West Virginia University. Other research efforts include the CAD Framework Initiative (CFI), the Open Systems Architecture for Computer-Integrated Manufacturing (CIM-OSA), and the Engineering Information System (EIS). These four research efforts are reviewed in [Jagannathan et al 1991]

Jagannathan et al identify three areas in which CE research is progressing: Management Processes, Technical Practices, and Information Technology. They then focus on the Information Technology issue, which the four previously mentioned systems support. Within this category, they identify five areas in which computers can be used to support CE: sharing information, collocating people and programs, integrating tools and services with frameworks, coordinating the team, and capturing corporate history.

Sharing information is necessary to promote cooperation among the members of multi-disciplinary design teams. The models each of the members uses in the design process may be quite different, however, and the system should be able to translate models produced by one designer into the format and vocabulary used by another.

Collocating people and programs is achievable by networking. The key to maintaining this virtual collocation is in making the access to programs, people, and data across the network transparent to the user.

Integrating tools and services with frameworks is a means of allowing designers to use different tools with ease. The idea is to support a single means of user interaction with all of the tools, to make them more uniform. This would lead to less training time on new tools and a greater chance that designers would use all of the tools available to them.

Coordinating the team is concerned with keeping all members of the design team apprised of the current state of the design. Whenever a design decision is proposed, all members of the team are informed.

Capturing corporate history is a means of keeping track of design decisions and the reasons for them. It uses an electronic design notebook, or some other means of recording decisions.

Londoño, et al [1989] are doing research to build a system to support CE for DICE. This system is designed to help designers participate in cooperative design. They have chosen to use a blackboard for communicating and for control of information flow. All data about parts is held in the Product, Process and Organization (PPO) database. This database is accessible by all of the product's designers. Individual designers use Local Object Workspaces (LOWs), in which they can modify current designs and test hypotheses before committing everyone on the design team to a decision. The whole design process is overseen by the Project Lead (PL). The PL must be able to keep track of new tasks and follow the progress of the tasks, as well as generate new tasks, in order to change the focus of the design, and assign them to designers.

Representations of the designed products are stored in frames and pointers to AutoCAD files. This information is stored in the PPO database. Designers must have access to all the information in the PPO database. The database should support user friendly searches for parts on which a designer wishes to work. Though the system defaults to helping the designer in a bottom-up design fashion, there exists the option to do all or part of the design top-down. Initialization of the blackboard includes adding design specifications, determining dependencies, deciding on initial tasks, and including known heuristics.

Global, local, and implicit constraints are supported by the system. Global constraints refer to overall specifications that apply to the design as a whole. Local constraints refer only to individual parts. Implicit constraints are constraints which need to be inferred from the constraints on subparts. Reasoning with constraints can be done to allow the system to spawn new tasks to be done. Dependencies are kept by the system to indicate the other aspects of the design that may be affected by a design decision about a part. Those on the dependency list are notified when a change occurs. Maintenance of versions has not yet been addressed in the system.

Heuristics are another form of design knowledge used in the system. They can be used to help schedule tasks and develop uniform plans of action during a design. Negotiation is an important consideration involving multiple designers. The designers are allowed to vote on accepting a change to the design.

The Function Advisor (see [Kott et al 1991]) is another CE support system being developed. Their main concern is with allowing "a computer-based advisory system to support the cooperation between multiple engineering agents." The research issues include: planning and management of the activities of multiple agents; representing and modeling the product; managing multiple representations and versions of the product; developing Engineering Databases capable of supporting Concurrent Engineering; managing and propagating constraints and avoiding inconsistencies in the current state of the product description; sharing the information between the multiple agents without creating an "information blizzard."

The objective of the system they are building is to help a group of knowledge workers (designers) design an engine and all of its component parts. They categorize the functional objectives of the Function Advisor they are developing into four groups: advising the human organizer of tasks to be performed; retrieving, organizing, and conveying pre-stored information relevant to the current needs of the organizer; inferring information that is not explicitly stored; and monitoring the design for consistency, completeness, and correctness.

The more specific functional goals for the Function Advisor are given below:

1. Provide an inference mechanism for reasoning about the design organization.
2. Alert human workers of abnormalities or inconsistencies between the design and the design goals.
3. Suggest a plan of design activities to human designers.
4. Advise the human designer on dependencies between parts of the product.
5. Find parts, constraints, or other design knowledge of use to a particular designer.
6. Prevent attempts to finalize a design before designs which affect it are finalized.
7. Collect and store all product documentation.
8. Detect those parts which are dependent on other design decisions.
9. Identify the need for special purpose design aids.
10. Guard against re-occurring design errors.

These goals are consistent with other CE research.

Subramanian, et al [1990] are developing a design support environment for CE. They used the system as a testbed for their theories about which kinds of information are used during group design. When different groups work on a design from different aspects the design activity is usually done concurrently. They propose that a computer system designed to help groups from the various aspects of product design should work similarly to the way that those groups do. The groups share computations, figures, and any other data they feel necessary. This means having mechanisms to represent designs in different ways and to maintain copies of all of the current design data. It also requires a means of scheduling the use of the data.

Knowledge-based systems which communicate by means of a blackboard are used in a project called DICE (Distributed and Integrated environment for Computer-aided Engineering) (see [Sriram et al 1989]). This project is not associated with the DARPA Initiative in CE. Sriram's DICE is made up of a control mechanism to coordinate the activities of its modules and users, a blackboard to post the current state of the design, and knowledge modules to represent various aspects of the design. The system uses an object oriented approach to design. It allows for negotiation between agents. It provides support for the databases that are needed for this kind of distributed knowledge design. The knowledge modules have various roles: Strategy (helps the control mechanism by determining the next course of action), Specialist (an aggregation of expert systems to help make decisions), Critics (keep track of consistency in the design), and Quantitative (algorithmic evaluation or CAD tool). The Blackboard is divided into three partitions: Coordination (keeps bookkeeping information), Solution (contains the object hierarchy and current design), and Negotiation (contains constraints on the design and a trace of the negotiation process). All of this allows the user to interact with various autonomous agents when designing part of a building.

Some of the tools which could be included in a CE system are mentioned in [Boothroyd & Dewhurst 1988] and [Lemon et al 1990]. Both sets of tools are concerned with cost estimation over the entire product's life cycle during the design phase. Incorporating these tools and their methods into design systems allows designers access to knowledge about other aspects of the product's life-cycle and the processes to be used by those aspects. Since cost is affected by the processes chosen in manufacturing, assembly and disposability as well as material selection and other, more concrete costs, analysis of life-cycle costs gives designers a means of focusing on other life-cycle aspects.

The issues concerned with version control are covered in [Katz 1990]. In his work, Katz attempts to unify terms used in version management and propose a scheme for developing a complete version management system. The details are quite extensive. What he proposes is an organization in the database which connects versions of different parts together to form a design. But it also connects those versions to their parents and offspring. So the entire database will contain links to different versions of the parts, and these can be combined in different ways to obtain different current designs. These links can be used to search for alternatives and to propagate constraints to all versions of a part.

There are several different thrusts in the area of computer support for CE. As Jagannathan et al state, CERC and the DICE projects concentrate on the availability of information and information sharing. Their main goal is to provide as many people as possible with as much useful data as possible so as to make decision-making easier. Jagannathan et al state that this is also the thrust of the CAD Framework Initiative (CFI), the Open Systems Architecture for Computer-Integrated Manufacturing (CIM-OSA), and the Engineering Information System (EIS). Londoño et al are also working to support this view. They are also concerned with supporting negotiation.

Subramanian et al are also concerned with sharing information among the members of the design team and supporting this kind of interchange. Sriram et al are concerned with communication between intelligent agents involved in the design. They are also concerned with the computer's handling of negotiation in the event that some decision is not amenable to all parties. Kott et al concentrate on using the computer to keep track of the design, maintain the focus, and monitor the consistency of the design.

This section has attempted to present a summary of some of the current CE research, and has tried to reflect what their main concerns are. It has also attempted to characterize (perhaps wrongly) what they consider as the key aspects to the problem of producing CE systems.

2.3 Artificial Intelligence

Artificial Intelligence (AI) is a term which covers the issues relating to the development of intelligent agents in this thesis. The purpose of these agents is to simulate members of the CE design team. There are two important issues which arise from this role. The agents must have particular roles and the agents must be able to communicate. The literature on these roles is covered in this Section.

Before discussing particular aspects of the intelligent agents in a design system, it is necessary to show the applicability of using expert systems as design aids. The development of the Computer-Aided Mechanical Expert System (CAMES) is discussed in [Campbell et al 1991]. The motivation for that project was their difficulty in finding experts to do machine design. This suggested an expert system to fill this gap. The first machine chosen as the design domain was one of the hardest for an expert designer to design. Campbell et al hoped to find all the problems this way and make development of further expert systems for less complicated machines easier. LISP was chosen as the AI language, and AutoCAD, which is written in LISP, was used as the design drawing system. The system maintains a library of components used, including spatial attributes. The expert system helps to design the machine. CAMES is a successful example of how expert systems can be used as design aids.

Brown [1992a] describes the task of design and the applicability of various AI techniques, including Knowledge Representation, Constraint Satisfaction, Search, Learning, Case-based Reasoning & Analogy, and Qualitative Reasoning, to the design task. Several roles for expert systems in a design system are identified in that paper. Some of these were adapted for use as the various intelligent agents in **SNEAKERS**. Each agent has a specific role, as described below:

1. **Advisor** – makes recommendations for the next decision based on the current state of the design. The user may ignore the advice.

2. **Critic** – compares the design to certain standards, and offers some criticism of the last action based on those standards.

3. **Suggestor** – takes the criticisms from the **Critic** and offers suggestions for satisfying them. Again, these suggestions may be ignored.

4. **Analyst** – offers numerical analysis to derive attributes, such as strength, cost, or size. These analyses are used when determining the success of the design.

5. **Evaluator** – evaluates the whole design from one perspective. Using the **Analysts**, evaluation determines how well the design takes into account the needs of that perspective.

These are only a few of the many possible roles that intelligent agents can play in a design system. Other roles can be extrapolated from the definitions given by Brown.

The literature on combining these agents into a cooperative structure is considerable. The method chosen to be modeled in **SNEAKERS** was a blackboard. Nii [1986a] [1986b] presents considerable background information about blackboard systems, some of which is summarized here.

In [Nii 1986a], the first part of this survey of blackboard systems, is a summary of what a blackboard system is and what its components are. The individual intelligent agents are referred to as Knowledge Sources (KS). The Blackboard data structure is a programming abstraction. It can be *read* by the individual KS's. KS's can also *write* to the Blackboard. The Control system is responsible for routing information to those KS's which can use it and decides which KS to use for a particular task. The actual implementation of these elements is different for different blackboard systems, but they are the essential ingredients. Nii also describes how this problem-solving method is applicable to many tasks, including the HEARSAY-I project, which was the first blackboard system developed. Its purpose was to recognize human speech.

[Nii 1986b] begins where the last article left off. It includes: HEARSAY-II, the successor to HEARSAY-I; HASP/SIAP, that maintained surveillance of surface ships and submarines from sonar data; CRYSALIS, which was designed to infer the three-dimensional structure of protein molecules; and TRICERO, which monitored an area of airspace for traffic. Nii then talks about skeletal systems, OPM in particular. She discusses applying Blackboards to the Scene Understanding problem, where the task is to label objects in a photo taken by a low flying aircraft. The final section deals with Knowledge Engineering issues which might lead to the decision to use a Blackboard approach. These include Problem Complexity and Ill-structured Problems. Nii suggests ways the approach could be used to formulate problems, as a system development tool, and as a research tool.

Blackboards are a part of work done by the Blackboard Technology Group. (See [Blackboard 1991].) As they state, blackboards allow separate knowledge systems to take information provided and either ask for information from others or give information to others. Their system is composed of a Controller and several Knowledge Sources (KS). The Controller uses its rules to determine which KS will run and when that KS will run. Each KS is also totally independent of the others. They can use different approaches to solving the same problem.

Durfee et al [1989] discuss the idea of Cooperative Distributed Problem Solving (CDPS), which is similar to the Blackboard problem solving approach. They discuss many issues concerning multiple distributed problem solving systems. These systems break a problem down into subproblems, in order to generate a solution to the problem. CDPS works through local problem decomposition. When a system is given a problem, it has the option of solving the problem it has been given by itself, or decomposing all or part of the problem and sending it off to another system. Contrary to this method, Blackboard systems work through global decomposition, with a Controller in charge of the whole problem. This is the main difference between CDPS and Blackboards. In CDPS, no system controls the whole process but each tells the others when to activate, whereas in a blackboard system, the scheduler has control.

Some of the important issues in the CDPS method are as follows: negotiation;

functionally accurate cooperation; organizational structure; multi-agent planning; sophisticated local control; and formal frameworks. Negotiation is a major concern here. The system that has control of the problem sends a request specifying the problem to be solved, and those systems that have the resources to solve it respond, listing their capabilities, so the "Controller"(for this individual subproblem) can decide among them.

Myers et al [1991] discusses the Intelligent Computer Aided Design System (ICADS) expert design advisor. This is a very extensive system which includes a user interface to a CAD system, six domain experts, a conflict resolver (i.e., negotiator), and a blackboard for handling communication. The user interface is a "wrapper" for a CAD drawing system, so that the user can interact with the design drawing and the expert advisors. The domain experts were written in CLIPS and each have a single machine dedicated to them. Therefore, they are truly distributed. The conflict resolver has the task of deciding among conflicting design decisions made by the domain experts. One of the major problems with the conflict resolver was giving it the ability to recognize when it had been in a certain situation before in order to avoid infinite loops of swapping values between conflicting experts. A blackboard was set up so that all of the experts would know where to voice their questions and answers. It is a separate system running on a separate machine.

2.4 User Interfaces for Design Systems

The following section covers some of the work in Human-Computer Interaction and User Interface development that was the basis for this thesis. The first book to mention is [Shneiderman 1992]. This reference provides a comprehensive view of the issues that should be considered in designing a computer's user interface to be user friendly. It includes concerns about information overload and positioning of data. Shneiderman also expresses a need to simplify input methods, using a menu or mouse where possible. This book acts as a basis of knowledge about user interfaces that was used in making later decisions.

[Lemke & Fischer 1990], [Fischer et al 1989], and [Fischer et al 1990] all describe similar work on developing programs to critique designs. In [Lemke & Fischer 1990], the design being critiqued is that of a User Interface. This is interesting because of the program's encoded knowledge, but also because of the design of human-computer interaction for their system. Lemke & Fischer list some of the features they require of a distributed problem solving tool, used for designing this software. Among these are some ideas which help make the system more user friendly:

1. A **Checklist** keeps the user's attention focused.
2. A **Palette** of components limits the number of choices the user has to make, and makes interaction simpler.
3. **Critics** analyze the design as it progresses. These must provide useful, constructive criticisms, and not be too intrusive.
4. **Specification Sheets** guide the user and help to focus the critics so that criticisms can be understood by the user in terms of these specifications.
5. A **Catalog** of things for reuse or a potential case-based solver to simplify the solution.
6. A **Code Generator** to make the design usable.

Some of these ideas appear in the list of issues given in Section 1.3.1.

Fischer et al [1989] discusses two systems, CRACK and VIEWPOINTS. CRACK has critics which analyze the design of a kitchen layout. These critics explain why certain decisions may be bad, and why others are good decisions. The text for the critics is all pre-written. VIEWPOINTS is a hypertext system which gives arguments and alternative viewpoints, and is attached to graphics views. However, VIEWPOINTS is not yet automated to find relevant information based on which critic is activated. The goal of Fischer et al is to tie the two systems together by using VIEWPOINTS as the explanation facility for CRACK.

In [Fischer et al 1990], several uses of critic systems are discussed, including supporting learning by offering criticism of solutions and/or methods of problem-solving;

providing design environments which communicate with users in their domain language; and developing cooperative problem-solving systems which combine the talents of humans and computers to come up with the best solutions.

Fischer et al then discuss work on the JANUS system, a combination of the above two: CRACK and VIEWPOINTS. They list several steps in the critiquing process:

1. **Goal Acquisition** – the critic has to understand the goal of the problem. This understanding may be different from user's understanding.

2. **Product Analysis** – the system either develops its own solution and compares it to the user's, or it analyzes the user's solution against certain criteria.

3. **Critiquing Strategies** – the system designer has to decide how often to critique, how strongly, etc. These decisions can affect the user's reactions to the system.

4. **Adaptation Capability** – the user should be able to turn off certain critics, or the critics should adapt to the user's needs.

5. **Explanation Capability** – the system's ability to explain why one decision is wrong and another is correct using hypertext.

6. **Advisory Capability** – the system should be able to offer potential solutions, instead of just criticisms of the user's solution. This makes the system less frustrating to use, and can help the user greatly.

This work was useful in determining how the agents would interact with the user in **SNEAKERS**, as well providing some ideas for screen layout.

The research of [Wong et al 1990] is part of the work on Sriram's Distributed and Integrated environment for Computer-aided Engineering (DICE) project, first mentioned at the end of Section 2.2. This article first gives an overview of the DICE system. It then identifies the requirements of the User Interface (UI), tools for visualization and understanding of the different components of the work, searching for information about a specific piece of data, data transformation between application and central database, communication and negotiation between users, monitoring blackboard, creation and modification of the object base, creating and updating documentation. The DICE UI has the following tools:

1. data management tools, which include visualization tools, editors, query tools, and documentation tools;

2. blackboard (BB) interface tools, which include translators from the BB to the local database, tools for displaying BB status, tools for displaying alert messages, and BB coordination tools;

3. communication tools, which include electronic mail and electronic conferencing;

4. application specific tools, which is a "catch-all" for anything else that might be thrown in for a particular application.

The local databases allow the designer to choose a hypothetical or real design session. Hypothetical sessions will not be saved. This whole system is very extensive, and the user interface is complex enough to be able to handle all of the user's possible alternatives. This complexity shows how the extent of the program can affect the user interface design.

Silverman & Mezher [1992] is an article on the use of critics as design aids. Silverman & Mezher introduce several different types of design mistakes that can be countered by proper use of critics. These types of mistakes can be split into two categories and then broken down into several subcategories under each of those:

1. misconceptions
   accidents
   cognitive biases
   motivational biases;

2. missing concepts
   insufficient training
   knowledge decay

promotion
interdisciplinary breadth of the engineering domain

Errors made during the design can be classified into these categories, and this classification can be used to guide the criticisms to be offered.

Silverman & Mezher discuss the use of critics in a Design Support Environment, which they define as a three-layered model of the design task and tools used: the knowledge based collaboration and synthesis layer, the visualization and analysis layer, and the information layer. The design process passes down these layers to generate the design, then passes back up through them to test the design. By generating an example critic, they were able to get some feedback as to how a critic could be properly used in a CAD system. The comments they received were that the critic should supply more before-task tutoring; the system should use graphics coloring to guide the users choices; praise information should be omitted; and there should be some prototype analogy module offered. Silverman & Mezher developed three distinct strategies for critiquing. Critics should follow one of these strategies:

1. **Influencers** try to avoid common errors and work before specific subtasks
2. **Debiasers** try to avoid biased information and provide the correct information
3. **Directors** guide the user toward good design choices.

They developed two principles based on their research. Principle 1 is "The critic builder should draw from a library of active Influencer, Debiaser, and Director strategies." Principle 2 is "To promote effective criticism, there must be a mutual, two-way exchange of ideas. The critic must be flexible and adaptive." These ideas were useful in developing agents which would interact with the user in the **SNEAKERS** system.

Another system, AskJef [Barber et al 1992] helps software engineers design interfaces by providing design examples, guidelines, errors, and stories. Much of the system is graphical, with examples showing what to avoid or to try. It uses text, graphics, animation, and voice to present the user with relevant information when presented with a specification of an interface design problem.

2.5 Summary

This Chapter has presented summaries of some of the current research which influenced the development of **SNEAKERS**. It includes coverage of Concurrent Engineering, Artificial Intelligence, and User Interfaces for design systems. This thesis does not constitute a comprehensive view of these issues, rather it provides a taste of the issues which were considered in developing this Concurrent Engineering Demonstration System – an educational tool which also shows the use of computer technology and AI in a CE support system. The next Chapter describes the tools and methods used to develop **SNEAKERS**.

Chapter 3

Methodology

3.1 Introduction

This chapter discusses the technology and methodology used in designing and implementing the Concurrent Engineering Demonstration System, called **SNEAKERS**. The first of these is the expert system development tool called CLIPS, which was used to develop the agents, and to maintain a database of the parts of the object being built. Next is the Motif user-interface development/prototyping tool VUIT. This tool was used to put together the pieces of the user-interface into a coherent structure. A technique known as storyboarding was used to design how the screen would look at crucial points during program execution, before the user interface was built. This chapter concludes with a discussion of the knowledge acquisition which helped to determine the features that should be displayed by the system and how best they could be shown, as well as the knowledge that went into the agents.

3.2 CLIPS and COOL

CLIPS is short for C Language Integrated Production System. It was developed for NASA by the Software Technology branch and the University of Georgia. CLIPS is detailed in [Giarratano 1991a], [Giarratano 1991b], [Giarratano 1991c], [Giarratano 1991d], and [Giarratano 1991e], and it was these references which made it possible to develop expert systems using CLIPS. CLIPS is available to the public and has several features which make it a desirable choice for producing the expert systems in **SNEAKERS**. It offers the ability to integrate expert systems developed in CLIPS into the environment of a standard C program. This allows for quick prototyping and testing of expert systems alongside the development of a control program written in C.

Since Motif was chosen as the user-interface development method, the ability to integrate CLIPS and Motif was a great benefit. In [Liu 1991], it is shown that CLIPS and Motif can be effectively integrated to form a useful and intelligent interface to a program which determines how tunnels should be made.

CLIPS version 5.0 was used, which introduces objects and object-oriented techniques, via COOL – CLIPS Object Oriented Language. These objects can be built into a part hierarchy which describes the parts of a tower. Towers are a part of the domain as discussed in chapter 4. These objects can be accessed by both a regular C program and the CLIPS expert systems. By using this part of the CLIPS package, a database was developed which contains knowledge about the domain, and knowledge about the relationships between the parts in the design. It is also possible to attach *daemons* to the objects. These daemons are programs which run whenever a specified action happens to an object. These programs can produce facts and transfer knowledge to appropriate agents as needed.

In the implementation of **SNEAKERS**, there is a common fact list that is controlled by CLIPS, which allows all knowledge to be seen by the separate agents. The fact list also allows the agents to communicate with each other by posting to the fact-list. The **Critic** posts criticisms to the fact-list, while the **Suggestor** reads these and adds suggestions to the list. Other agents can use the fact-list as a communication center as well. The benefit of this common fact-list is to overcome the problem of information routing which is mentioned in Section 5.2.3.

CLIPS also has the advantage of having an interpreted mode. This mode can be used for prototyping the agents before including them in the entire program. It is also useful for checking syntax and for trying several ideas about the way in which rules should be written. This feature was very useful for fast development of several agents simultaneously.

CLIPS was a very good choice for the expert building tool to use in conjunction with the other tools used. CLIPS is very robust and lends itself to inclusion in a variety of larger programs. The features listed above allow CLIPS to provide the required flexibility and power for developing the agents of **SNEAKERS**.

3.3 Motif and VUIT

Motif is a popular set of graphic interface tools for use with the X window system. The basic construct in Motif is a *widget*. A widget is the name given to any graphical object in Motif. Each different type of widget has its own specific properties. Widgets are arranged in a type hierarchy, and many attributes are inherited through this hierarchy. A widget can be as simple as a single button or as complex as a dialog box, containing a message and several buttons. For more information on Motif and Motif programming see [Berlage 1991].

VUIT, Visual User Interface Tool, a product of Digital Equipment Corporation, is used for building Motif based graphical user interfaces. One of the advantages of this product is that actual Motif widgets are used to build the screen. This means that the interface designer can see what the interface will look like when it is finished, as it is being built. More about VUIT can be found in [Digital 1990].

The interface is built by dragging Motif widgets from a menu palette and placing them on the screen building area. The menu is arranged hierarchically: all buttons are grouped together; all windows, dialogs, etc. The physical attributes of these widgets are set by directly manipulating the widgets. Attributes such as height, width, and physical positioning are set in this way. Other attributes can be added by choosing from a list of valid attributes and values. The system offers a great amount of guidance and keeps the user from making invalid choices.

Callbacks are the means of interacting in a Motif built environment. A callback is executed when the user performs an action on a widget. The callbacks needed in a particular application can be added to the widgets as they are built in VUIT.

VUIT can output the code for implementing the interface that has been developed using it. It can output this code in several languages. However, C was the only one used for developing **SNEAKERS**. VUIT outputs a *makefile*, a file used to build the entire program from a set of individual modules. It also outputs the C program necessary to invoke the X and Motif environments. Finally, it outputs a *uil* file, which contains all the information about the widgets, their attributes, and the callbacks associated with them. The user still has to assign the functionality to the callbacks that are assigned to each widget, but the code generated by VUIT is sufficient to run a skeleton simulation of the interface, lacking full functionality.

These tools are useful for quickly developing the interface after it has been designed. Functionality can be added after the basic interface structure has been established.

3.4 Storyboarding

Storyboarding is a technique useful for designing the look and feel of a user interface. The result is a hand-drawn picture of what the system's user interface might look like. This picture can be critiqued and revised. As a result, the designer is better able to determine the abilities of the system. These abilities have to be addressed before the system can even be prototyped. In [Shneiderman 1992], many of the concerns that should be taken into account when designing a user interface are discussed. Storyboarding helps focus on those issues early, when changes can be easily made.

The process is begun by developing an overall view of the screen. The major components of the interface are determined. These components are arranged in the desired format to show what the screen will look like at startup time. Then the possible user actions are highlighted and the activation methods are determined. If a button is to offer a specific action, that button must be drawn in at the appropriate place. One action is chosen, and the screen is drawn on a second page to show the differences between the two steps. This process continues until every possible screen picture has been drawn to show how the various user actions affect the look of the system.

Many decisions about the user's abilities must be made during the time spent storyboarding the interface. In **SNEAKERS**, the effects of choosing agents had to be decided; the changing look of the design decision palette also had to be shown; the options available in menus had to be addressed; the means of entering requirements had to be solidified; and the division of the roles of each part of the interface was determined.

From these hand-drawn sketches of the interface, it is possible to determine the widgets necessary to carry out the desired tasks. These widgets can then be produced using VUIT, or some other tool, and a prototype interface, with most of the surface functionality evident, can be developed in short order.

3.5 Knowledge Acquisition

There were two forms of knowledge acquisition that had to be done to develop **SNEAKERS**. The first concerned concurrent engineering. The second concerned the choice of a domain for the project and knowledge about design within that domain.

Knowledge about concurrent engineering was gathered though several sources. A survey of the literature, summarized in Chapter 2, provided a basic understanding of the topic. Professors Bausch and Zenger of the WPI Manufacturing Engineering department also sparked ideas and guided the search for information. After this, a meeting with Paul Posco, A. J. Overton, John Hamer, and David Meeker of the Competitive Product Development Institute at DEC offered some other suggestions.

One point of that meeting was to determine how close our analysis of the literature was to what they were saying about concurrent engineering. They confirmed the points gathered through the literature. The other point to the meeting was to attempt to determine the important features that should be included in **SNEAKERS**.

The choice for the domain is detailed in Chapter 4. Professor Zenger helped to determine the domain, as his expertise was needed to choose a domain that would meet the requirements of being intuitive and multi-disciplinary.

Once the domain was selected, Professor Zenger helped provide knowledge for the agents. He provided sample rules and acted as an expert in tower building. Knowledge about how to design a tower, and when to invoke other agents, was gained by carefully stepping through the process of designing several towers. This knowledge was captured in part through the storyboarding described in Section 3.4.

3.6 Summary

This chapter presented the tools and techniques used in developing the Concurrent Engineering Demonstration System, **SNEAKERS**. The first item discussed was the expert system shell, CLIPS; then Motif and the Motif interface-building tool, VUIT. Next, an overview of the technique known as storyboarding was given. Finally, the methods used to gain knowledge for use in the expert systems and the interface were discussed. The next chapter will show how a domain for **SNEAKERS** was selected.

Chapter 4

Domain Selection

4.1 Introduction

In order to build an Intelligent Computer-Aided Design (IntCAD) tool, a domain needed to be selected and defined. This is because a general purpose CAD tool would be beyond the scope of a master's thesis. In this chapter, the requirements of the design system and reasons for those requirements are discussed. Then the choice of tower design as the domain is detailed. The sections following that show how this domain meets the requirements set forth. The user's and system's functions are discussed. The last part of the chapter discusses the various agents which can be built for this domain.

4.2 Domain Requirements

In order to keep the domain manageable, it had to meet several requirements. It had to be intuitive, meaning that the objects to be designed, the rules the experts would follow, and the actions needed to design the objects all had to be something most people could envision in their heads and not expect to need books of tables and parts lists. This was so that the system could be used by managers, executives, and engineers with equal ease.

The domain had to lend itself to a common design paradigm. In our case we chose the specification of a design from abstract to concrete as the paradigm. Using this system, design decisions would refine a structure from its most abstract form to a concrete collection of objects with specified parameters. Other paradigms include modification of alternative designs.

The user must be able to make most of the design decisions. Design decisions might include material selection, size of objects, placement, etc. As many of these as possible should be left in the hands of the user to keep the user in charge of the design.

The system should help make the choices, limiting the number of actions among which the user has to choose. The system should handle anything which would be too repetitive or difficult for the user. Anything that might require the user to use a calculator or book should be controlled by the system, so that the design can remain intuitive.

Finally, the domain should make itself available for commentary from many aspects and also from many different types of agents. The types of agents are defined in chapter 2. This is necessary to show the power of concurrent engineering even applied to a simple domain such as this, and to complete the main task, which is to show how a simple design decision affects multiple aspects in different ways.

4.3 Tinker Toys

The domain chosen was the design of several types of towers which can be built using objects similar to PLAYSKOOL's *TinkerToys*^{TM}. The idea for this is derived from a workshop used in the Design for Manufacture class at WPI under the direction of Prof. Zenger. The idea is that different colored pieces have different costs, and manufacturability concerns, and the participants must take as much of this into account as possible and build a sixty inch tower out of the building materials.

The basic construction elements remain the same in our system: supports, of varying length and connectors with a holes for connecting supports at 45° and 90° angles. For towers in **SNEAKERS**, in order to allow more flexibility, and a little realism, the type of material of which the support is made is also allowed to vary. Different connectors are also defined, which each allow different angles, and can connect different materials.

Specifically, supports can be 1, 2, 4, or 6 feet long. (Units have been changed to feet for convenience in thinking.) They can be made of aluminum, steel, or wood. Details about these materials can be found in Table 4.1.

Material	Cost Ratio	Weight Ratio	Strength Ratio	Manufacturing Time Ratio	Disposal Cost	Illegal Connectors
Aluminum	8	3	4	2	1	None
Steel	2	10	10	4	2	None
Wood	1	2	1	1	3	Bolt, Weld

Table 4.1: Support Material Attributes

Connectors come in the form of bolts, snaps, and welds. Some limitations exist, for example, wood cannot be welded. Bolts work at 45° incremental angles starting at 0°. Snaps only work at 90° increments starting at 0°. Welds are the most versatile, they have the same angles as a bolt, plus the addition of angles at 30° increments. These attributes are shown in Table 4.2.

Connector	Cost Ratio	Assembly Time Ratio	Disposal Cost	Allowed Angles	Illegal Materials
Bolt	1	3	1	0°, 45°, 90°	Wood
Snap	2	1	1	0°, 90°	None
Weld	5	8	4	0°, 30°, 45°, 60°, 90°	Wood

Table 4.2: Connector Attributes

This set of *TinkerToy*^{TM}-like materials still allows for great variety in the types of towers that can be designed. It is still fairly intuitive, but needs to be guided to keep the design flowing toward an acceptable design of a tower.

This domain lends itself to the Abstract-Detailed design paradigm. The *Abstract Design* phase consists of choosing one of the abstract towers listed in section 4.4.

The *Intermediate Design* phase consists of selecting the height, base and platform measurements. Then, the *Detailed Design* phase consists of specifying the actual components of the tower from those listed in Table 4.1 and Table 4.2.

4.4 Abstract Towers

In order to further constrain the design task, and to map to the flow of a design from abstract to concrete, a set of abstract towers were defined. Three abstract towers are allowed, each of which has a distinctive shape. The design of a tower is guided by the abstract tower chosen to be the model. The user then chooses components to build a tower as close to the abstract "ideal" as possible. The abstract supports are replaced by actual supports placed on top of each other, with angles between the joints determined automatically according to the connectors at the joints. There are three types of abstract towers supported by **SNEAKERS**. They are the I-type, the A-type, and the X-type. Each is discussed below and in Table 4.3.

Type	Cost Ratio	Relative Strength	Relative No. of Pieces	Assembly Time Ratio	Marketability Ratio
I	1	1	1	1	8
A	6	3	6	6	3
X	5	4	5	7	6

Table 4.3: Abstract Tower Attributes

4.4.1 I-type Towers

An I-type abstract tower is the easiest tower to visualize. It is composed of a single support. Its base and platform sizes are relatively meaningless, since the support is placed in their center (see Figure 4.1). The supports which replace this support can, therefore, usually be snap-fitted together and the tower can be designed to use a few longer supports and few connectors to make up the height of the tower. There is no bracing to support this tower, and therefore, the tower is also the least stable.

It is also the cheapest and quickest to build, because of the small number of pieces required to build it. It is a good choice for a tower that is not required to support a great deal of weight, or that is not expected to encounter any great wind forces.

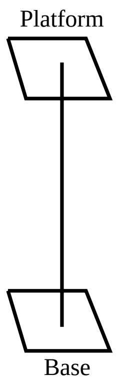


Figure 4.1: Abstract View of an I-type Tower

4.4.2 A-type Towers

An A-type abstract tower has a wide base and a narrower platform level. It has three supports placed at the corners of the base of the tower (see Figure 4.2). The base and platform are isosceles triangles. The longest side of each triangle is twice as long as the shorter sides. This is done to simplify the math needed to place components. Because arbitrary angles are not available to the connectors, an A-tower design may significantly deviate from the original abstract tower, when the abstract support guides are replaced by actual supports and connectors. The resulting tower may look quite different, depending on the angles required to achieve the chosen height, base, and platform measurements. Any angle can be simulated using the proper arrangement of support lengths and angles. However, some angles will require a larger number of pieces to model, increasing the cost and construction time. It is very sturdy, but it also requires a large building site. It will not topple, and is not likely to buckle.

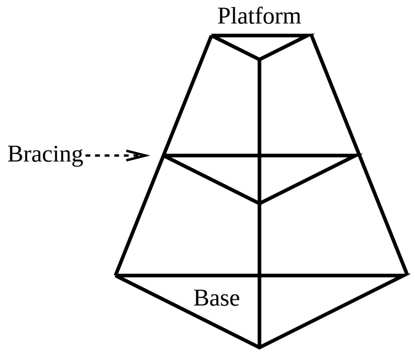


Figure 4.2: Abstract View of an A-type Tower

4.4.3 X-type Towers

An X-type abstract tower is generally the most stable. It is composed of crossing supports, connected by bracing at the center. The supports attach to the corners of the base and platform (see Figure 4.3). The abstract support guides are replaced by supports and connectors in the same way that the A-tower's support guides are replaced, bringing out similar concerns about number of pieces, and following the model. A further concern is the need to make the supports cross at a point where two supports end, so that they can be connected by a single connector. Failing this, they have to be placed one inside the other, seriously decreasing the stability of the tower. It is a greater effort to design and build one of these towers. But it is even sturdier than the A-tower and the crossed supports allow extra support against buckling. It, too, is not prone to toppling.

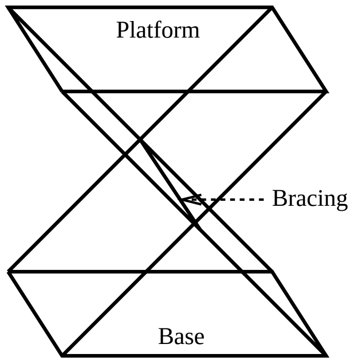


Figure 4.3: Abstract View of an X-type Tower

4.5 User Controlled Features

The user of the system has control over a variety of the features used in designing a tower. Although total control is not available, the user does have the ability to significantly influence the design. Below is a summary of the user's options.

Requirements: The user can specify a variety of requirements that the tower must meet. These include the *weight* that the tower must support, the *time* needed to construct the tower, and the importance of *minimizing cost* (which can affect some choices of material, etc). Environmental factors, such as *windspeed*, *rainfall*, *smog*, and *acid rain* can affect the choice of materials, and the shape of the tower. The user can affect the final outcome based on what values are chosen for these requirements.

Abstract Tower: The user has the final say in the type of tower, either *I*, *A*, or *X*. This choice may be contrary to the advice of some expert systems, but is still within the user's control. This choice affects how the tower will ultimately look.

Measurements: The user can change the *height*, *base size*, and *platform size* of the tower. This affects the angles that the final pieces will have to follow and can affect the stability of the tower and the number of pieces that are needed to build it.

Component Selection and Positioning: The user selects the order in which components are added to the design. To add a component, the user specifies the position at which the component should be placed. Then the exact attributes of the component are selected: length and material for *supports*, type for *connectors*. These choices affect the angles of the pieces and the compatibility of the pieces. The user can use the *Undo* feature to remove components that have been added to the design.

4.6 System Controlled Functions

Some of the functions that go into designing a tower have been left for **SNEAKERS** to handle. This is due to several factors, including the problem of handling all of the user's possible requests. By controlling some of the functionality, **SNEAKERS** is better able to guide the designer in the design task, rather than requiring the program to be an arbitrary drawing program.

Design Phase: The progression of the design from *Abstract* through *Intermediate* to *Detailed* is controlled by **SNEAKERS**. The user must follow this path and cannot skip a step in this progression. The user decides when a level of the design is completed, but **SNEAKERS** changes the interface to reflect the new level. This change sets up the widgets needed to get input at this next level. This guidance helps to focus the design, and does not allow the user to change focus except by restarting the design.

Component Selection: The system will only allow components to be added in a prescribed order. A connector can only be placed on top of a support. If no supports have been placed, or all supports have connectors, no new connectors may be added. Except for the initial supports, which must be placed as prescribed by the abstract tower, all supports must be placed on top of another support which has a connector already in place. This helps keep the design correct, i.e., no unconnected supports on top of each other. It also helps guide the design so that it follows the pattern of the abstract tower.

Component Positioning: The *supports* are placed on top of each other at various angles. The angles to be used are determined automatically by **SNEAKERS** to help follow the abstract tower. This is a four step process:

1. The system determines the ideal angle from the end of the last support to the designed top of that leg of the tower.

2. The ideal angle is compared with the allowed angles as determined by the type of connector being used to connect the two supports.

3. The closest allowed angle to the ideal angle is chosen.

4. The support's end point is determined based on the chosen angle and the support's length, and the support is placed.

The maximum angle allowed is 90°, so that the tower will continue to be designed toward the goal at the top. *Connectors* are always placed on top of supports. *Bracing* is always added between two connectors. The *platform* is placed between the ends of the highest *supports*. Its placement signifies the end of the tower design.

4.7 Aspects Available

One of the most important features of the domain is its need to be analyzed from multiple aspects, in order to demonstrate the effects of concurrent engineering on the design process. This section is a brief overview of the design aspects and their concerns that may impact the tower design. Figure 4.4 shows the agents which are implemented in **SNEAKERS**. The rules which they express are shown in Appendix D. The sections below offer a cursory view of the types of rules that can be written about this domain.

	Advisor	**Critic**	**Suggestor**	**Analyst**	**Evaluator**
Design	■	■	■	■	■
Manufacturing		■	■	■	
Assembly		■	■	■	
Cost		■	■	■	■
Marketing	■				
Safety		■	■	■	■
Disposal	■	■	■	■	
Packaging		■	■		

Figure 4.4: Agents Included in **SNEAKERS**

Design: The design aspect includes rules of thumb and analytical tools that are used strictly for the design process. It is not concerned with any other aspects. In **SNEAKERS**, the design aspect is represented by rules about structure, rules about the order in which to design, rules supporting symmetry, and analytic tools to measure stress and buckling.

Manufacturing: The manufacturing aspect of design contains knowledge of the manufacturing process, and how a design relates to that process. In *SNEAKERS* manufacturing is the connecting of pieces on site and in house. Manufacturing is represented by rules about which connectors can be used with which materials and the time needed to connect the components.

Assembly: The assembly aspect is concerned more with the ability to put the connections together than manufacturing is. In **SNEAKERS**, the height of the tower, number of pieces, and connector type selection are the domain of the assembly aspect.

Cost: The cost aspect is concerned with the ability to estimate costs and reduce costs where possible. It contains rules about tower type, number of pieces, and the costs of certain processes, and how these relate to the overall cost of a design.

Marketing: The marketing aspect of the tower's design is concerned with the ability to sell the tower that has been designed. The rules try to get the designer to design sleek, thin towers, with very few jagged edges and points, and a base as small as the platform. It prefers *I* and *X*-type towers.

Safety: The safety aspect is concerned with designing towers which are safe for human use. This means that towers should be designed which would have a lower likelihood of an accident, or a lower probability of serious injury occurring from an accident. The design aspect has already handled concern over tower stability, but the safety aspect tries to increase the amounts of bracing, for hand holds, have a minimum platform size, and design shorter towers.

Disposal: The disposal aspect is concerned with the end phase of the tower's life cycle. It is concerned with the ability to take the tower apart and also to recycle it. Snaps, followed by bolts are, therefore, the preferred methods of connection. It is also easier to dispose of the tower all at once if it is made of one material.

Packaging: The packaging aspect is concerned with getting the pieces of the tower to the building site. As a result, smaller pieces are preferred, so that they can all be fit into a single truck. If there is very little welding to be done, it could be done at the manufacturing site, and then transported welded. Otherwise, welding equipment also needs to be transported.

4.8 Summary

This chapter detailed the choice of a domain for the Concurrent Engineering Demonstration System which would meet the requirements of being intuitive and yet complex enough to offer several aspects from which to critique a design. The details of the components, both concrete and abstract were discussed, as were the actions involved in designing a tower within the domain. Finally, the various aspects which enable the system's multi-disciplinary view were introduced. This domain had to be decided before **SNEAKERS** could even be designed. The knowledge presented in this chapter was used in designing the overall system. The next chapter presents the software design of **SNEAKERS**, from which the implementation was drawn.

Chapter 5

Design

5.1 Introduction

This chapter presents the design of the **SNEAKERS** system. Individual implementational details are not discussed, only the overall look and feel of the system and its major parts. During the design phase of this project, the exact details of how the system would perform the necessary functions was not considered. The purpose of the design is to provide a logical view to use as a basis for implementing the entire system. Section 5.2 is a view of the interaction between the various parts of the system, which includes logical control and data flows. It also includes a view of the communication between the various agents. Section 5.3 is an overview of the screen, with details about the various parts of the interface. This screen design is based on the drawings made by storyboarding, as described in Section 3.4. The domain choice, discussed separately in Chapter 4, was also an important product of the system design.

5.2 Logical View

There are three views that can be discussed when referring to a logical view of **SNEAKERS**. These are the control flow, the data flow, and the agent communications. Each of these views had to be taken into account simultaneously during the implementation of **SNEAKERS**, but for the purpose of the design, they can be looked at separately. The views presented in the next three sections are the ideals. The changes that had to be made when implementing **SNEAKERS** are shown in Chapter 6.

5.2.1 Control Flow

The control flow is the order in which the computer controls various pieces of the system. The system can be broken down into major subsystems, as shown in Figure 5.1. This figure shows how control passes from one subsystem to the next. For our purposes, the user is considered a subsystem with the ability to add information to the system and affect the control and data flows.

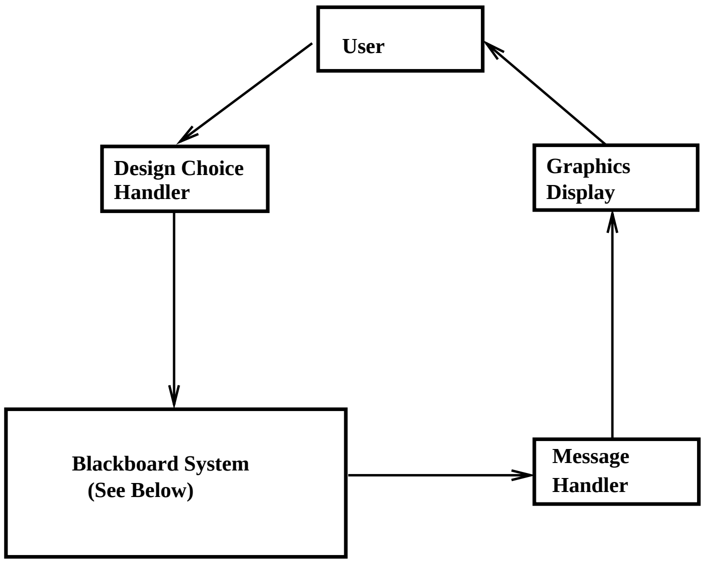


Figure 5.1: Control Flow Diagram

The **User** interacts with the system through a user interface. Design choices made by the user through the interface are handled by different modules, depending upon the choice. For the sake of simplicity, these are all contained in the subsystem labeled **Design Choice Handler** in Figure 5.1. This module translates the choice into a fact that the expert systems use in the **Blackboard System**. The blackboard system is described in greater detail in Section 5.2.3. It is essentially the collection of agents, representing each of the aspects, pulled together in one package. These agents communicate using the blackboard to develop various criticisms, suggestions, and advice, which are then fed back through the user interface to be displayed as messages to the user. The **Message Handler** subsystem handles formatting the output to the user, so that it will be presented properly. After this, the **Graphics Display** subsystem must update the picture of the tower on the screen to reflect the design choice that was made. When all updates are complete, control is returned to the **User**.

This control structure allows the user of the system to have a great amount of influence on the system. If the system provided completely automatic design, the user might be totally cut out of the design process. As this is a demonstration system, it is deemed an appropriate design decision to keep the user in the control flow loop at all times.

5.2.2 Data Flow

A data flow diagram shows how information is passed between subsystems in a system. The data flow of **SNEAKERS** is shown in Figure 5.2. The subsystems are broken down as they were in Section 5.2.1. Direct communication includes local data storage and explicit data passing through arguments. Indirect communication is simply writing a change to the design database. When each subsystem in the loop has control, it can read the change from the design database.

The **User** makes a choice which is given directly to the **Design Choice Handler** subsystem. This system then writes to the **Design Database**. The **Blackboard System** reads from the database, and makes changes or adds more complete information. It also passes messages to the **Message Handler** subsystem, which outputs them to the user. The **Graphics Display** subsystem updates the view of the tower, by reading changes from the **Design Database**, and outputs the new view to the **User**.

The data flow described in this section shows how the various subsystems communicate. This communication scheme is useful in determining the kinds of data that

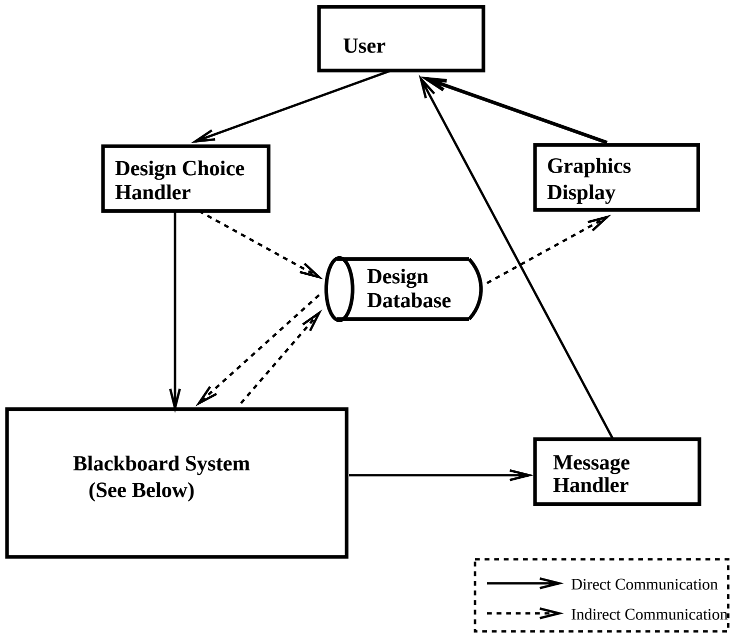


Figure 5.2: Data Flow Diagram

will be available to the subsystems as they are built.

5.2.3 Blackboard and Expert Systems

The blackboard system is designed according to the details given in Section 2.3. A view of the blackboard system is shown in Figure 5.3.

This figure shows the separation among the various expert systems. Though they are separated, they communicate through the blackboard. Each expert system posts information to the blackboard, and reads relevant information (i.e., concerning its aspect) from the blackboard. Each expert system reacts, if possible. The communication between the user and the blackboard is a logical one only. As was shown in Sections 5.2.1 and 5.2.2, the information to and from the user passes through several other subsystems to achieve the required communication. However, this does not change the view of the system that the user gets when interacting with a blackboard. The user is still able to get information from other agents and post information to them, just as if they were using a blackboard.

The blackboard is used to keep the agents separated at the logical level, so that

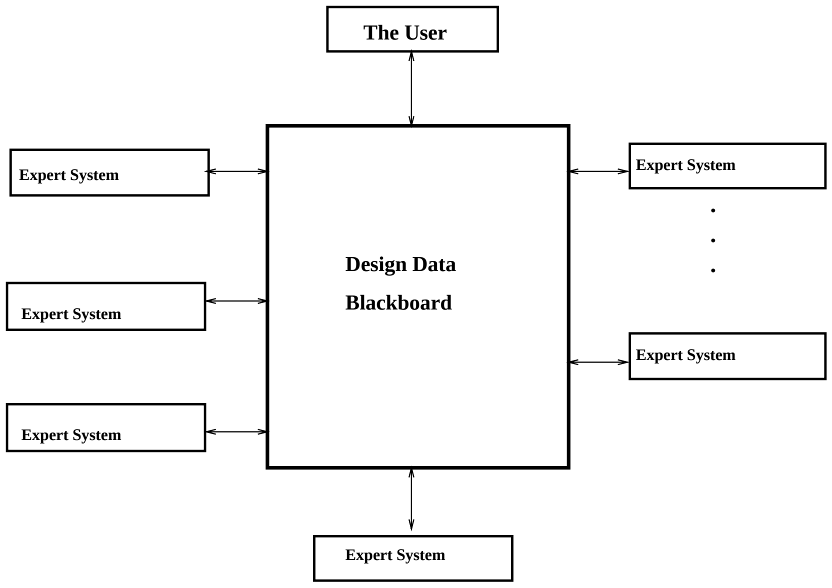


Figure 5.3: Logical View of the Blackboard System

they can represent different “people”. This maintains the view of concurrent engineering as being the combined efforts of many different design agents. By having the agents share information they simulate a group of people checking the design. Information is placed where everyone can see it and each agent is allowed to add any relevant information to it.

Some of the problems which must be tackled with a traditional blackboard approach include information routing to individual agents and agent control. At the design level, it appears that the agents decide what information they need, have access to all information, and are autonomous.

5.3 Screen Layout

The screen design is part of the user interface development, and is a result of the storyboarding described in Section 3.4. The various parts are shown in Figure 5.4. In the sections below, the various pieces are described, along with their required and expected functionality. This provides a good overview of how the user interface development had to be broken down to meet the design requirements.

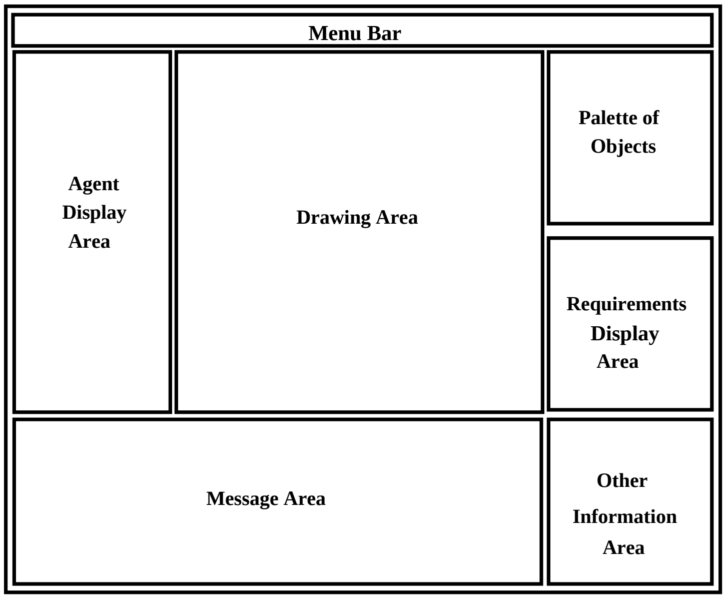


Figure 5.4: Screen Layout

5.3.1 Menu Bar

The **Menu Bar** is located at the top of the screen, and provides access to several functions available throughout the design session. These functions are pulled together into several individual menus and placed in the **Menu Bar** to group them. The functions performed by the system's menus include *File* and *Edit* options, and context sensitive *Help*. The *File* options include starting a *New* design, continuing an *Old* design, *Saving* the current design, and *Quitting* the program. The *Edit* options include *Undoing* the last command and *Changing the requirements*, even in the middle of a design. This **Menu Bar** runs across the whole top of the screen. It is easily accessible and in the position expected for Motif applications. This consistency provides the user with a friendly framework, which should be well received by a user who has had previous exposure to Motif applications.

5.3.2 Drawing Area

The **Drawing Area**, located in the middle of the screen, is the area where all of the graphical display is done. It displays the current tower design and must communicate a great deal of information to the user. Therefore, it needs to be large, occupying the central area of the screen, in order to keep the user's focus. Options available in this area include rotating the tower view to make it easier to see the actual positioning of pieces in the design. It is also in this area that the user specifies the positioning of components during the design.

5.3.3 Agent Display Area

The **Agent Display Area** is located on the left side of the screen. It is the area where messages from each of the agents are accumulated individually. This area also provides a means of signaling when a new comment by a particular agent has been added, by highlighting that agent's buttons. It is used by the user to focus on a single aspect, and to get as much information as possible about that aspect without the interference of information concerning other aspects, such as that given by the **Message Area** (see Section 5.3.4). The messages of all the agents are too many to view simultaneously, so the user must select an individual agent to get the information for a particular aspect. This information can then be used to make the next design decision.

5.3.4 Message Area

The **Message Area** is located at the bottom of the screen. It contains messages from all of the agents, separated by a record of the user's actions. Messages are labeled to correspond to particular design steps by the user, so that the output can be matched to a particular user action, and to maintain focus on what is important for the next design decision being made. The messages are designed to offer alternative views to the user, so that the user can make informed choices for the next design decision. By placing the **Message Area** at the bottom of the screen, new information appears below the most important feature of the design system, the **Drawing Area**, but still occupies a large surface area to imply its importance.

5.3.5 Palette of Objects

The **Palette of Objects** is located in the upper right. It contains the buttons to choose among the objects that the user can place into the current tower design. The user uses this palette to choose the component to add next. The palette reflects the current phase of the design (i.e., abstract, intermediate, or detailed) as put forth in Chapter 4. At the abstract level, abstract tower types are the only objects in the palette. At the intermediate level, various sized towers are selectable. At the detailed level, rods, braces, connectors, and platforms are available. The availability of objects specifically for each level leads the user through the decisions that need to be made to develop a complete design.

5.3.6 Requirements Display Area

The **Requirements Display Area** is located in the center, on the right side of the screen, just below the **Palette of Objects**. It is placed here so that the requirements can be consulted when making a choice from the palette. The requirements need to be constantly displayed so that the user can maintain a focus on the goal of the designed tower. Focus is necessary when the user has to choose between conflicting suggestions from various agents. Placing the requirements in this position on the screen maintains their accessibility, but does not overstate their importance.

5.3.7 Other Information Area

The **Other Information Area** is in the lower right corner, and is used to inform the user about the system's expectations. It gives information about where on the screen to look to make the next design decision. It may inform the user to choose from the file menu, or to select an item from the palette. An expert user could expect to find this information useless, so it is placed in a position which is not intrusive. It might be considered proper to place it near the place on the screen where the information is telling the user to act, but as the user is required to act in all of the other parts of the screen for one reason or another, it is not feasible to place it in any one place to help with this concern. So the optimal choice is to put it in a corner, out of the way of the other areas of the screen.

5.3.8 Pop-up Dialogs

Various **Pop-up Dialogs** appear in the center of the screen when some aspect of the system needs immediate attention. These include the *Requirements Selection* dialog and various warnings about illegal or undesirable actions by the user. By appearing in the center of the screen, they offer an unambiguous suggestion for the user's next step. The user must react to the dialog immediately. After this immediate concern is handled, the pop-up disappears, and the design can continue.

5.3.9 The Whole Picture

The pieces mentioned above are described separately, but must be integrated to show the whole system and to have the desired effect. The **Message Area** and **Agent Display Area** both provide complementary information from the agents. The **Drawing Area**, **Palette of Objects**, and **Other Information Area** guide the user's actions. All of the parts of the screen design have to be integrated properly to make the system function efficiently.

5.4 Summary

The design is a logical view which is used as a guide for implementing a system. Although the actual system may differ in many ways from the design, the design is the initial blueprint. Most of the decisions which have to be made in order to implement the complete system are made during the design phase. A properly designed system is easier to implement, because the communication strategy has been determined, and the problem decomposed. With the problem decomposed, the individual smaller pieces can be implemented. The screen layout is needed to simplify the task of building the pieces of the screen. The interactions between various parts of the screen have to be determined before they can be built individually. The difference between the design and the implementation of **SNEAKERS** is described in the next chapter.

Chapter 6

Implementation

6.1 Introduction

This chapter presents the implementation of the program designed in the previous chapter. Though a logical design is a good guide to building a system, some changes and additional specifications have to be made to turn the design into a working program. Several concessions had to be made, and some logical views had to be reworked when they were implemented.

This chapter gives insight into the actual code written to implement **SNEAKERS** and the problems that arose from changing from a logical design to a physical implementation. The first section shows how the control flows between the various levels of implementation, and then discusses the responsibilities of the individual tools which were used to write the control code for **SNEAKERS**. The next section focuses on the changes to the logical view of the blackboard that were made because of the decision to use CLIPS to implement both the expert systems and the blackboard structure. The changes made were not drastic, and do not diverge from the logical view of the entire system put forth in Chapter 5. One of the strengths of the original design is that it is still visible in the implementation.

6.2 Implementational Control

This section discusses the implemented control flow between the various tools that were used to develop **SNEAKERS**. The three main tools were Motif, which controls the pieces of the user-interface; C, in which all the system control functions, design output, and math functions were written; CLIPS, in which the expert systems were written; and COOL, the object system part of CLIPS, in which the tower database was created. Motif, CLIPS, and COOL were described in detail in Chapter 3.

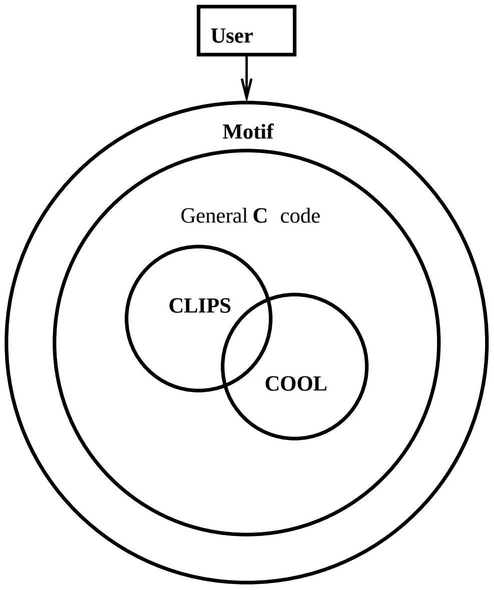


Figure 6.1: Implementational View of Control and Data Flow

The relationship of these three tools to the user is shown in Figure 6.1. The user interacts through Motif widgets to produce design changes, which are handled by C code. These changes are then asserted as facts to CLIPS, which runs the rules and outputs decisions by working back up the control chain to the interface. The C code can also act directly on the objects in the design by communicating with COOL. COOL can also be controlled from within CLIPS. The individual responsibilities of the major tools used are detailed in the subsections below.

Figures 5.1 and 5.2 show the logical version of the control and data flow shown in Figure 6.1. In the implementation, the blocks entitled **Design Choices**, **Messages**, and **Graphics** are all implemented in C. Motif shares the block entitled **User** with the actual user of the system. The **Blackboard System** block is implemented by CLIPS entirely. The **Design Database** in Figure 5.2 is implemented in COOL.

6.2.1 Motif

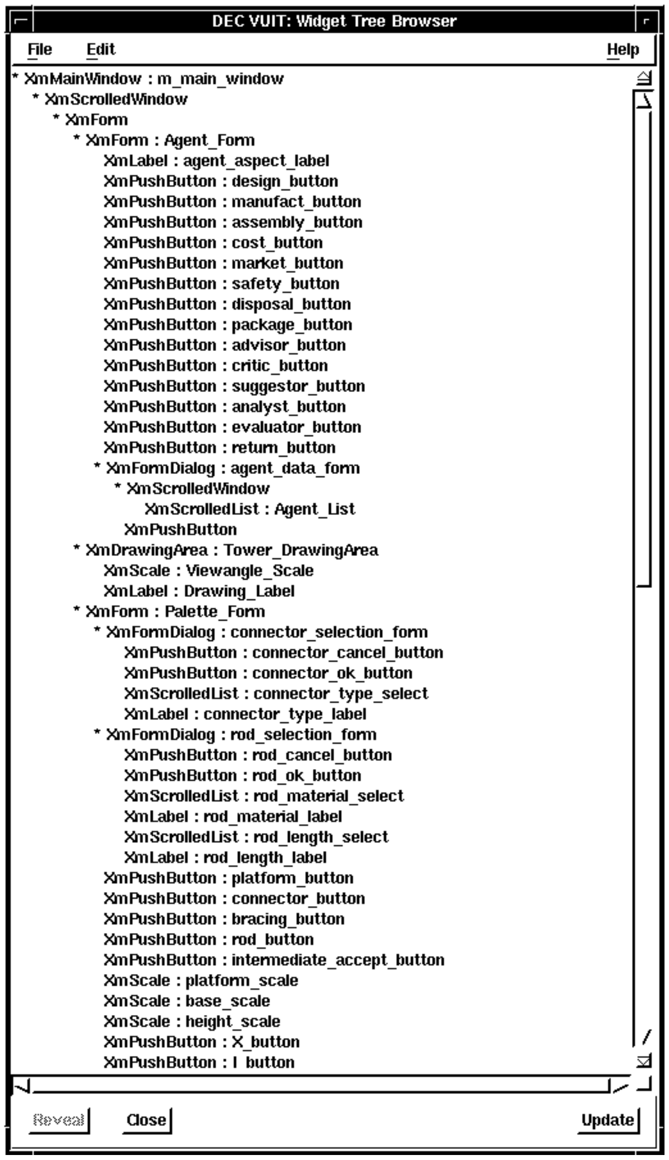
Motif was introduced in Section 3.3. Motif is responsible for controlling the behavior of the widgets which comprise the screen. It is responsible for making a *pushbutton* look and act like the button to which a user of Motif software is accustomed and for allowing the user to select items from lists. It is responsible for all the widgets which are able to offer the user a wide variety of allowable actions and to control how the user's actions affect the rest of the code.

Motif controls the execution of the program by running a loop which waits for events to occur. These events can be button presses, or screen refreshes, or anything that the user can do. A callback is assigned to the events for each widget. These callbacks are written in C code and the call back is executed in full before returning control to the main loop to wait for the next event to be processed.

Each of the individual areas of the screen had to be implemented as an individual widget which would provide the functions that the area was required to support according to the system design. The widgets which were used to implement the screen are shown in Figures 6.2 and 6.3. These screendumps of the widget hierarchy outline are from VUIT's Widget Tree Browser, used to help modify individual pieces of the interface. Several of the buttons are referred to as *Gadgets.* These perform the same functions as widgets, but are faster during run time.

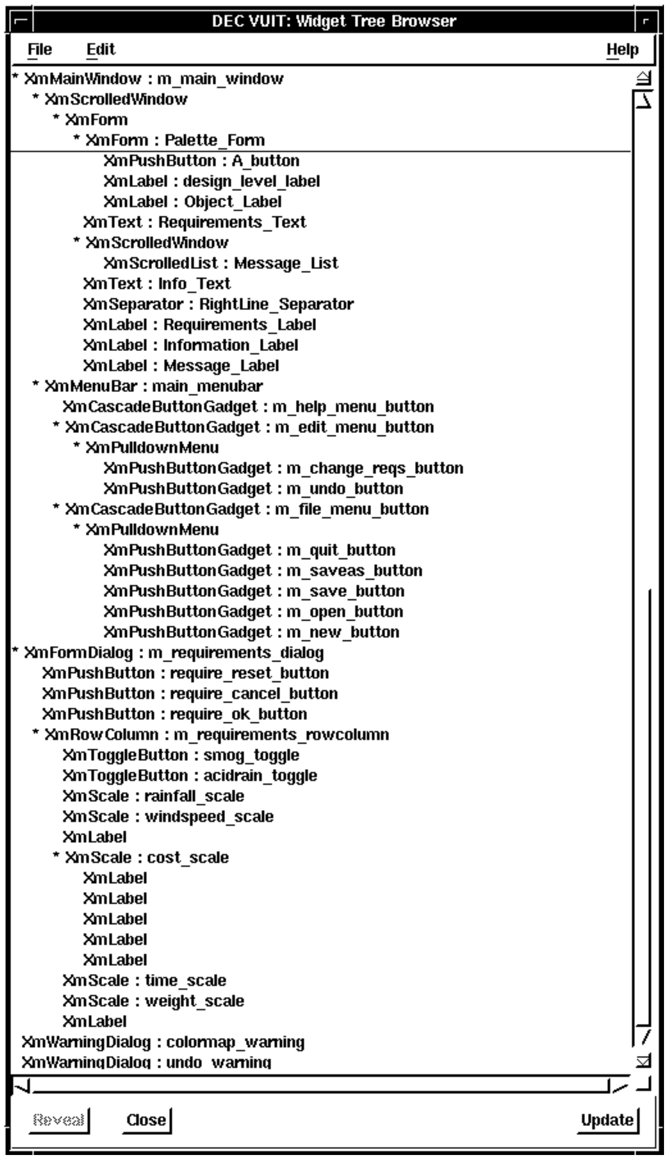
The whole screen is contained in a *main window* widget. The **Menu Bar** is a *menu bar* containing three *cascade buttons* which each contain a *pull-down menu* with several *pushbuttons.* The **Drawing Area** is a *drawing area* widget, which can show the results of calls to X primitives which allow line drawing, and a *scale bar* used to rotate the view.

The **Agent Display Area** is rather complicated. It is a *form* widget containing thirteen *pushbuttons*, representing the aspects and the agents, and a return *pushbutton.* There is also a *pop-up dialog* which displays the agents output in a scrollable *list*



```
* XmMainWindow : m_main_window
  * XmScrolledWindow
    * XmForm
      * XmForm : Agent_Form
          XmLabel : agent_aspect_label
          XmPushButton : design_button
          XmPushButton : manufact_button
          XmPushButton : assembly_button
          XmPushButton : cost_button
          XmPushButton : market_button
          XmPushButton : safety_button
          XmPushButton : disposal_button
          XmPushButton : package_button
          XmPushButton : advisor_button
          XmPushButton : critic_button
          XmPushButton : suggestor_button
          XmPushButton : analyst_button
          XmPushButton : evaluator_button
          XmPushButton : return_button
        * XmFormDialog : agent_data_form
          * XmScrolledWindow
              XmScrolledList : Agent_List
            XmPushButton
      * XmDrawingArea : Tower_DrawingArea
          XmScale : Viewangle_Scale
          XmLabel : Drawing_Label
      * XmForm : Palette_Form
        * XmFormDialog : connector_selection_form
            XmPushButton : connector_cancel_button
            XmPushButton : connector_ok_button
            XmScrolledList : connector_type_select
            XmLabel : connector_type_label
        * XmFormDialog : rod_selection_form
            XmPushButton : rod_cancel_button
            XmPushButton : rod_ok_button
            XmScrolledList : rod_material_select
            XmLabel : rod_material_label
            XmScrolledList : rod_length_select
            XmLabel : rod_length_label
          XmPushButton : platform_button
          XmPushButton : connector_button
          XmPushButton : bracing_button
          XmPushButton : rod_button
          XmPushButton : intermediate_accept_button
          XmScale : platform_scale
          XmScale : base_scale
          XmScale : height_scale
          XmPushButton : X_button
          XmPushButton : I_button
```

Figure 6.2: Widget Hierarchy Outline, screen 1



```
* XmMainWindow : m_main_window
  * XmScrolledWindow
    * XmForm
      * XmForm : Palette_Form
          XmPushButton : A_button
          XmLabel : design_level_label
          XmLabel : Object_Label
        XmText : Requirements_Text
      * XmScrolledWindow
          XmScrolledList : Message_List
        XmText : Info_Text
        XmSeparator : RightLine_Separator
        XmLabel : Requirements_Label
        XmLabel : Information_Label
        XmLabel : Message_Label
  * XmMenuBar : main_menubar
      XmCascadeButtonGadget : m_help_menu_button
    * XmCascadeButtonGadget : m_edit_menu_button
      * XmPulldownMenu
          XmPushButtonGadget : m_change_reqs_button
          XmPushButtonGadget : m_undo_button
    * XmCascadeButtonGadget : m_file_menu_button
      * XmPulldownMenu
          XmPushButtonGadget : m_quit_button
          XmPushButtonGadget : m_saveas_button
          XmPushButtonGadget : m_save_button
          XmPushButtonGadget : m_open_button
          XmPushButtonGadget : m_new_button
* XmFormDialog : m_requirements_dialog
    XmPushButton : require_reset_button
    XmPushButton : require_cancel_button
    XmPushButton : require_ok_button
  * XmRowColumn : m_requirements_rowcolumn
      XmToggleButton : smog_toggle
      XmToggleButton : acidrain_toggle
      XmScale : rainfall_scale
      XmScale : windspeed_scale
      XmLabel
    * XmScale : cost_scale
        XmLabel
        XmLabel
        XmLabel
        XmLabel
        XmLabel
      XmScale : time_scale
      XmScale : weight_scale
      XmLabel
 XmWarningDialog : colormap_warning
 XmWarningDialog : undo_warning
```

Figure 6.3: Widget Hierarchy Outline, screen 2

of messages.

The **Palette of Objects** is a *form* widget containing a combination of *pushbuttons* and scales which appear at the required times. The **Message Area** is a scrollable *list* of messages, just like that used in the **Agent Display Area's** *pop-up dialog*.

The **Requirements Display Area** and *Other Information Area* are simple *text* widgets which display the string they are given. The **Requirements Pop-up Dialog** is a *form dialog* widget containing several *scales* and *toggle buttons*.

This is a general view of the implementation of the interface in Motif. The outline shown in Figures 6.2 and 6.3 are more detailed. For more information about the actual widgets and their functionality, see [Berlage 1991].

6.2.2 General C Code

When a callback is executed, it runs the C code written to interpret the user's action, to control the design choices, and to inform, change and control CLIPS and COOL. The functions of the C code include determining the viewing angles for the graphics routines to convert three dimensional points onto a two dimensional screen. Another function is to keep track of the current level of the design. It also keeps track of various lists to determine where new pieces can be added to the current design.

The greatest responsibility of the C code is as a link between Motif and CLIPS. It must convert the user's action into facts that can be asserted in CLIPS, or into changes made to object instances in COOL. These changes can affect the view of the tower. CLIPS developed agents are able to output advice to the user via Motif, using the C functions.

The C code represents a large piece of the entire objects coding, but lacks the "glamorous" duties that would make it more interesting to study.

6.2.3 CLIPS and COOL

CLIPS, as was mentioned in Section 3.2, was used to build the expert systems which represent the various types of agents and the various aspects of the design. COOL, CLIPS Object Oriented Language, was used to act as an object-oriented design database. Some knowledge about the domain of *TinkerToys*^{TM} and the towers that can be built using them is encoded there.

All of the rules written in CLIPS operate by attempting to match facts in the condition part of each rule to those in the fact-list, and then implementing an action when those facts are matched. The control view of CLIPS is that the agents are all read into the **SNEAKERS** environment when it is started and that CLIPS maintains a single fact-list, which is used by all the agents. The C code described above and the CLIPS code described here, both assert facts in the form:

```
(designfact (time ?value) (level ?value) (aspect ?value)
            (agent ?value) (information ?value))
```

All facts used in the expert systems are in this form, whether they are in the condition part of a rule or the asserted facts that constitute deductions made by an agent.

Each of the fields contained in parentheses describes something about the design decision just made. The **time** field is the number of user actions made since the beginning of the design. The **level** field is the level of abstraction at which the design decision was made, eg. *ABSTRACT*. The **aspect** field is the name of the aspect to which the agent who posted the fact belongs, eg. *DESIGN*. The **agent** field is the agent type who posted the fact, eg. *ADVISOR*. The **information** field is the most relevant field, because it is the information the agent wishes to communicate, eg. *New design started*.

The agents are written so that the right combination of *designfacts* will cause them to either post another *designfact* or inform the user of some information. All agents have a chance to react to every *designfact* that is posted.

The COOL database contains all of the objects that can exist in a tower design. These objects are shown in Appendix C. Each object has a number of attributes, which can be values or other objects. These objects in the database act as classes, i.e. they represent sets. When a design is performed, instances of these classes are created which have specific values for their attributes, and which can then be reasoned about. By referencing the instances of these objects, the agents can get a detailed view of the current state of the tower's design. The instances are also referenced by the general C code to display the graphical view of the tower. Attributes of the instances are used to progress through the design. For example, the system knows, from the class information, that a tower is composed of supports, so it will continue through the design until all the supports needed have been specified as attributes of the tower.

6.3 Implementational View of Blackboard

This section details the difference between the logical view of a blackboard structure for controlling the agents, and the implementational view of an actual set of agents written in CLIPS. Figure 5.3 shows the abstract view of this system, while Figure 6.4 shows a view which is more concerned with implementational issues.

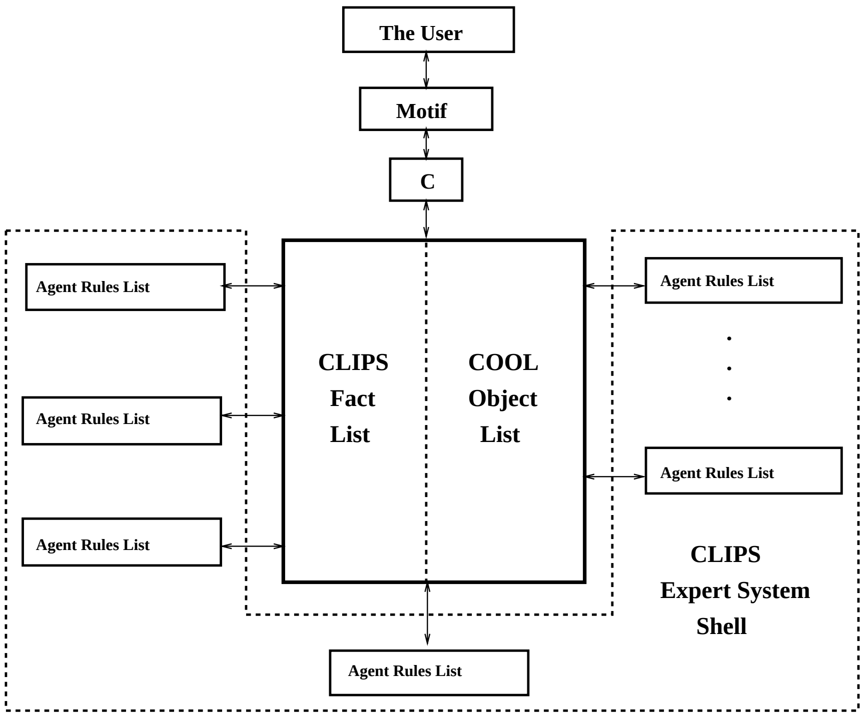


Figure 6.4: Implementational View of the Blackboard System

In Figure 6.4, the blackboard is shown to be a combination of the CLIPS fact list and the COOL object list. These lists are available to the user through the C program and Motif, and to each of the sets of agent rules. As such, each of the agents has the access to whatever information it may find relevant. The agent rules are separated into different files and read into the CLIPS shell on program execution. So there are seemingly separate agents, reacting to the same information and providing each other with information from across the different aspects.

There are ways in which this implementation is not consistent with the blackboard model:

1. The agents act differently from the way the user acts. They react to the user, and continue to act until all agents have had a chance to react to the user's action. In this sense, the user is treated as a different kind of agent, and this is inconsistent with the ideal model. Ideally, the user would act just like any other agent, without the actions of the others revolving around the user's actions.

2. The execution of CLIPS is controlled from outside of CLIPS. The C code has to be invoked to send a "RunCLIPS" command to execute the rules. If the system were a blackboard it would inform the agents when a new fact appeared, and they would be able to react independent of the user's actions. CLIPS would be running constantly, and not have to be called upon to run only when new facts have been asserted.

3. Even though the agents seem to be separated, the rules are still read in and organized together and indexed by CLIPS. A real blackboard system would be responsible for indexing them and keeping them separate.

Despite these failings, the user's view of **SNEAKERS** is still that of a blackboard structure with a number of independent agents, which can be heeded individually or collectively.

6.4 Summary

This chapter examined some of the details of the implementation of the design presented in Chapter 5. There were some changes that had to be made, but the design remained mostly intact. This chapter showed some details about Motif, CLIPS, and the C code that holds them together. A view of the control structure for the expert system agents was given with an analysis of the similarities and differences between the implementational and design views of the blackboard structure. The next chapter is an evaluation of the system, conclusions, and suggestions for future work.

Chapter 7

Conclusions

7.1 Introduction

This chapter discusses how well this thesis meets the goals described in Section 1.2. It also discusses the results of an evaluation performed by graduate students at Worcester Polytechnic Institute. These two points are covered in Section 7.2, below. Section 7.3 discusses some other work which could be added to this thesis in order to extend the idea of developing educational software to teach about Concurrent Engineering.

7.2 Evaluation

SNEAKERS cannot unequivocally be stated to be a success, because only time will tell if it truly meets its primary goal of educating managers and engineers in the CE design methodology. However, there are some more immediate evaluations which were performed, against which **SNEAKERS** can be said to have succeeded.

The first measure of evaluation is in how well **SNEAKERS** matches the list of ingredients which were first introduced in Section 1.3.1. The purpose of this thesis, in part, was to develop this list and then to create a Concurrent Engineering tool which would demonstrate as many of these ingredients as possible, while always keeping in mind the educational nature of the project. The ingredients are listed again below, along with a short description of how **SNEAKERS** meets the needs of each.

1. **Design Agents** – There are 26 independent agents in **SNEAKERS**.

2. **Multi-disciplinary Goal and Specification Representation** – Several requirements and results are multi-disciplinary, but this would be considered marginally included.

3. A **Catalog** – The possible components are given in a palette during the design.

4. A **Design Database** – The COOL object list keeps the current design representation.

5. An **Accumulated Database** – Each of the agents can be accessed individually to see messages which that agent has provided to the user.

6. **Shareability of Design Information** – The blackboard structure allows all the agents to know everything about the design.

7. **Communication** – All agents in CLIPS communicate and they also communicate with the user, but there is no support for communication with other people.

8. A **Manager** – The user serves as the manager of the tower design.

9. A **Design History** – The scrollable message area has a complete listing of user actions together with agents' reactions.

10. A **Checklist** – The other information window attempts to serve this purpose, but this, too, may be considered marginal.

11. A **Standard Interface** – Motif provides a common interface that is familiar to many users, and the system itself has only a single interface which tries to remain consistent throughout the system.

12. **Virtual Collocation of People** – As this is a single-user system, this ingredient is not satisfied by this system.

Of the 12 ingredients listed, 8 are included outright, 3 are marginally included, and only 1 is not included at all.

Besides this evaluation, **SNEAKERS** was also evaluated by five graduate students at WPI, who all have experience in industry or with Concurrent Engineering. They were Sundar Victor, Peter McCann, Eric Rasmussen, Jingwen Liu, and Kathy Urbanowicz.

Sundar Victor, who is working toward his master's degree in Computer Science, was a member of WPI's Intelligent, Interactive, and Integrated Design (I3D) team which did Army funded CE research. He has also worked in industry.

Peter McCann holds a bachelor's degree in Computer Science, has worked for five years in industry, and is now pursuing his master's degree in Manufacturing Engineering. He was also a member of the I3D team.

Eric Rasmussen, a third member of the I3D team, is pursuing his master's degree in Manufacturing Engineering and has had significant computer experience.

Jingwen Liu has nearly completed his PhD in Computer Science, with a concentration on Artificial Intelligence in Design.

Kathy Urbanowicz has worked in industry, and has recently returned to school to work full-time on her master's degree in Computer Science.

Each of these people read the user's guide, and were able to ask questions for clarification. Their questions and comments on the user's guide helped to improve that document, and make it less ambiguous. They were also given a short introduction to the domain. They had all seen at least one presentation on **SNEAKERS**, so they were familiar with the project.

They were allowed to run the system alone, and made comments in an informal setting. Their comments were noted. Some of the comments suggested reasonable changes to the system, including changing the wording of some of the expert systems, changing the messages in the other information area, and guarding against some user actions which could cause errors in the system, but which had not previously been trapped. These comments are considered reasonable because either they could be satisfied with less than an hour of effort, or the consequences of leaving them were detrimental to the goal of the system. All of the reasonable comments were immediately coded into the system in under ten minutes and the evaluator was able to see the system with the change. Some of the more unreasonable comments included forcing the cursor to onto a particular palette choice when an agent suggests or advises that that choice be made and increasing the control over the user's actions.

Aside from these suggested changes, the other comments were that the interface was very helpful for the task; that the rules made sense to the users; that it was easy to use and that there was no need for someone to sit nearby to tell the user what to do; and that the highlighting of the agent's buttons as they make new comments really does convey the needs of other aspects in the design. As a consequence of using **SNEAKERS**, the members of the I3D team want to build a similar interface for their CE support system.

From these comments, it appears that **SNEAKERS** has lived up to the goals of being intuitive, easy to use, and educational. It is not perfect, but its flaws are outweighed by the usability of the system.

7.3 Future Work

The future of this project will be determined by its usefulness to the Competitive Product Development Institute (CPDI) at Digital Equipment Corporation. However, even if **SNEAKERS** meets the needs of CPDI, it still has room for improvement.

All of the unimplemented options in **SNEAKERS** can be implemented. There is the potential for forty agents in the system as it stands, but only twenty-six are included. Also, the agent types and aspects are not all inclusive, so even more types of agents could be added. The rule sets for each agent could be expanded. Some agents have only one rule, while others have ten.

There is no geometric reasoning in the system. For example two supports are allowed to pass through each other. Bracing is handled as an arbitrary length rod of undecided material. Bracing could be changed to be a particular rod length and material type, just as supports are, and the tower designed to accommodate the allowable bracings as well. All of this would make a better system, but would not greatly increase its usefulness in educating managers and engineers.

If this interface were to be used as a realistic general CE support system, then there would have to be extensive changes. Areas that would have to be added include: negotiation, database management, agent communication, and support for multiple users.

Negotiation between agents with conflicting suggestions is a difficult problem, but it would be necessary in a general system to weed out some of the comments that the user sees, through negotiation. Otherwise, the number of comments would be unmanageable. Precedence for this work is also given in [Sriram et al 1989].

More work would have to be done to expand the COOL representation of the database. There is no support for version control and time dependence in the databases. Research into these areas of database management can be found in [Katz & Chang 1987], [Katz 1990], and [Snodgrass 1990].

A more realistic system would need to take into account the research reported in [Daley & Fotta 1991] and [Fotta & Daley 1991] which are concerned with communication among agents with varied backgrounds. The domain in **SNEAKERS** is simple enough that the agents can communicate with an easily managed vocabulary.

A full-blown system would have to use multiple processes on multiple machines and be able to support multiple users. It should have a means of keeping all users informed of the agents' comments and allow communication between the users.

There are many ways in which this project could be expanded. **SNEAKERS** itself is a fairly complete system, and needs no major improvements to fulfill its task. But it is not a general CE support system, and therefore, would have to be radically altered in order to fill that role.

7.4 Summary

This thesis has shown that there is a large amount of work being done in Concurrent Engineering, and that there is a need to bring that research into practical use. The Concurrent Engineering Demonstration System, **SNEAKERS** is one foray into this area. Though much has been accomplished there is still room for expansion and improvement. CE is a growing area of interest for large companies, and the work done on this project is an attempt to increase that interest among managers and engineers alike.

Bibliography

[Altamuro 1991]	V.M. Altamuro. "Strategic Product Design". *Concurrent Engineering*, vol. 1, no. 2, 1991, pp. 39-45.
[Arpino & Groppetti 1988]	F. Arpino, R. Groppetti. "ASSYST: A consultation system for the integration of product and assembly system design", 1988. Reprinted in *Design For Manufacture*, J. Corbett, M. Dooner, J. Meleka, C. Pym, eds., Addison-Wesley, Reading, MA, 1991, pp. 246-257.
[Barber et al 1992]	J. Barber, S. Bhatta, A. Goel, M. Jacobson, M. Pearce, L. Penberthy, M. Shankas, R. Simpson, E. Stroulia. "AskJef: Integration of Case-based and Multimedia Technologies for Interface Design Support." *AI in Design '91*, J.S. Gero, ed., Butterworth-Heinemann, Ltd., Boston, 1991, pp. 457-475.
[Bedworth et al 1991]	D.D. Bedworth, M.R. Henderson, P.M. Wolfe. *Computer-Integrated Design And Manufacturing*, McGraw-Hill, New York, NY, 1991, pp. 134-176.
[Berlage 1991]	T. Berlage. *OSF/Motif: Concepts and Programming*. Addison-Wesley, Reading, MA, 1991.
[Biegel & Pecht 1991]	P. Biegel, M.G. Pecht. "Design Trade-Offs Made Easy". *Concurrent Engineering*, vol. 1, no. 3, 1991, pp. 29-40.
[Blackboard 1991]	Blackboard Technology Group, Inc. "The Blackboard Problem-Solving Approach". *AI Review*, Summer 1991.
[Bloor et al 1988]	M.S. Bloor, A. de Pennington, S.B. Harris, D. Holdsworth, A. McKay, and N.K. Shaw. "Towards integrated design and manufacturing", 1988. Reprinted in *Design For Manufacture*, J. Corbett, M. Dooner, J. Meleka, C. Pym, eds., Addison-Wesley, Reading, MA, 1991, pp. 258-269.

[Boothroyd & Dewhurst 1988] G. Boothroyd, P. Dewhurst. "Product design for manufacture and assembly", 1988. Reprinted in *Design For Manufacture*, J. Corbett, M. Dooner, J. Meleka, C. Pym, eds., Addison-Wesley, Reading, MA, 1991, pp. 165-173.

[Brown et al 1989] A.D. Brown, P.R. Hale, J. Parnaby. "An integrated approach to quality engineering in support of design for manufacture", 1989. Reprinted in *Design For Manufacture*, J. Corbett, M. Dooner, J. Meleka, C. Pym, eds., Addison-Wesley, Reading, MA, 1991, pp. 146-164.

[Brown 1992a] D.C. Brown. "Design". *Encyclopedia of AI*, 2nd edn., S. Shapiro, ed., J. Wiley, New York, NY, 1992.

[Brown 1992b] D.C. Brown. "Routineness Revisited". *Mechanical Design: Theory and Methodology*, M. Waldron, K. Waldron, eds., Springer-Verlag, New York, to appear in 1992.

[Campbell et al 1991] I.C. Campbell, K.J. Luczynski, and S.K. Hood. "Putting Knowledge-Based Concepts to Work for Mechanical Design". *Innovative Applications of Artificial Intelligence*, R. Smith, C. Scott, eds., AAAI Press, Menlo Park, CA, 1991.

[Corbett 1987] J. Corbett. "How design can boost profit", 1987. Reprinted in *Design For Manufacture*, J. Corbett, M. Dooner, J. Meleka, C. Pym, eds., Addison-Wesley, Reading, MA, 1991, pp. 49-55.

[Cunningham & Dixon 1988] J.J. Cunningham, J.R. Dixon. "Designing with Features: The Origin of Features". *Computers in Engineering*, ASME, 1988, pp. 237-243.

[Daley & Fotta 1991] R. Daley, M. Fotta. "Applying PCT to Product Development Teams", Personal Construct Psychology Conference, Albany, NY, 1991.

[Digital 1990] Digital. *DEC VUIT: User's Guide*. Digital Equipment Corporation, Maynard, MA, 1990.

[Dimancesco 1991] D. Dimancesco. "The Rugby Metaphor". *Concurrent Engineering*, vol. 1, no. 5, 1991, pp. 44-46.

[Douglas & Brown 1992] R.E. Douglas, Jr., D.C. Brown. "Concurrent Accumulation of Knowledge: A View of Concurrent Engineering". *The Handbook of Concurrent Engineering*, H.R. Parsaei, W.G. Sullivan, eds., Chapman and Hall, London, England, to appear in 1992.

[Durfee et al 1989] E.H. Durfee, V.R. Lesser, D.D. Corkill. "Cooperative Distributed Problem Solving". *The Handbook of AI*, vol. 4, chp. XVII, A. Barr, P.R. Cohen, E.A. Feigenbaum, eds., Addison-Wesley, Reading, MA, 1989.

[Finger et al 1988] S. Finger, M.S. Fox, D. Navinchandra, F.B. Prinz, J.R. Rinderle. "Extended Abstract for Design Fusion: A Product Life-Cycle View for Engineering Designs". IFIP WG 5.2, IntCAD, September, 1988.

[Fischer et al 1989] G. Fischer, R. McCall, A. Morch. "Design Environments for Constructive and Argumentative Design". *CHI'89 Proceedings*, 1989, pp. 269-275.

[Fischer et al 1990] G. Fischer, A.C. Lemke, T. Mastaglio, A.I. Morch. "Using Critics to Empower Users". *CHI'90 Proceedings*, 1990, pp. 337-347.

[Fotta & Daley 1991] M.E. Fotta, R.A. Daley. "Using Entity-attribute Grids to Identify Intra-Team Communication Problems". CALS/CE conference, Washington, DC, 1991.

[Gesner et al 1991] S.M. Gesner, B.G. Nickerson, L. Tompkins. "ACE - A Knowledge-Based Assistant For Cost Estimating". 4th UNB AI symposium, Fredericton, NB, Canada, September, 1991.

[Giarratano 1991a] J.C. Giarratano. *CLIPS User's Guide: Volume 1, Rules*, CLIPS version 5.0. Software Technology Branch, Lyndon B. Johnson Space Center, Houston, TX, 1991.

[Giarratano 1991b] J.C. Giarratano. *CLIPS User's Guide: Volume 2, Objects*, CLIPS version 5.0. Software Technology Branch, Lyndon B. Johnson Space Center, Houston, TX, 1991.

[Giarratano 1991c] J.C. Giarratano. *CLIPS Reference Manual: Volume I, Basic Programming Guide*, CLIPS version 5.0. Software Technology Branch, Lyndon B. Johnson Space Center, Houston, TX, 1991.

[Giarratano 1991d] J.C. Giarratano. *CLIPS Reference Manual: Volume II, Advanced Programming Guide*, CLIPS version 5.0. Software Technology Branch, Lyndon B. Johnson Space Center, Houston, TX, 1991.

[Giarratano 1991e] J.C. Giarratano. *CLIPS Reference Manual: Volume III, Utilities and Interface Guide*, CLIPS version 5.0. Software Technology Branch, Lyndon B. Johnson Space Center, Houston, TX, 1991.

[Hill 1990] F.S. Hill, Jr. *Computer Graphics*. Macmillan Publishing Company, New York, NY, 1990, pp. 60-102, 213-266, 305-383.

[Hurley 1987] J.F. Hurley. *Calculus*. Wadsworth Publishing Company, Belmont, CA, 1987, pp. 628-682.

[Jagannathan et al 1991] V. Jagannathan, K.J. Cleetus, R. Kannan, A.S. Matsumoto, J.W. Lewis. "Computer Support for Concurrent Engineering". *Concurrent Engineering*, vol. 1, no. 5, 1991, pp. 14-30.

[Katz & Chang 1987] R.H. Katz and E. Chang. "Managing Change in a Computer-Aided Design Database". *Proceediings of the 13th VLDB Conference*, Brighton, England, 1987.

[Katz 1990] R.H. Katz. "Towards a Unified Framework for Version Modeling in Engineering Databases". *ACM Computing Surveys*, vol. 22, no. 4, 1990, pp. 375-408.

[Kim et al 1991] S.H. Kim, S. Hom, and S. Parthasarathy. "Design and manufacturing advisor for turbine disk", 1988. Reprinted in *Design For Manufacture*, J. Corbett, M. Dooner, J. Meleka, C. Pym, eds., Addison-Wesley, Reading, MA, 1991, pp. 215-230.

[Kott et al 1991] A. Kott, C. Kollar, A. Cederquist. "The Role of Product Modeling in Concurrent Engineering Environments". Submitted article, *Journal of Systems Automation*, 1991.

[Kroll et al 1988] E. Kroll, E. Lenz, J.R. Wolberg. "A knowledge-based solution to the design for assembly problem", 1988. Reprinted in *Design For Manufacture*, J. Corbett, M. Dooner, J. Meleka, C. Pym, eds., Addison-Wesley, Reading, MA, 1991, pp. 203-214.

[Lemke & Fischer 1990] A.C. Lemke, G. Fischer. "A Cooperative Problem Solving System for User Interface Design". *AAAI-90, Proceedings of the 8th National Conference on Artificial Intelligence*, MIT Press, Cambridge, MA, 1990, pp. 479-484.

[Lemon et al 1990] J.R. Lemon, W.E. Dacey, E.J. Carl. "Concurrent Product/Process Development". International TechneGroup Incorporated Technical Report, 1990.

[Lewis 1991] M.H. Lewis. "Concurrent Engineering at Loral Defense Systems" *Concurrent Engineering*, vol. 1, no. 2, 1991, pp. 5-10.

[Liu 1991] S.X. Liu. *A Knowledge Based User Interface For A Tunneling Simulation System.* Unpublished Master's Thesis, WPI, 1991, pp. 58-71.

[Londoño et al 1989] F. Londoño, K.J. Cleetus, Y.V. Reddy. "A Blackboard Scheme for Cooperative Problem-Solving by Human Experts". In *Proceedings of MIT-JSME Workshop on Cooperative Product Development*, D. Sriram, R. Logduer, S. Fukuda, eds., New York, NY, 1989.

[Medler 1991] D.K. Medler. "Stereolithography and Concurrent Engineering: A Powerful Combination". *Concurrent Engineering*, vol. 1, no. 4, 1991, pp. 5-10.

[Myers et al 1991] L. Myers, J. Pohl, A. Chapman. "The ICADS Expert Design Advisor: Concepts and Directions". *AI in Design '91*, J.S. Gero, ed., Butterworth-Heinemann, Ltd., Boston, 1991, pp. 897-920.

[Nii 1986a] H.P. Nii. "The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures". *The AI Magazine*, Summer 1986, pp. 38-53.

[Nii 1986b] H.P. Nii. "Blackboard Application Systems and a Knowledge Engineering Perspective". *The AI Magazine*, August 1986, pp. 82-107.

[Roberts 1991] R.S. Roberts. "Simultaneous Engineering and DFM at Cadillac". *Concurrent Engineering*, vol. 1, no. 1, 1991, pp. 29-36.

[Shneiderman 1992] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 2nd edition. Addison-Wesley Publishing Company, Reading, MA, 1992.

[Silverman & Mezher 1992] B.G. Silverman, T.M. Mezher. "Expert Critics in Engineering Design: Lessons Learned and Research Needs". *AI Magazine*, Spring, 1992, pp. 45-62.

[Snodgrass 1990] R. Snodgrass. "Temporal Databases: Status and Research Directions". *SIGMOD RECORD*, vol.19, No.4, December, 1990.

[Sobolewski 1991] M. Sobolewski. "Integration of Declarative and Procedural Knowledge in Engineering Applications". The World Congress on Expert Systems, Orlando, FL, 1991.

[Sriram et al 1989] D. Sriram, R.D. Logcher, N. Groleau, J. Cherneff. "DICE: An Object Oriented Programming Environment For Cooperative Engineering Design". Technical Report No: IESL-89-03, Mass. Institute of Technology, Cambridge, MA, 1989.

[Stoll 1986] H.W. Stoll. "Design for manufacture: an overview", 1986. Reprinted in *Design For Manufacture*, J. Corbett, M. Dooner, J. Meleka, C. Pym, eds., Addison-Wesley, Reading, MA, 1991, pp. 107-129.

[Subramanian et al 1990] E. Subramanian, G. Podnar, A. Westerberg. "A Shared Computational Environment for Concurrent Engineering". Engineering Design Research Center Technical Report, Carnegie Mellon Univ., Pittsburgh, PA, 1990.

[Swift 1989] K.G. Swift, M.E. Uddin, M.G. Limage, and M.S. Bielby. "Production-oriented design: a knowledge-based approach", 1989. Reprinted in *Design For Manufacture*, J. Corbett, M. Dooner, J. Meleka, C. Pym, eds., Addison-Wesley, Reading, MA, 1991, pp. 231-245.

[Whitney 1988] D.E. Whitney. "Manufacturing by design", 1988. Reprinted in *Design For Manufacture*, J. Corbett, M. Dooner, J. Meleka, C. Pym, eds., Addison-Wesley, Reading, MA, 1991, pp. 37-48.

[Wilson 1991] C.C. Wilson. "Potential Pitfalls of Concurrent Engineering". *Concurrent Engineering*, vol. 1, no. 1, 1991, pp. 37-43.

[Wong et al 1990] A. Wong, D. Sriram, R. Logcher. "User Interfaces for Cooperative Product Development". *Proceedings of the Second National Symposium on Concurrent Engineering*, West Virginia University, 1990.

Appendix A

User's Guide for SNEAKERS

SNEAKERS is a single-user, mouse-driven, windowed system, which is fairly simple to learn and use. To start the system, type

```
sneakers
```

at the command line. The system will start and create the main window.

The goal of the system is to design a tower by adding components similar to *TinkerToys*^{TM} to an abstract design. You start by giving the requirements for the design. During the design, the intelligent agents incorporated in the system provide information and feedback to help you with the design. The advice they give can be heeded or ignored as you see fit. The design is evaluated upon completion for its ability to meet the requirements.

A.1 Screen Layout

Below is a brief overview of the different areas of the screen. The user's options in each of these areas are detailed in Section A.2. Figure A.1 shows how the areas of the screen are placed.

Menu Bar: The **Menu Bar** is located at the top of the screen, and provides access to several functions available throughout the design session.

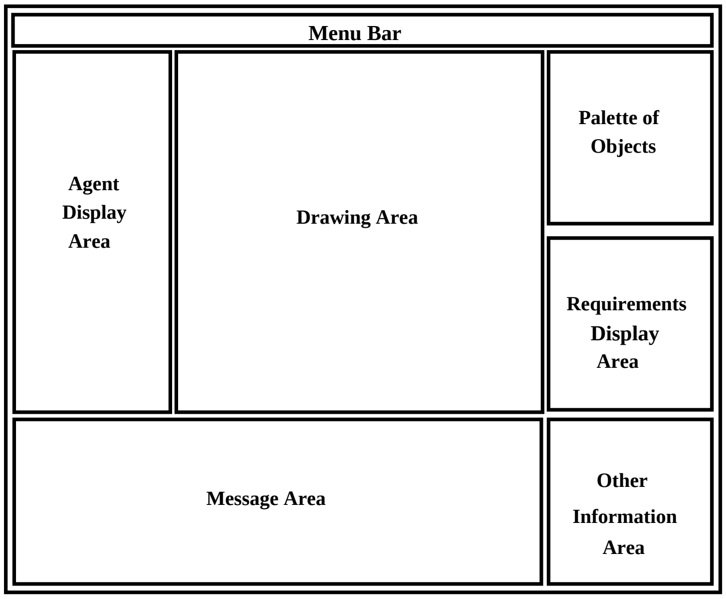


Figure A.1: Screen Layout

Drawing Area: The **Drawing Area**, located in the middle of the screen, is the area where all of the graphical display is done. It displays the current tower design.

Agent Display Area: The **Agent Display Area** is located on the left side of the screen. It is the area where messages from each of the agents are accumulated individually. This area also provides a means of signaling when a new comment by a particular agent has been added, by highlighting that agent's buttons.

Message Area: The **Message Area** is located at the bottom of the screen. It contains messages from all of the agents, separated by a record of the user's actions.

Palette of Objects: The **Palette of Objects** is located in the upper right. It contains buttons to choose among the objects that the user can place into the current tower design.

Requirements Display Area: The **Requirements Display Area** is located in the center, on the right side of the screen, just below the **Palette of Objects**. It is placed here so that the requirements can be consulted when making a choice from the palette.

Other Information Area: The **Other Information Area** is in the lower right corner, and gives information about where on the screen to look to make the next design decision. It may inform the user to choose from the file menu, or to select an item from the palette.

Pop-up Dialogs: Various **Pop-up Dialogs** appear in the center of the screen when some aspect of the system needs immediate attention. These include the **Requirements Selection** dialog and various warnings about illegal or undesirable actions by the user.

A.2 User Options

All interactions with the system can be performed with the mouse. Choosing an option requires that you simply click the left mouse button once. The right mouse buttons allow you to cancel the placement of supports, connectors, and bracing during *Detailed Design*. The available options are summarized below. Some of these options have not been implemented.

- **Menu Bar** options
    - **File** menu options
        - **New** – Starts a new design session.
        - **Old** – Reads in a previously saved design session. (Not implemented.)
        - **Save** – Saves a design session. (Not implemented.)
        - **Save As** – Saves a design session to a user named file. (Not implemented.)
        - **Quit** – Ends the design session.
    - **Edit** menu options

        - **Undo** – Removes the last user action and wipes out all consequences of that action. (**Note:** Available only during *Detailed Design*.)
        - **Change Requirements** – Changes the initial requirements for a tower after the start of a design session. (Not implemented.)
    - **Help** – Gives context-sensitive help.

- *Abstract Design* options – Choose the abstract tower type to follow in the rest of the design.
    - **A style tower**
    - **I style tower**
    - **X style tower**

- *Intermediate Design* options – Select desired values for the size of the tower.
    - **Desired Height**
    - **Desired Base Size**
    - **Desired Platform Size**
    - **Accept Prototype** – Moves design to *Detailed Design* phase.

- *Detailed Design* options
    - **Place Support** – You then pick a position in the **Drawing Area**. Valid positions are at the bottom of abstract support guides and on connectors. Finally you must select the length and material of the support.
    - **Place Connector** – You then pick a position which is at the top of a support. Finally you must select the type of connector.
    - **Place Bracing** – You then pick two connectors from the **Drawing Area** to which to attach the bracing. (**Note:** Available only after all *supports* and *connectors* are placed.)
    - **Place Platform** – This also completes the design, and ends the session.

(**Note:** Available only after all *supports* and *connectors* are placed.)

- Other options – These are the other possible user actions.
    - **Agent Display Area** options
        - Choose an aspect – Looks at a particular aspect.
        - Choose an agent type – Shows messages from that agent in the chosen aspect. (**Note:** Available only after an aspect is chosen.)
        - **Return** – Moves from looking at an agent to looking at an aspect, and from looking at an aspect to looking at all aspects.
    - **Drawing Area** options
        - Move the slider – Rotates the tower.
    - **Message Area** options
        - Move the slider – Scrolls the messages list.

When finished with the design, choose **Quit**. The window will close and a final report is generated in the file *sneakers.rpt*.

I hope you enjoy **SNEAKERS**.

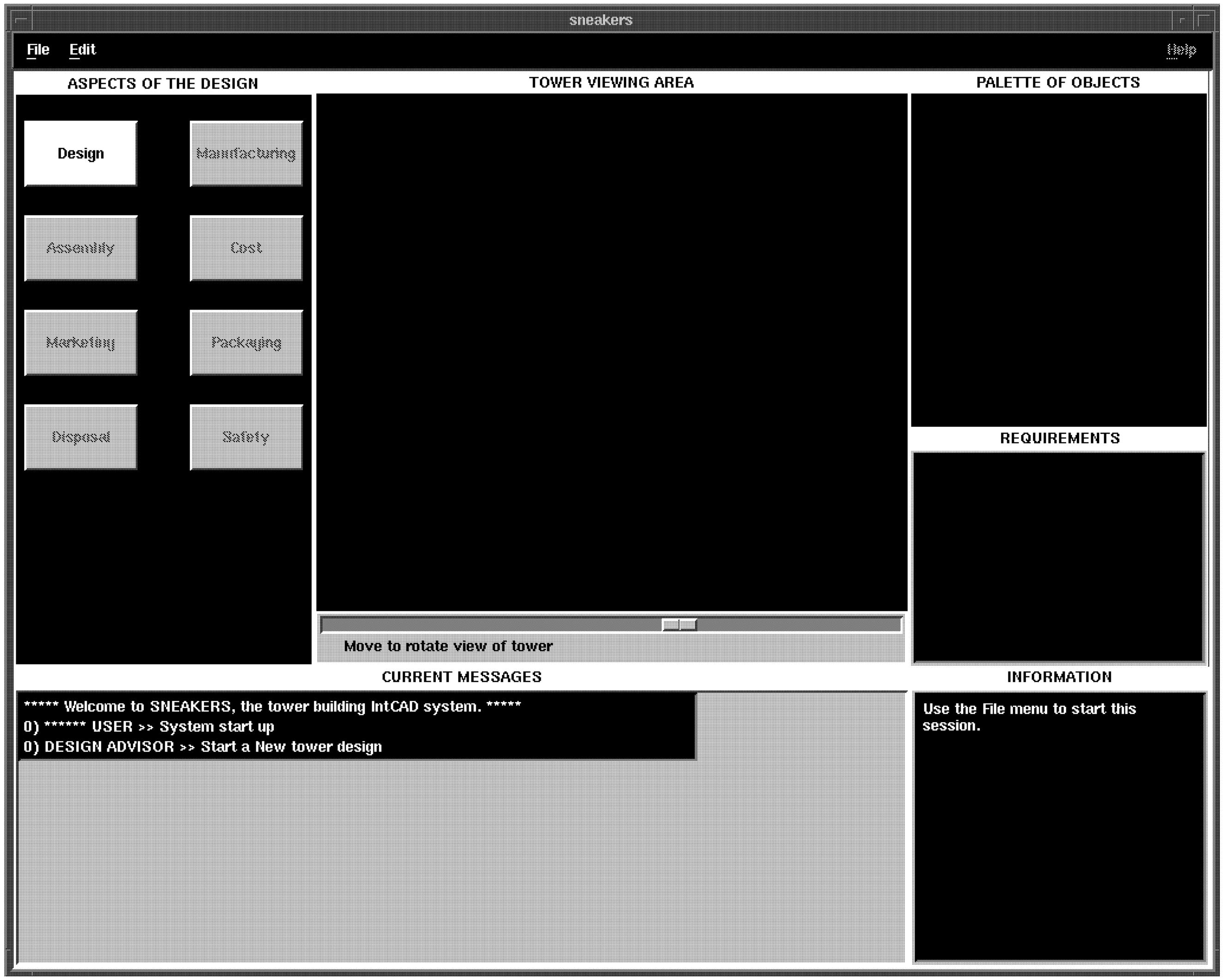
Appendix B

Sample Run

The following pages contain screen dumps of a sample run of **SNEAKERS**. These pictures show how a design session done with **SNEAKERS** looks. The figures shown are gray-scale views of the screen. The actual screen is in color, which makes it easier to see some of the features of the system and improves the quality of the interaction. **SNEAKERS** can be run on a monochrome monitor, but has the limitations shown in these screen dumps.

Also included in this appendix is the report generated for this particular design session. This report is stored in the file *sneakers.rpt* after the platform is placed to complete the design. The report includes a parts list and assembly instructions for the tower designed.

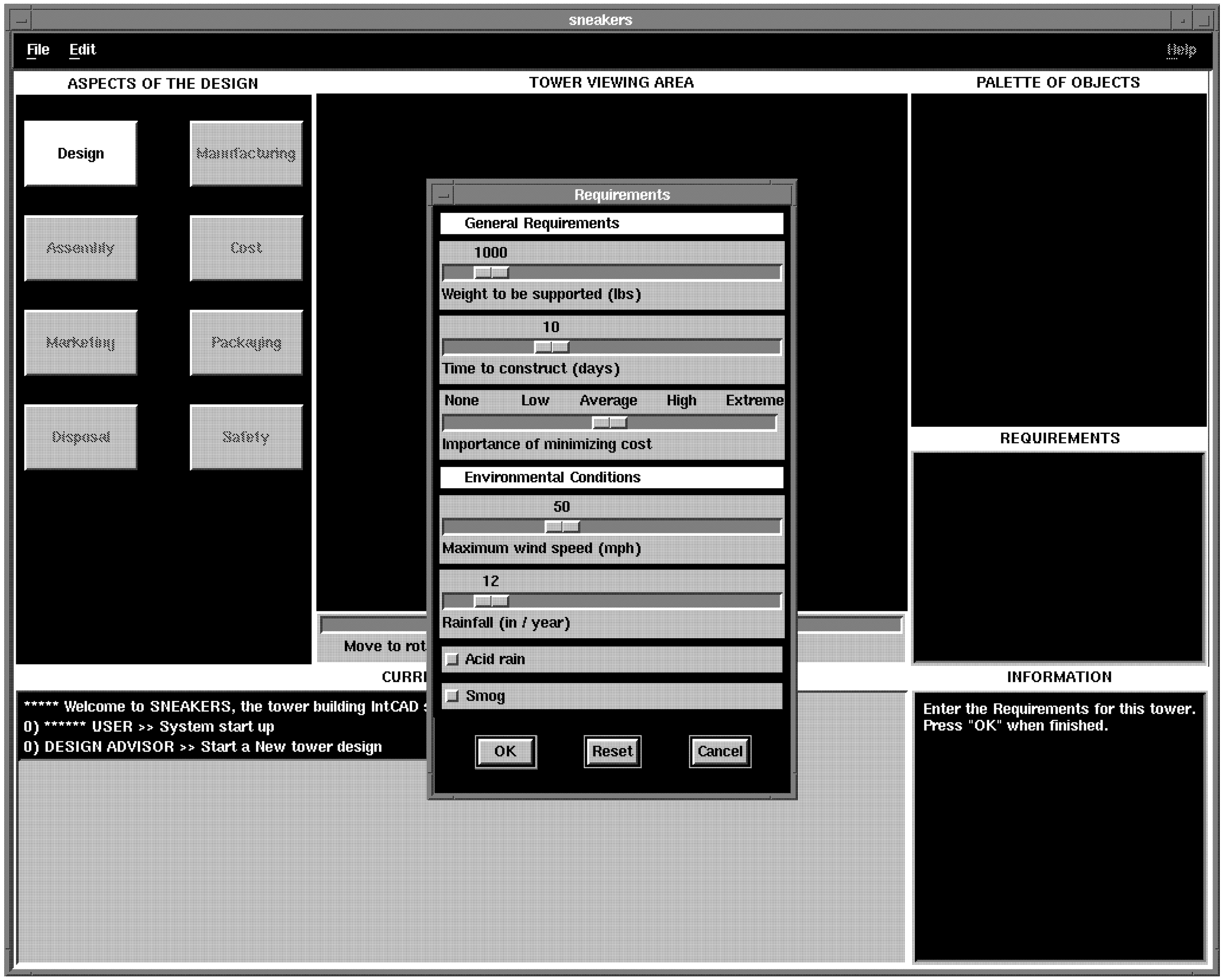
This sample run is intended to give the reader some feeling for the system and can also act as an example for learning the system. This sample and the user's guide in appendix A should be sufficient to learn to use **SNEAKERS** effectively.



```
***** Welcome to SNEAKERS, the tower building IntCAD system. *****
0) ****** USER >> System start up
0) DESIGN ADVISOR >> Start a New tower design
```

Use the File menu to start this session.

Figure B.1: System Start Up



General Requirements

1000
Weight to be supported (lbs)

10
Time to construct (days)

None Low Average High Extreme
Importance of minimizing cost

Environmental Conditions

50
Maximum wind speed (mph)

12
Rainfall (in / year)

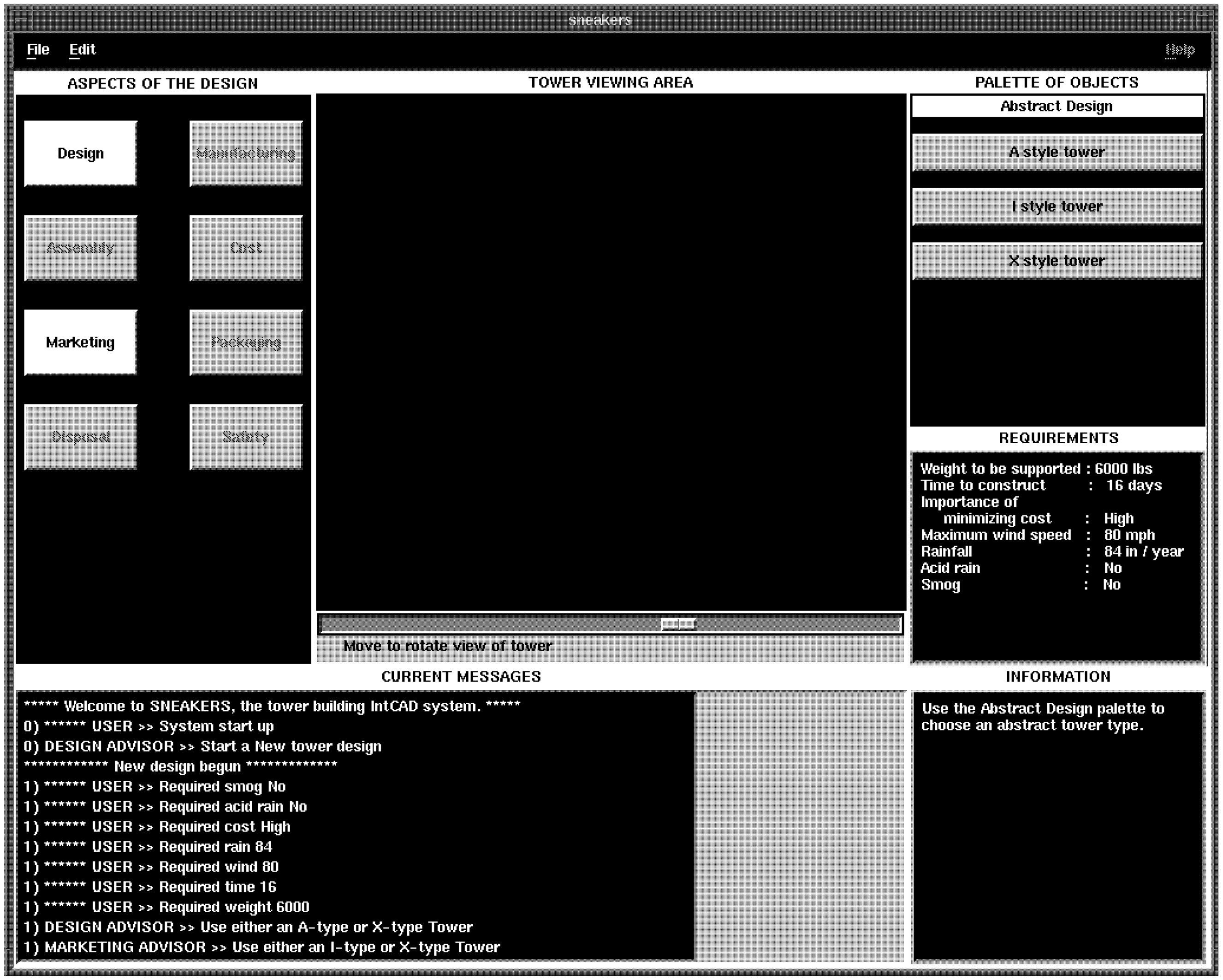
☐ Acid rain

☐ Smog

```
***** Welcome to SNEAKERS, the tower building IntCAD :
0) ****** USER >> System start up
0) DESIGN ADVISOR >> Start a New tower design
```

Enter the Requirements for this tower.
Press "OK" when finished.

Figure B.2: Enter Requirements

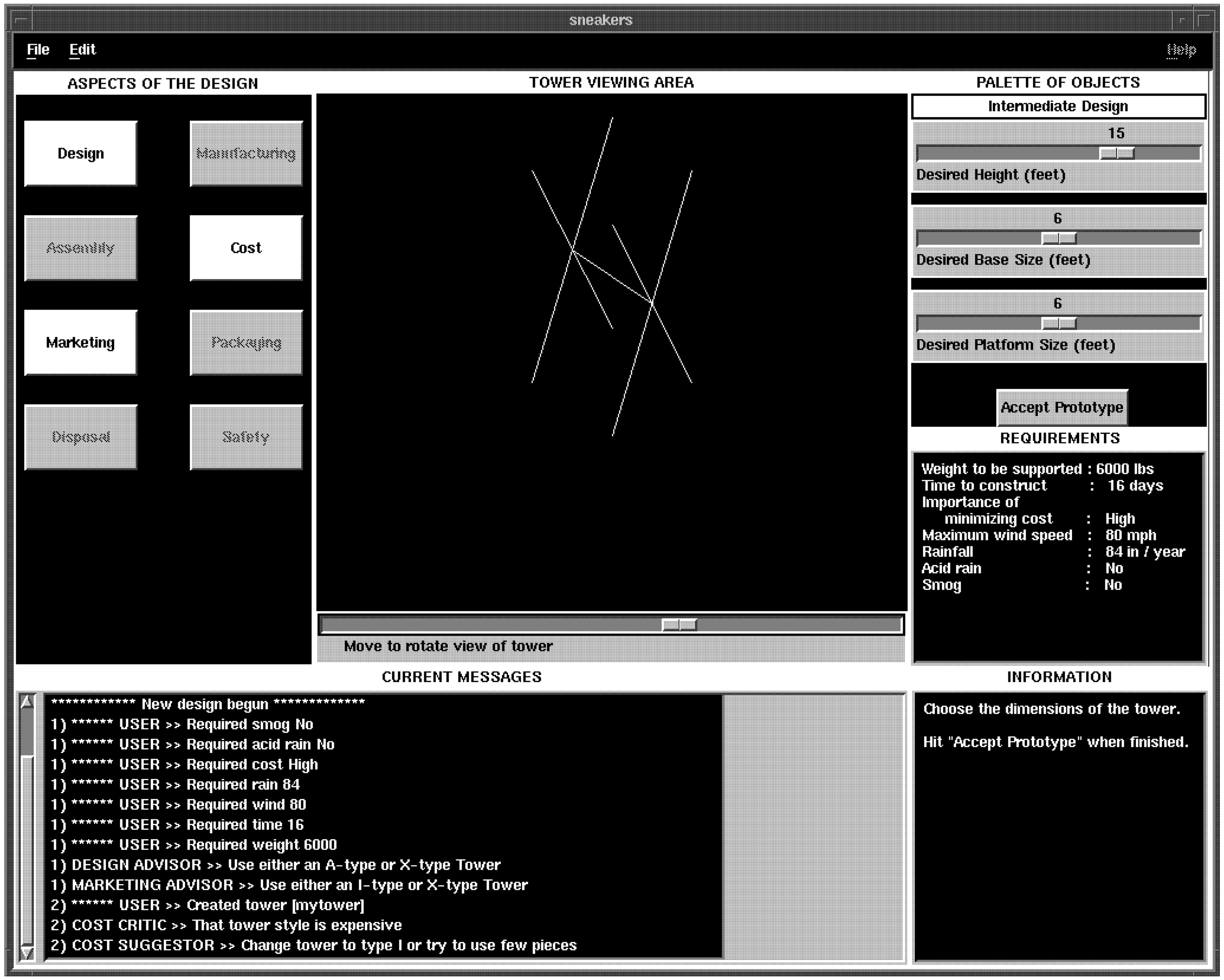


```
Weight to be supported : 6000 lbs
Time to construct      :  16 days
Importance of
    minimizing cost    :  High
Maximum wind speed     :  80 mph
Rainfall               :  84 in / year
Acid rain              :  No
Smog                   :  No
```

```
***** Welcome to SNEAKERS, the tower building IntCAD system. *****
0) ****** USER >> System start up
0) DESIGN ADVISOR >> Start a New tower design
************ New design begun *************
1) ****** USER >> Required smog No
1) ****** USER >> Required acid rain No
1) ****** USER >> Required cost High
1) ****** USER >> Required rain 84
1) ****** USER >> Required wind 80
1) ****** USER >> Required time 16
1) ****** USER >> Required weight 6000
1) DESIGN ADVISOR >> Use either an A-type or X-type Tower
1) MARKETING ADVISOR >> Use either an I-type or X-type Tower
```

Use the Abstract Design palette to choose an abstract tower type.

Figure B.3: Choose an Abstract Tower



15
Desired Height (feet)

6
Desired Base Size (feet)

6
Desired Platform Size (feet)

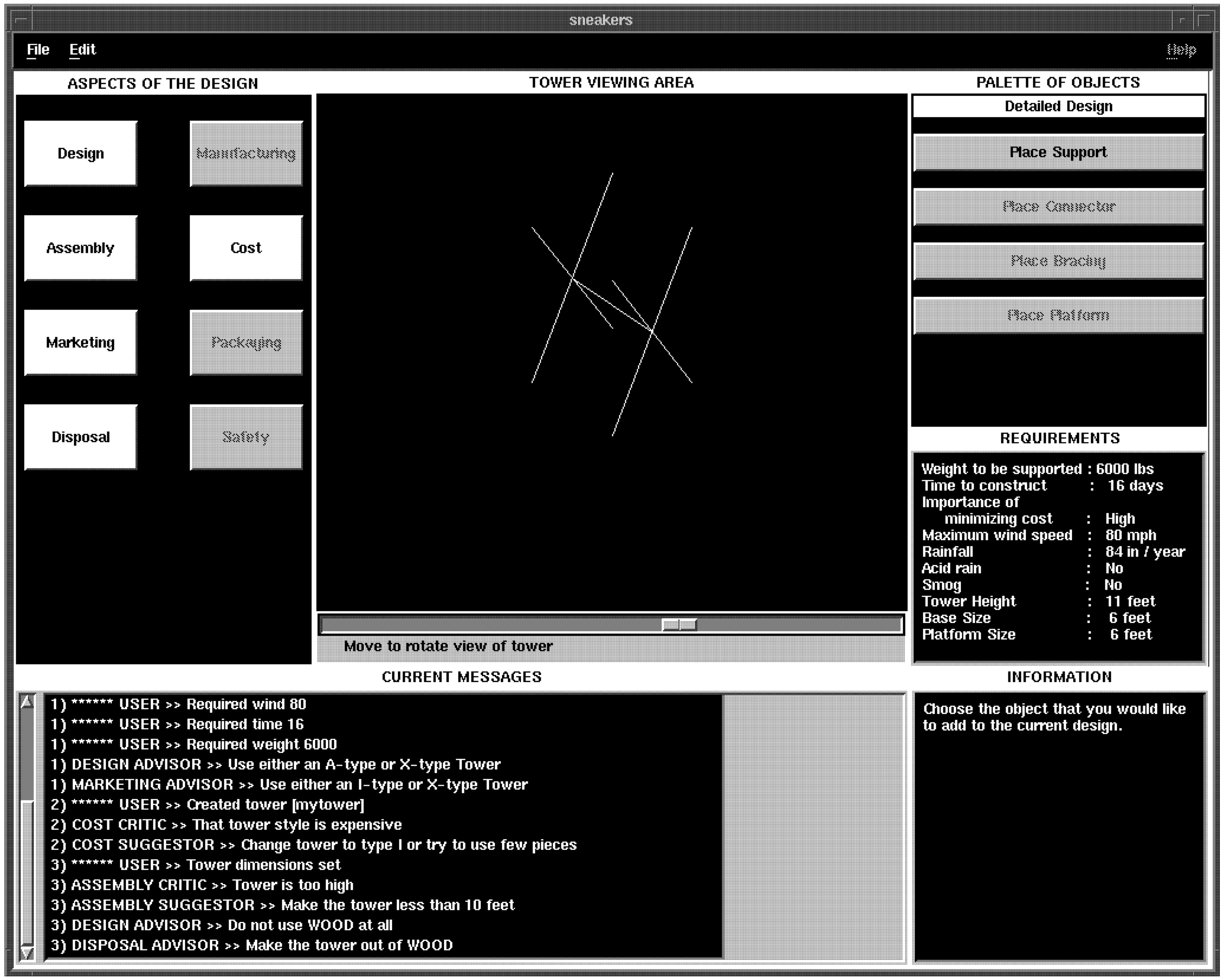
```
Weight to be supported : 6000 lbs
Time to construct      :  16 days
Importance of
    minimizing cost    :  High
Maximum wind speed     :  80 mph
Rainfall               :  84 in / year
Acid rain              :  No
Smog                   :  No
```

```
************ New design begun *************
1) ****** USER >> Required smog No
1) ****** USER >> Required acid rain No
1) ****** USER >> Required cost High
1) ****** USER >> Required rain 84
1) ****** USER >> Required wind 80
1) ****** USER >> Required time 16
1) ****** USER >> Required weight 6000
1) DESIGN ADVISOR >> Use either an A-type or X-type Tower
1) MARKETING ADVISOR >> Use either an I-type or X-type Tower
2) ****** USER >> Created tower [mytower]
2) COST CRITIC >> That tower style is expensive
2) COST SUGGESTOR >> Change tower to type I or try to use few pieces
```

Choose the dimensions of the tower.

Hit "Accept Prototype" when finished.

Figure B.4: Choose Tower Dimensions

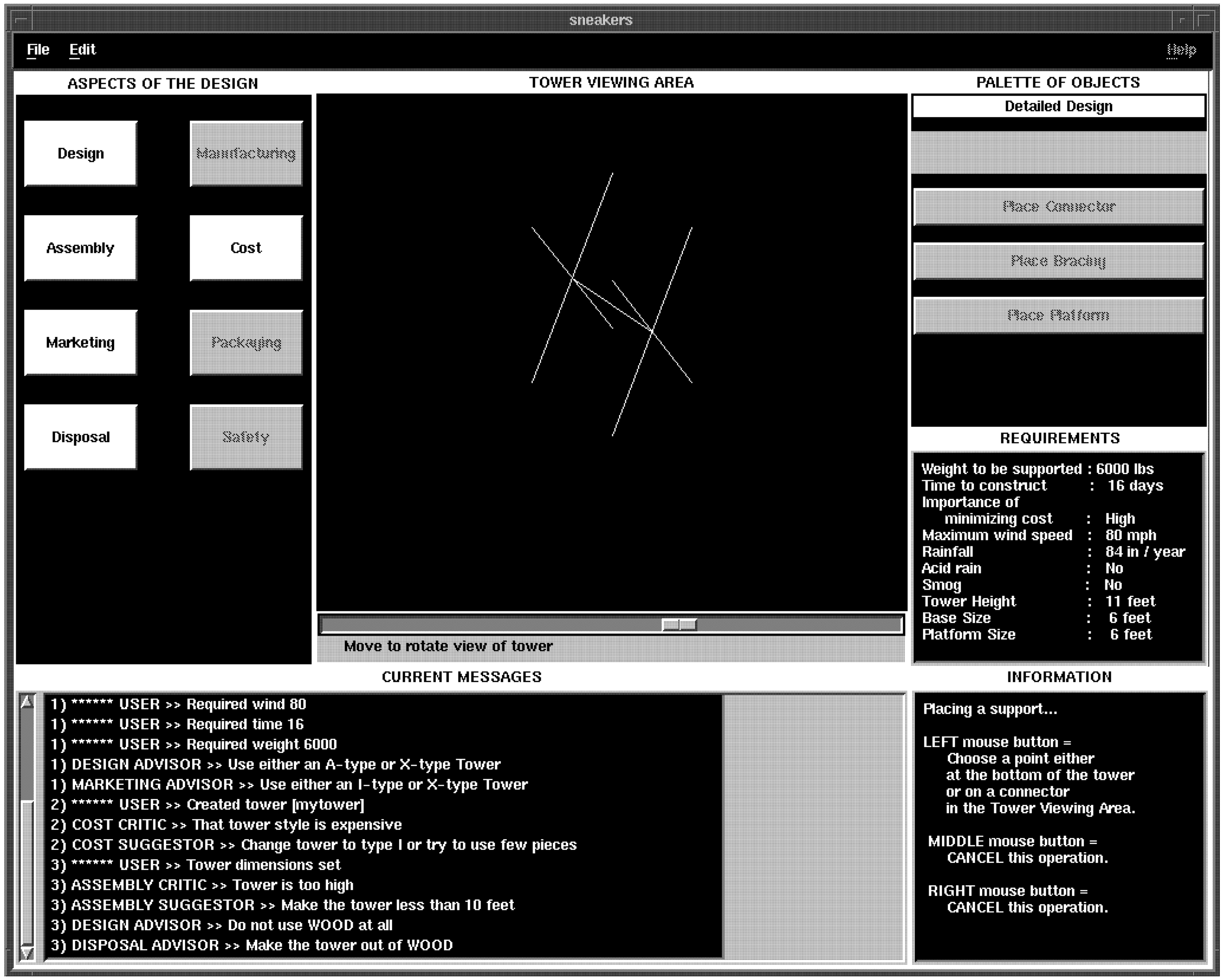


```
Weight to be supported : 6000 lbs
Time to construct      :  16 days
Importance of
    minimizing cost    :  High
Maximum wind speed     :  80 mph
Rainfall               :  84 in / year
Acid rain              :  No
Smog                   :  No
Tower Height           :  11 feet
Base Size              :   6 feet
Platform Size          :   6 feet
```

```
1) ****** USER >> Required wind 80
1) ****** USER >> Required time 16
1) ****** USER >> Required weight 6000
1) DESIGN ADVISOR >> Use either an A-type or X-type Tower
1) MARKETING ADVISOR >> Use either an I-type or X-type Tower
2) ****** USER >> Created tower [mytower]
2) COST CRITIC >> That tower style is expensive
2) COST SUGGESTOR >> Change tower to type I or try to use few pieces
3) ****** USER >> Tower dimensions set
3) ASSEMBLY CRITIC >> Tower is too high
3) ASSEMBLY SUGGESTOR >> Make the tower less than 10 feet
3) DESIGN ADVISOR >> Do not use WOOD at all
3) DISPOSAL ADVISOR >> Make the tower out of WOOD
```

Choose the object that you would like to add to the current design.

Figure B.5: Ready to Place the First Piece



Weight to be supported	:	6000 lbs
Time to construct	:	16 days
Importance of minimizing cost	:	High
Maximum wind speed	:	80 mph
Rainfall	:	84 in / year
Acid rain	:	No
Smog	:	No
Tower Height	:	11 feet
Base Size	:	6 feet
Platform Size	:	6 feet

```
1) ****** USER >> Required wind 80
1) ****** USER >> Required time 16
1) ****** USER >> Required weight 6000
1) DESIGN ADVISOR >> Use either an A-type or X-type Tower
1) MARKETING ADVISOR >> Use either an I-type or X-type Tower
2) ****** USER >> Created tower [mytower]
2) COST CRITIC >> That tower style is expensive
2) COST SUGGESTOR >> Change tower to type I or try to use few pieces
3) ****** USER >> Tower dimensions set
3) ASSEMBLY CRITIC >> Tower is too high
3) ASSEMBLY SUGGESTOR >> Make the tower less than 10 feet
3) DESIGN ADVISOR >> Do not use WOOD at all
3) DISPOSAL ADVISOR >> Make the tower out of WOOD
```

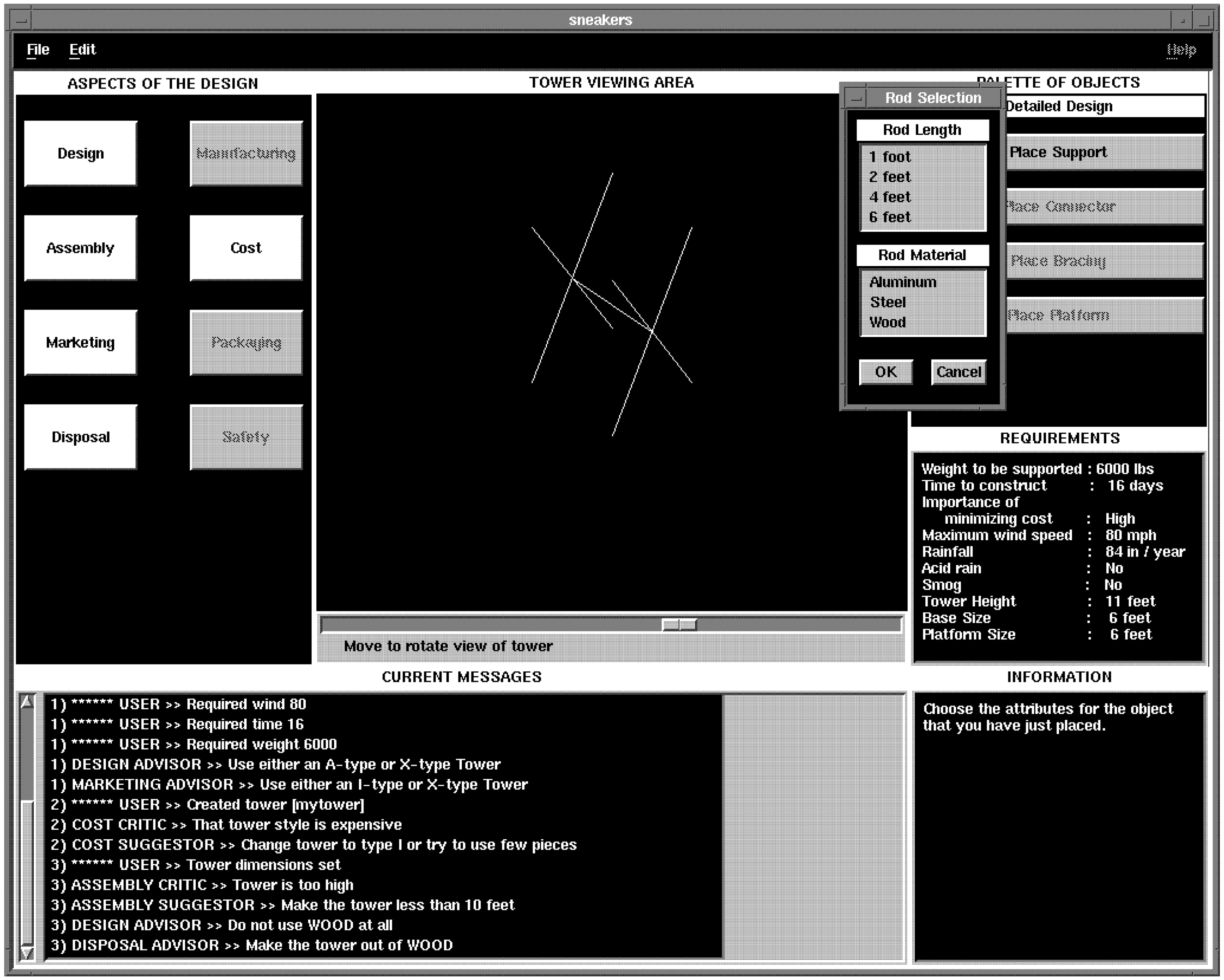
```
Placing a support...

LEFT mouse button =
    Choose a point either
    at the bottom of the tower
    or on a connector
    in the Tower Viewing Area.

MIDDLE mouse button =
    CANCEL this operation.

RIGHT mouse button =
    CANCEL this operation.
```

Figure B.6: After Choosing "Place Support" Button

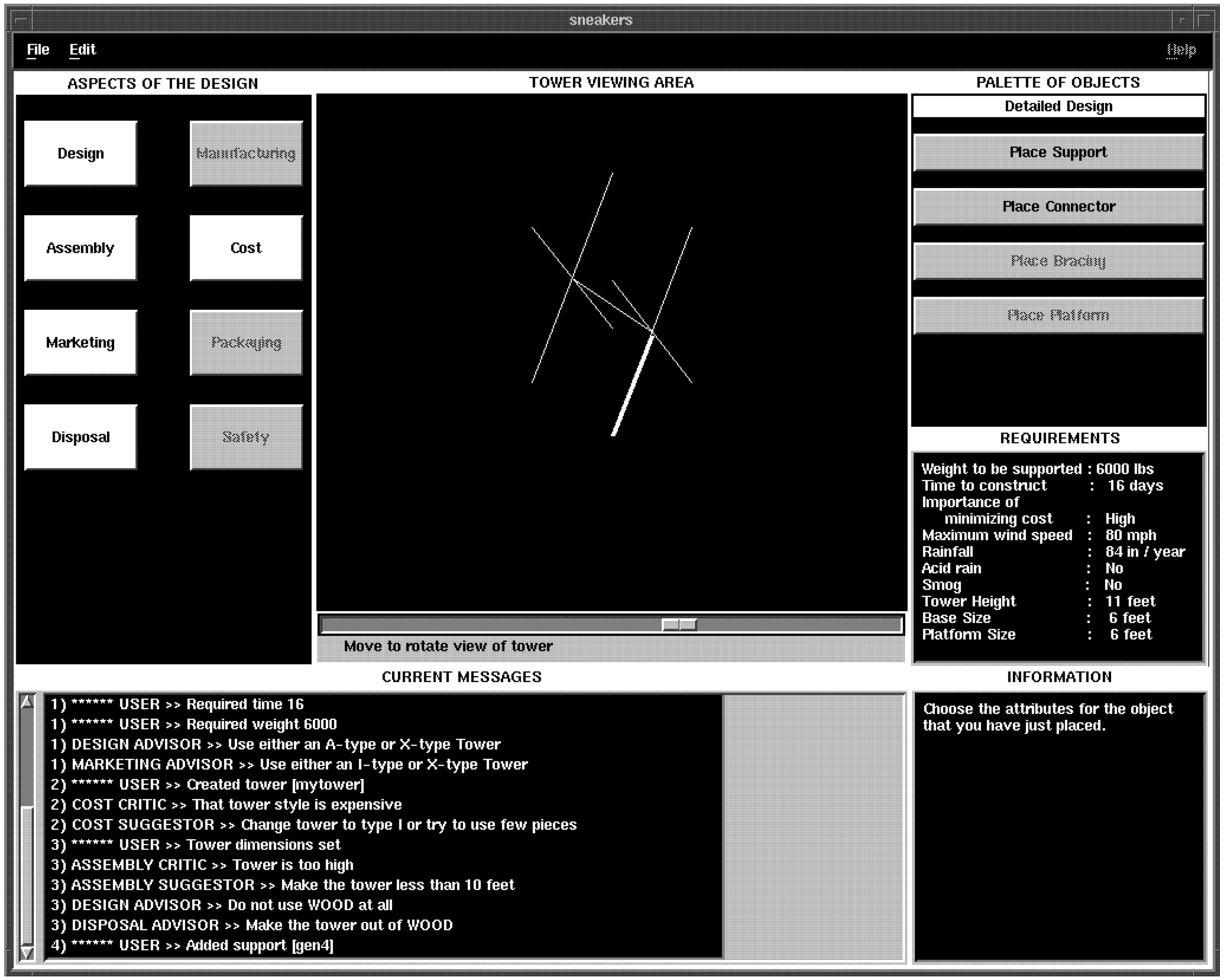


```
Weight to be supported : 6000 lbs
Time to construct      :  16 days
Importance of
    minimizing cost    :  High
Maximum wind speed     :  80 mph
Rainfall               :  84 in / year
Acid rain              :  No
Smog                   :  No
Tower Height           :  11 feet
Base Size              :   6 feet
Platform Size          :   6 feet
```

```
1) ****** USER >> Required wind 80
1) ****** USER >> Required time 16
1) ****** USER >> Required weight 6000
1) DESIGN ADVISOR >> Use either an A-type or X-type Tower
1) MARKETING ADVISOR >> Use either an I-type or X-type Tower
2) ****** USER >> Created tower [mytower]
2) COST CRITIC >> That tower style is expensive
2) COST SUGGESTOR >> Change tower to type I or try to use few pieces
3) ****** USER >> Tower dimensions set
3) ASSEMBLY CRITIC >> Tower is too high
3) ASSEMBLY SUGGESTOR >> Make the tower less than 10 feet
3) DESIGN ADVISOR >> Do not use WOOD at all
3) DISPOSAL ADVISOR >> Make the tower out of WOOD
```

Choose the attributes for the object that you have just placed.

Figure B.7: Selecting the Support's Attributes

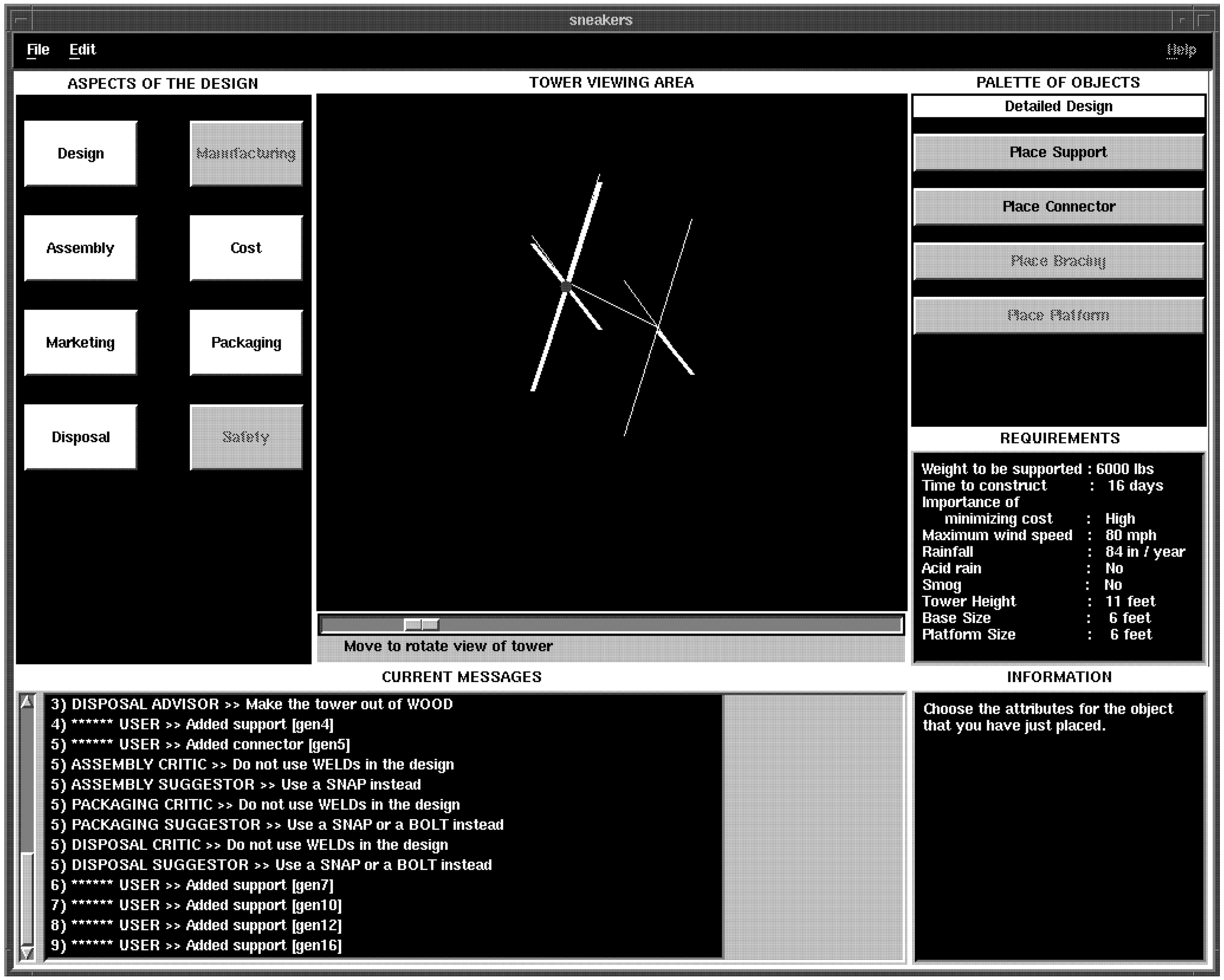


Weight to be supported	:	6000 lbs
Time to construct	:	16 days
Importance of minimizing cost	:	High
Maximum wind speed	:	80 mph
Rainfall	:	84 in / year
Acid rain	:	No
Smog	:	No
Tower Height	:	11 feet
Base Size	:	6 feet
Platform Size	:	6 feet

```
1) ****** USER >> Required time 16
1) ****** USER >> Required weight 6000
1) DESIGN ADVISOR >> Use either an A-type or X-type Tower
1) MARKETING ADVISOR >> Use either an I-type or X-type Tower
2) ****** USER >> Created tower [mytower]
2) COST CRITIC >> That tower style is expensive
2) COST SUGGESTOR >> Change tower to type I or try to use few pieces
3) ****** USER >> Tower dimensions set
3) ASSEMBLY CRITIC >> Tower is too high
3) ASSEMBLY SUGGESTOR >> Make the tower less than 10 feet
3) DESIGN ADVISOR >> Do not use WOOD at all
3) DISPOSAL ADVISOR >> Make the tower out of WOOD
4) ****** USER >> Added support [gen4]
```

Choose the attributes for the object that you have just placed.

Figure B.8: Support Placed

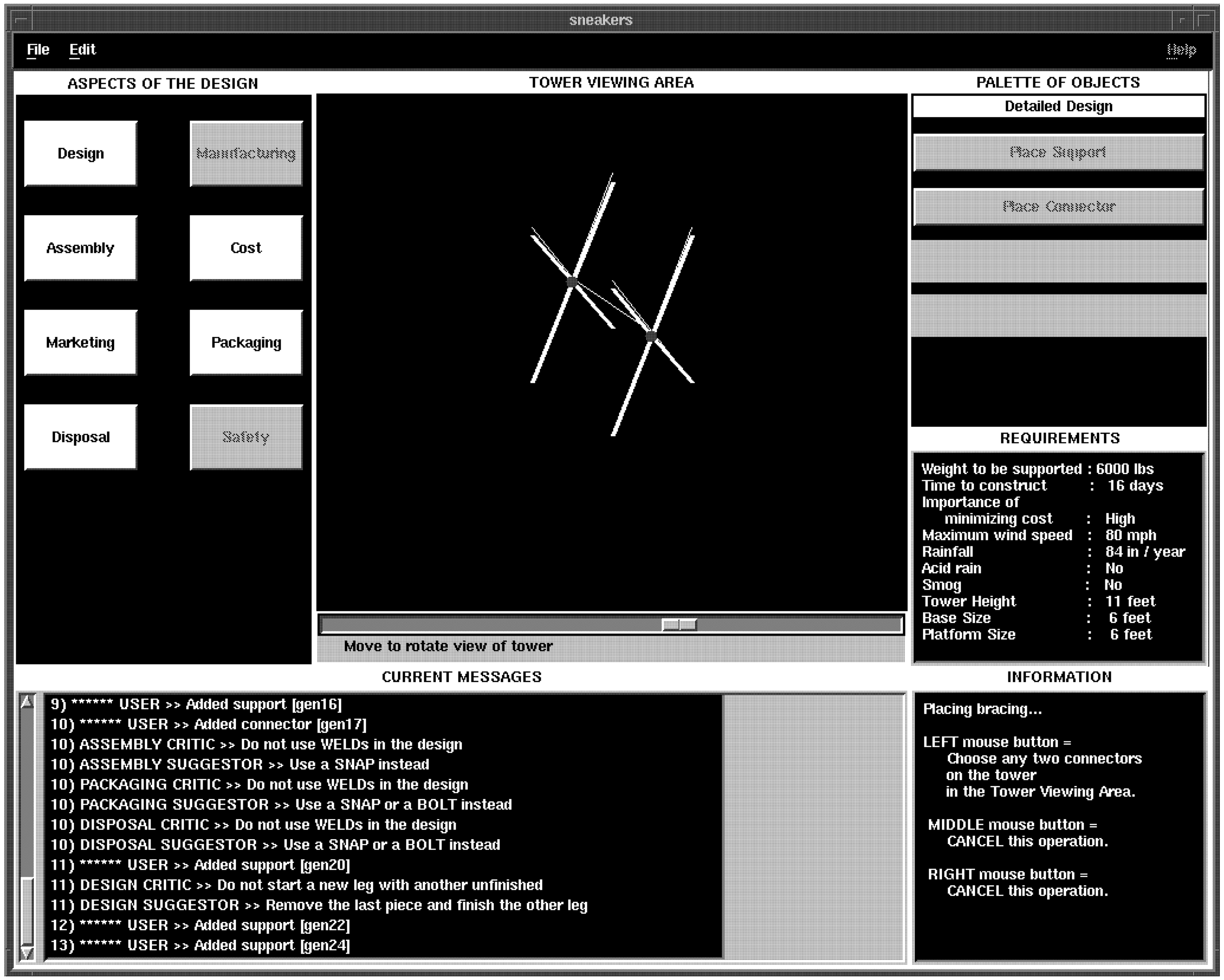


Weight to be supported	:	6000 lbs
Time to construct	:	16 days
Importance of minimizing cost	:	High
Maximum wind speed	:	80 mph
Rainfall	:	84 in / year
Acid rain	:	No
Smog	:	No
Tower Height	:	11 feet
Base Size	:	6 feet
Platform Size	:	6 feet

```
3) DISPOSAL ADVISOR >> Make the tower out of WOOD
4) ****** USER >> Added support [gen4]
5) ****** USER >> Added connector [gen5]
5) ASSEMBLY CRITIC >> Do not use WELDs in the design
5) ASSEMBLY SUGGESTOR >> Use a SNAP instead
5) PACKAGING CRITIC >> Do not use WELDs in the design
5) PACKAGING SUGGESTOR >> Use a SNAP or a BOLT instead
5) DISPOSAL CRITIC >> Do not use WELDs in the design
5) DISPOSAL SUGGESTOR >> Use a SNAP or a BOLT instead
6) ****** USER >> Added support [gen7]
7) ****** USER >> Added support [gen10]
8) ****** USER >> Added support [gen12]
9) ****** USER >> Added support [gen16]
```

Choose the attributes for the object that you have just placed.

Figure B.9: Middle of the Design



```
Weight to be supported : 6000 lbs
Time to construct      :  16 days
Importance of
    minimizing cost    :  High
Maximum wind speed     :  80 mph
Rainfall               :  84 in / year
Acid rain              :  No
Smog                   :  No
Tower Height           :  11 feet
Base Size              :   6 feet
Platform Size          :   6 feet
```

```
9) ****** USER >> Added support [gen16]
10) ****** USER >> Added connector [gen17]
10) ASSEMBLY CRITIC >> Do not use WELDs in the design
10) ASSEMBLY SUGGESTOR >> Use a SNAP instead
10) PACKAGING CRITIC >> Do not use WELDs in the design
10) PACKAGING SUGGESTOR >> Use a SNAP or a BOLT instead
10) DISPOSAL CRITIC >> Do not use WELDs in the design
10) DISPOSAL SUGGESTOR >> Use a SNAP or a BOLT instead
11) ****** USER >> Added support [gen20]
11) DESIGN CRITIC >> Do not start a new leg with another unfinished
11) DESIGN SUGGESTOR >> Remove the last piece and finish the other leg
12) ****** USER >> Added support [gen22]
13) ****** USER >> Added support [gen24]
```

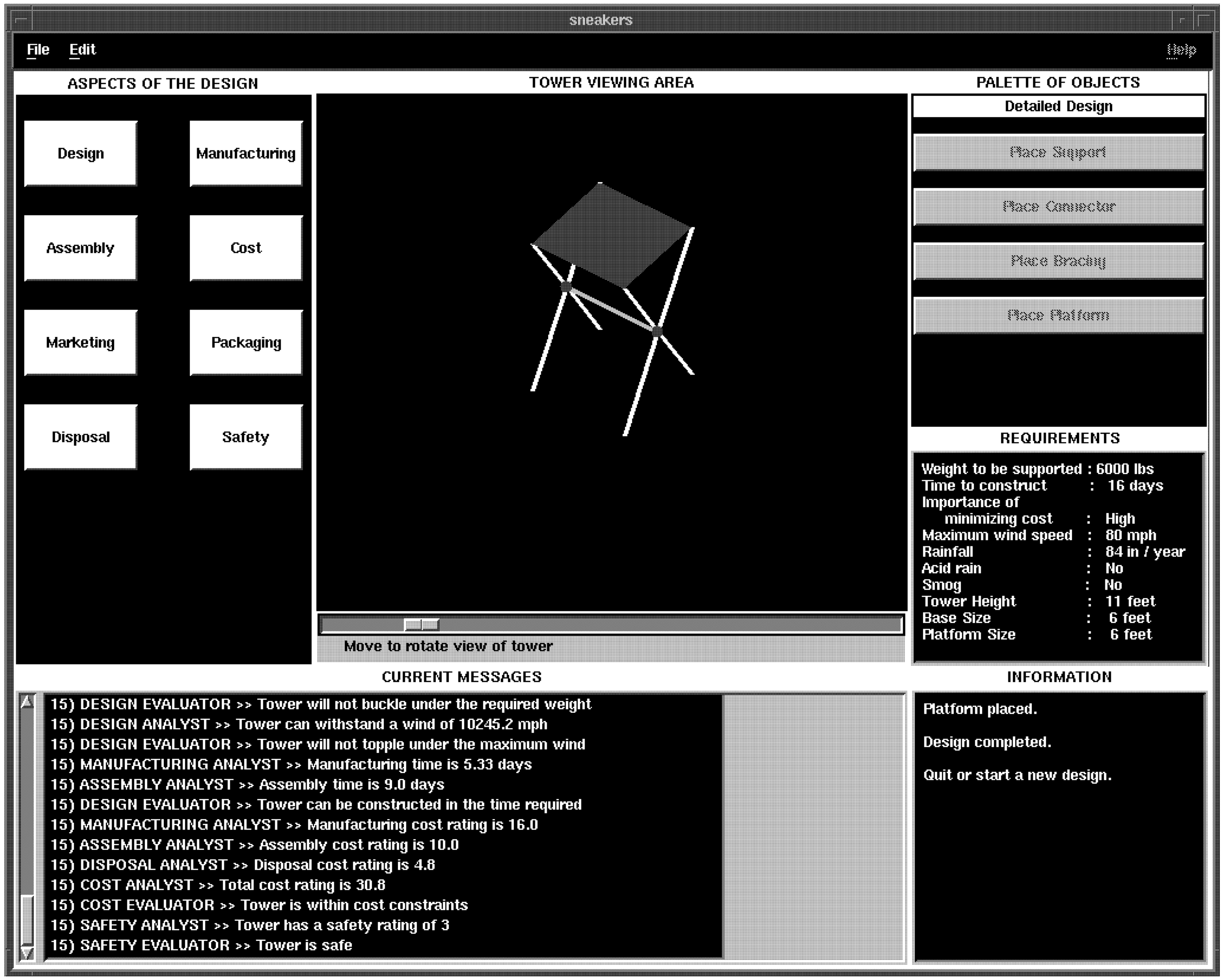
```
Placing bracing...

LEFT mouse button =
    Choose any two connectors
    on the tower
    in the Tower Viewing Area.

 MIDDLE mouse button =
    CANCEL this operation.

 RIGHT mouse button =
    CANCEL this operation.
```

Figure B.10: After Choosing "Place Bracing" Button



Weight to be supported	:	6000 lbs
Time to construct	:	16 days
Importance of minimizing cost	:	High
Maximum wind speed	:	80 mph
Rainfall	:	84 in / year
Acid rain	:	No
Smog	:	No
Tower Height	:	11 feet
Base Size	:	6 feet
Platform Size	:	6 feet

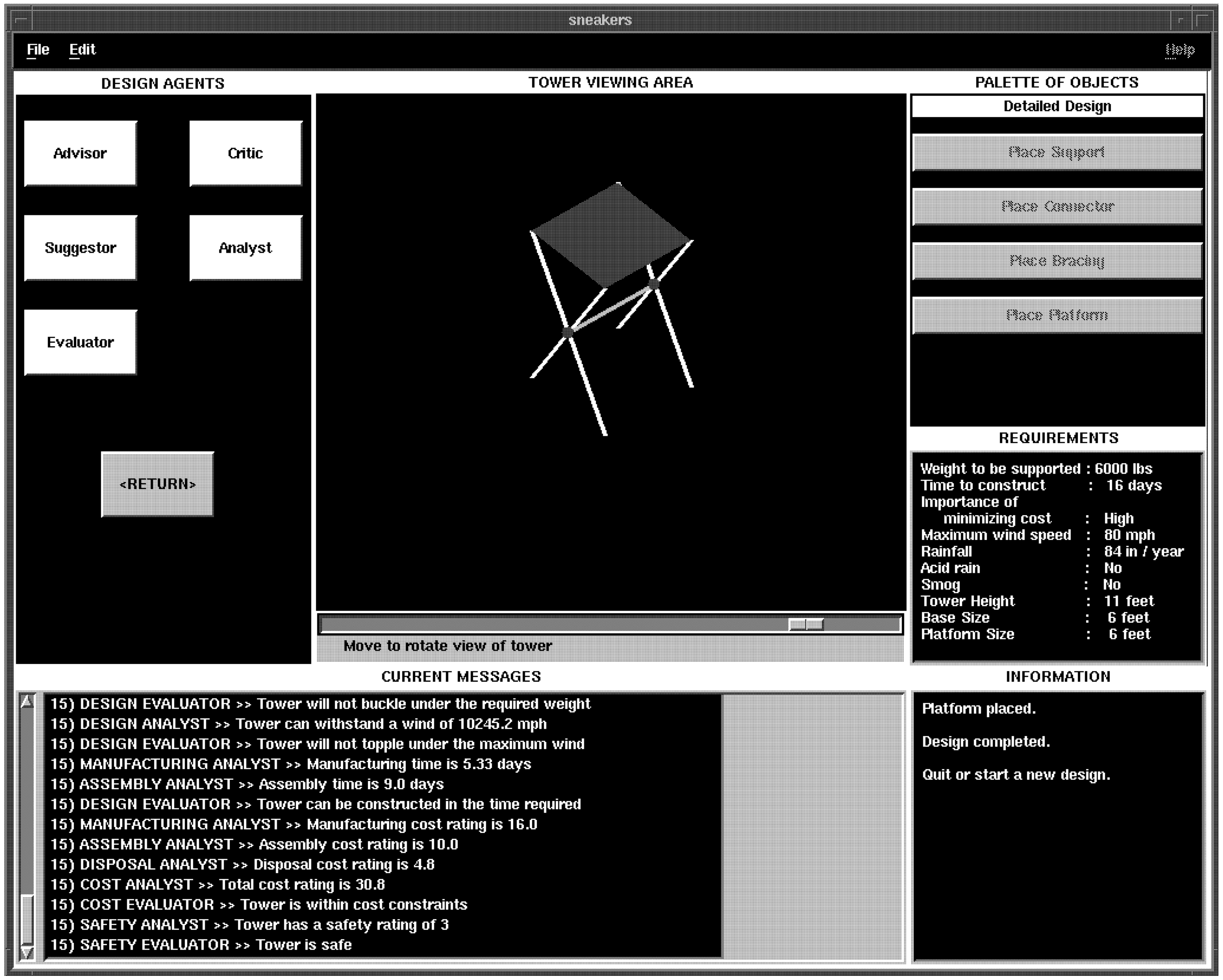
```
15) DESIGN EVALUATOR >> Tower will not buckle under the required weight
15) DESIGN ANALYST >> Tower can withstand a wind of 10245.2 mph
15) DESIGN EVALUATOR >> Tower will not topple under the maximum wind
15) MANUFACTURING ANALYST >> Manufacturing time is 5.33 days
15) ASSEMBLY ANALYST >> Assembly time is 9.0 days
15) DESIGN EVALUATOR >> Tower can be constructed in the time required
15) MANUFACTURING ANALYST >> Manufacturing cost rating is 16.0
15) ASSEMBLY ANALYST >> Assembly cost rating is 10.0
15) DISPOSAL ANALYST >> Disposal cost rating is 4.8
15) COST ANALYST >> Total cost rating is 30.8
15) COST EVALUATOR >> Tower is within cost constraints
15) SAFETY ANALYST >> Tower has a safety rating of 3
15) SAFETY EVALUATOR >> Tower is safe
```

Platform placed.

Design completed.

Quit or start a new design.

Figure B.11: Platform & Bracing Placed and Design Complete



Weight to be supported	6000 lbs
Time to construct	16 days
Importance of minimizing cost	High
Maximum wind speed	80 mph
Rainfall	84 in / year
Acid rain	No
Smog	No
Tower Height	11 feet
Base Size	6 feet
Platform Size	6 feet

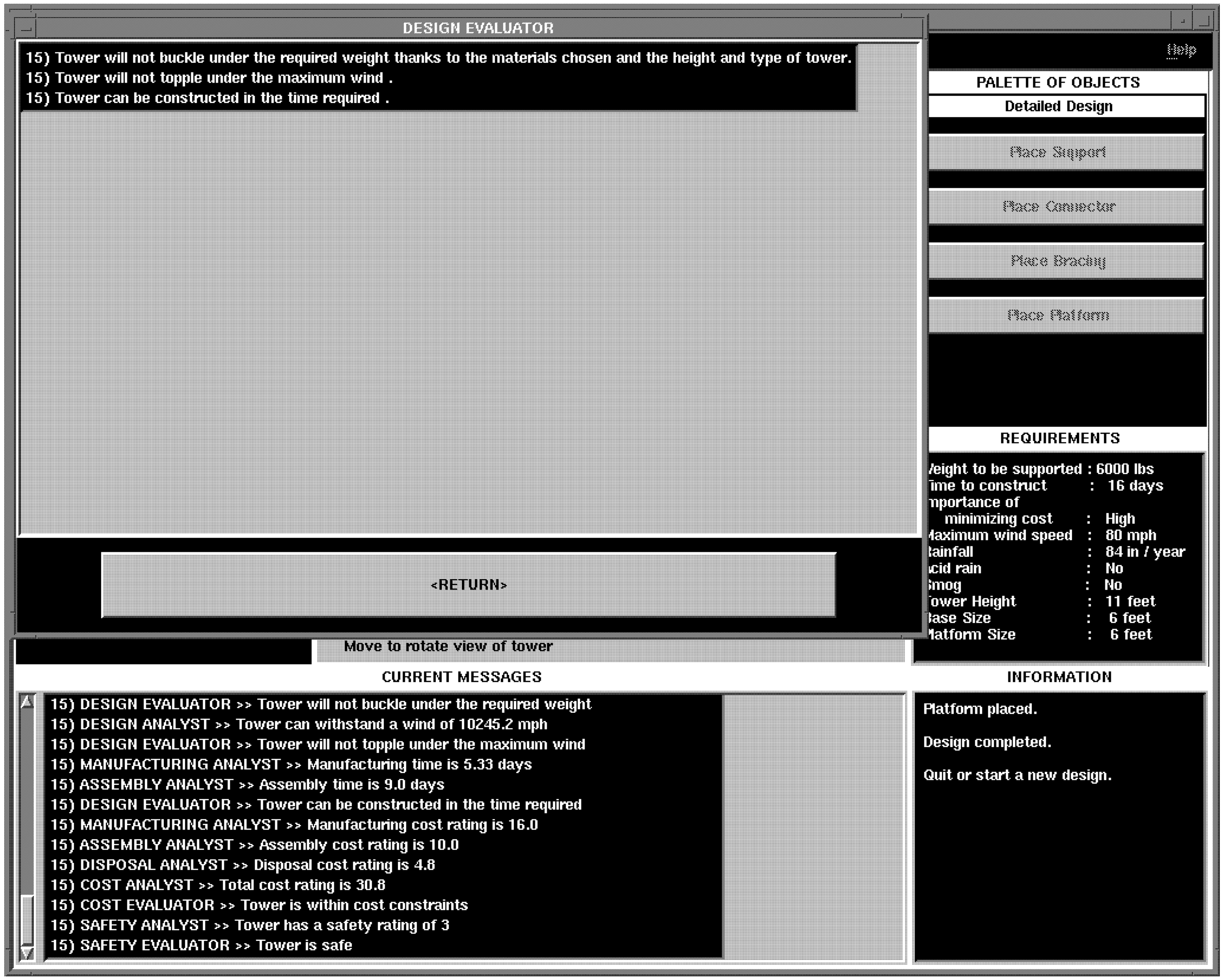
```
15) DESIGN EVALUATOR >> Tower will not buckle under the required weight
15) DESIGN ANALYST >> Tower can withstand a wind of 10245.2 mph
15) DESIGN EVALUATOR >> Tower will not topple under the maximum wind
15) MANUFACTURING ANALYST >> Manufacturing time is 5.33 days
15) ASSEMBLY ANALYST >> Assembly time is 9.0 days
15) DESIGN EVALUATOR >> Tower can be constructed in the time required
15) MANUFACTURING ANALYST >> Manufacturing cost rating is 16.0
15) ASSEMBLY ANALYST >> Assembly cost rating is 10.0
15) DISPOSAL ANALYST >> Disposal cost rating is 4.8
15) COST ANALYST >> Total cost rating is 30.8
15) COST EVALUATOR >> Tower is within cost constraints
15) SAFETY ANALYST >> Tower has a safety rating of 3
15) SAFETY EVALUATOR >> Tower is safe
```

Platform placed.

Design completed.

Quit or start a new design.

Figure B.12: Agent Buttons within the Design Aspect



```
15) Tower will not buckle under the required weight thanks to the materials chosen and the height and type of tower.
15) Tower will not topple under the maximum wind .
15) Tower can be constructed in the time required .
```

```
/eight to be supported : 6000 lbs
ime to construct       :  16 days
nportance of
  minimizing cost      :  High
laximum wind speed     :  80 mph
ainfall                :  84 in / year
cid rain               :  No
mog                    :  No
ower Height            :  11 feet
ase Size               :   6 feet
latform Size           :   6 feet
```

```
15) DESIGN EVALUATOR >> Tower will not buckle under the required weight
15) DESIGN ANALYST >> Tower can withstand a wind of 10245.2 mph
15) DESIGN EVALUATOR >> Tower will not topple under the maximum wind
15) MANUFACTURING ANALYST >> Manufacturing time is 5.33 days
15) ASSEMBLY ANALYST >> Assembly time is 9.0 days
15) DESIGN EVALUATOR >> Tower can be constructed in the time required
15) MANUFACTURING ANALYST >> Manufacturing cost rating is 16.0
15) ASSEMBLY ANALYST >> Assembly cost rating is 10.0
15) DISPOSAL ANALYST >> Disposal cost rating is 4.8
15) COST ANALYST >> Total cost rating is 30.8
15) COST EVALUATOR >> Tower is within cost constraints
15) SAFETY ANALYST >> Tower has a safety rating of 3
15) SAFETY EVALUATOR >> Tower is safe
```

```
Platform placed.

Design completed.

Quit or start a new design.
```

Figure B.13: The Design Evaluator's Messages

```
                   TOWER DESIGN REPORT

PARTS LIST
----------
Steel Supports - 6 foot = 8

Welds = 2

ASSEMBLY SEQUENCE
-----------------
Place a 6 foot STEEL support from (3.0, 3.0, 0.0) to (3.0, 0.0, 5.2)
Place a WELD at (3.0, 0.0, 5.2)
Place a 6 foot STEEL support from (3.0, 0.0, 5.2) to (3.0, -3.0, 10.39)
Place a 6 foot STEEL support from (3.0, -3.0, 0.0) to (3.0, 0.0, 5.2)
Place a 6 foot STEEL support from (3.0, 0.0, 5.2) to (3.0, 3.0, 10.39)
Place a 6 foot STEEL support from (-3.0, 3.0, 0.0) to (-3.0, 0.0, 5.2)
Place a WELD at (-3.0, 0.0, 5.2)
Place a 6 foot STEEL support from (-3.0, -3.0, 0.0) to (-3.0, 0.0, 5.2)
Place a 6 foot STEEL support from (-3.0, 0.0, 5.2) to (-3.0, -3.0, 10.39)
Place a 6 foot STEEL support from (-3.0, 0.0, 5.2) to (-3.0, 3.0, 10.39)
Place a brace from (-3.0, 0.0, 5.2) to (3.0, 0.0, 5.2)
```

Appendix C

Tower Building Objects

This appendix shows the objects which are used in COOL – CLIPS Object Oriented Language – and their relationships to each other. Some of the objects are related by an **IS-A** relationship. This means that one of the objects **is a** more specific version of the other. Objects are also related by a **COMPOSED-OF** relationship, in which an object is **composed of** instances of other objects, eg. a tower is composed of supports. Figure C.1 shows these relationships. Also included in this appendix is a listing of the COOL code which implements these objects and relationships.

```
;*******************************************************************
; TOYS.CLP - the objects used in SNEAKERS
;*******************************************************************

;***** TOYS in SNEAKERS ****

(defclass TOY (is-a USER)
        (slot time))

;********** POINTS *********

(defclass POINT (is-a TOY)
        (slot x)
        (slot y)
        (slot z))

;*********** PLATFORMS ************
```

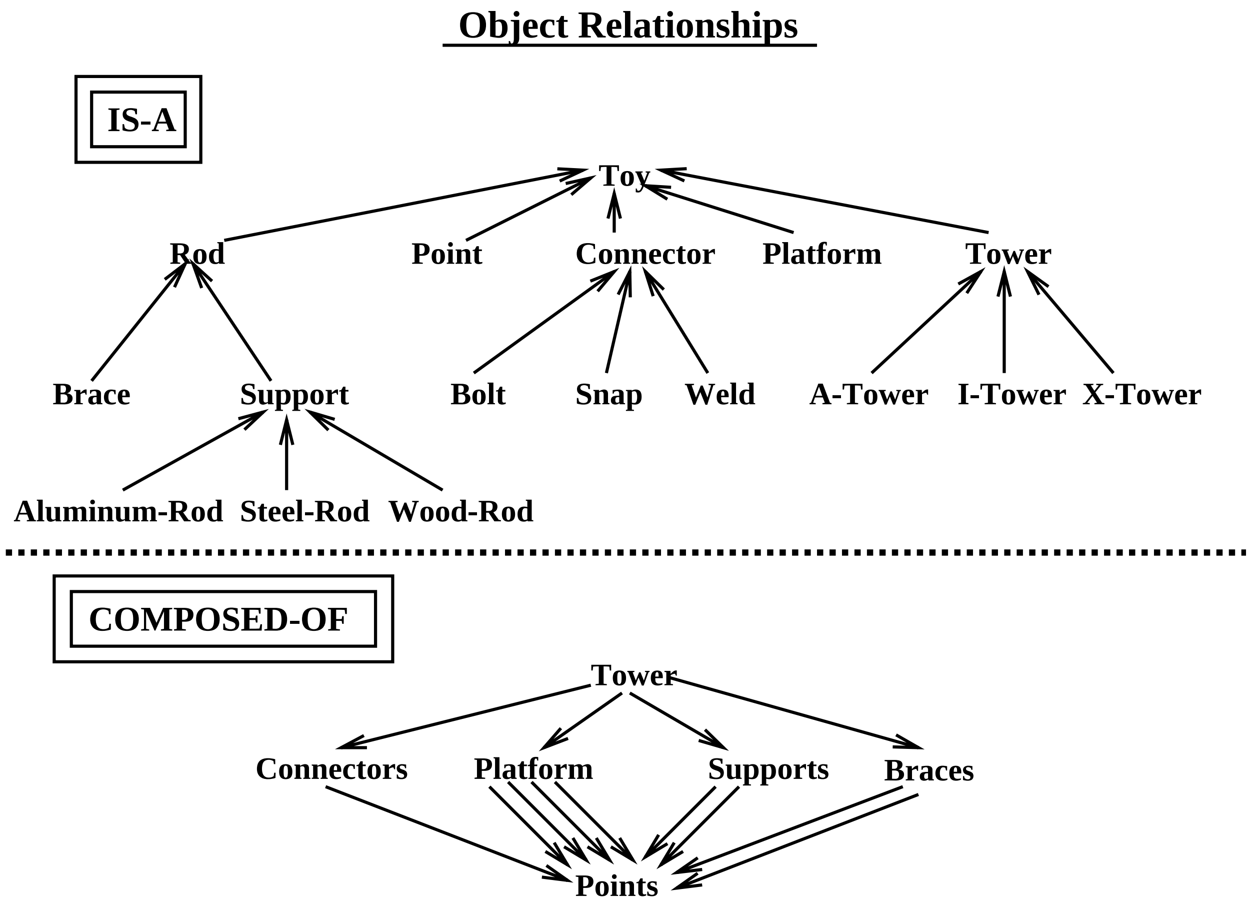


Figure C.1: Relationships of Objects in COOL

```
(defclass PLATFORM (is-a TOY)
        (slot shape (allowed-symbols SQUARE TRIANGLE))
        (slot corner1)
        (slot corner2)
        (slot corner3)
        (slot corner4))

;********* SUPPORTS **********

(defclass ROD (is-a TOY)
        (slot start-point)
        (slot end-point))

(defclass BRACE (is-a ROD))

(defclass SUPPORT (is-a ROD)
        (slot length (allowed-integers 1 2 4 6))
        (slot destination)
        (slot material (read-only)(allowed-symbols ALUMINUM STEEL WOOD))
        (slot cost)
        (slot weight)
        (slot durability (read-only))
        (slot strength (read-only))
        (slot manufact-time (read-only))
        (slot disposal-cost (read-only))
        (slot illegal-connectors (multiple)(read-only)
                (allowed-symbols WELD BOLT SNAP)))

(defclass ALUMINUM-ROD (is-a SUPPORT)
        (slot material (composite)(default ALUMINUM))
        (slot cost (default 8))
        (slot weight (default 3))
        (slot durability (composite) (default 6))
        (slot strength (composite) (default 4))
        (slot manufact-time (composite) (default 2))
        (slot disposal-cost (composite) (default 1))
        (slot illegal-connectors (composite)))

(defclass STEEL-ROD (is-a SUPPORT)
        (slot material (composite)(default STEEL))
        (slot cost (default 2))
        (slot weight (default 10))
        (slot durability (composite) (default 3))
        (slot strength (composite)(default 10))
        (slot manufact-time (composite) (default 4))
        (slot disposal-cost (composite) (default 2))
        (slot illegal-connectors (composite)))

(defclass WOOD-ROD (is-a SUPPORT)
        (slot material (composite)(default WOOD))
        (slot cost (default 1))
        (slot weight (default 2))
        (slot durability (composite) (default 1))
        (slot strength (composite) (default 1))
        (slot manufact-time (composite) (default 1))
        (slot disposal-cost (composite) (default 3))
        (slot illegal-connectors (composite) (default WELD SNAP)))

;*********** CONNECTORS *************

(defclass CONNECTOR (is-a TOY)
        (slot position)
        (slot type (read-only)(allowed-symbols WELD BOLT SNAP))
        (slot cost (read-only))
        (slot assembly-time (read-only))
        (slot disposal-cost (read-only))
        (slot illegal-materials        (multiple)(read-only)
                (allowed-symbols ALUMINUM STEEL WOOD))
        (slot angles-allowed (multiple)(read-only)))

(defclass BOLT (is-a CONNECTOR)
        (slot type (composite)(default BOLT))
        (slot cost (composite) (default 1))
        (slot assembly-time (composite) (default 3))
        (slot disposal-cost (composite) (default 1))
        (slot illegal-materials (composite))
        (slot angles-allowed (composite)(default 0 45 90)))

(defclass SNAP (is-a CONNECTOR)
        (slot type (shared)(read-only)(default SNAP))
        (slot cost (composite) (default 2))
        (slot assembly-time (composite) (default 1))
        (slot disposal-cost (composite) (default 1))
        (slot illegal-materials (composite)(default WOOD))
        (slot angles-allowed (composite)(default 0 90)))

(defclass WELD (is-a CONNECTOR)
        (slot type (composite)(default WELD))
        (slot cost (composite) (default 5))
        (slot assembly-time (composite) (default 8))
        (slot disposal-cost (composite) (default 4))
        (slot illegal-materials (composite)(default WOOD))
        (slot angles-allowed (composite)(default 0 30 45 60 90)))
```

```
;*********** TOWERS ***********

(defclass TOWER (is-a TOY)
        (slot height (default 15))
        (slot basesize)
        (slot platformsize)
        (slot supports (multiple))
        (slot no_of_supports (default 0))
        (slot connectors (multiple))
        (slot no_of_connectors (default 0))
        (slot braces (multiple))
        (slot no_of_braces (default 0))
        (slot platform)
        (slot style (read-only)(allowed-symbols I A X))
        (slot cost (read-only))
        (slot strength (read-only))
        (slot assembly-time (read-only))
        (slot marketability (read-only)))

(defclass I-TOWER (is-a TOWER)
        (slot style (composite)(default I))
        (slot basesize (default 2))
        (slot platformsize (default 2))
        (slot cost (composite)(default 1))
        (slot strength (composite)(default 1))
        (slot assembly-time (composite)(default 1))
        (slot marketability (composite)(default 8)))

(defclass A-TOWER (is-a TOWER)
        (slot style (composite)(default A))
        (slot basesize (default 4))
        (slot platformsize (default 1))
        (slot cost (composite)(default 6))
        (slot strength (composite)(default 3))
        (slot assembly-time (composite)(default 6))
        (slot marketability (composite)(default 3)))

(defclass X-TOWER (is-a TOWER)
        (slot style (composite)(default X))
        (slot basesize (default 6))
        (slot platformsize (default 6))
        (slot cost (composite)(default 4))
        (slot strength (composite)(default 5))
        (slot assembly-time (composite)(default 7))
        (slot marketability (composite)(default 6)))
```

Appendix D

List of Rules

The following is a list of all of the rules included in **SNEAKERS**. They are separated alphabetically by aspect, then agent type. The name of each rule is given in all capitals, with a description of the rule following that.

```
;*********************************************************************
; ASSEMBLY_ANALYST.CLP
;*********************************************************************


;*********************************************************************
; CALC-ASSEMBLY-TIME - Calculates the total time to assemble the tower.
;*********************************************************************

;*********************************************************************
; GET-ASSEMTIME-WHEN-DONE - If design is finished, then calculate
;      assembly time, and inform the user.
;*********************************************************************

;*********************************************************************
; FIND-ASSEMBLY-COST - Calculate total assembly cost.
;*********************************************************************

;*********************************************************************
; ASSEMBLY-COST-NEEDED - If cost information needed, then calculate
;      assembly cost and inform the user.
;*********************************************************************
```

```
;**********************************************************************
; ASSEMBLY_CRITIC.CLP
;**********************************************************************


;**********************************************************************
; ASSEMBLY-WATCH-HEIGHT - If tower height is greater than 10 feet,
;      then inform the user that the tower is too high for assembly.
;**********************************************************************

;**********************************************************************
; FIND-CONNECTOR-TYPE - If a connector is added, then look up the type
;      of the connector.
;**********************************************************************

;**********************************************************************
; DONT-USE-WELD - If the connector is a weld, then inform the user not
;      to use welds because of their very long assembly times.
;**********************************************************************

;**********************************************************************
; DONT-USE-BOLT - If the connector is a bolt, then inform the user not
;      to use bolts because of their long assembly times.
;**********************************************************************




;**********************************************************************
; ASSEMBLY_SUGGESTOR.CLP
;**********************************************************************

;**********************************************************************
; ASSEMBLY-LOWER-TOWER - If the tower is too high for assembly,
;      then suggest that the tower be lowered to less than 10 feet.
;**********************************************************************

;**********************************************************************
; USE-SNAP - If a connector type was found unsatisfactory for assembly
;      reasons, suggest the user use a snap.
;**********************************************************************
```

```
;*********************************************************************
; COST_ANALYST.CLP
;*********************************************************************


;*********************************************************************
; FIND-COSTS - If the design is finished, then ask for cost
;      information.
;*********************************************************************

;*********************************************************************
; COUNT-COSTS - If all cost information is received, then add the
;      costs and inform the user.
;*********************************************************************




;*********************************************************************
; COST_CRITIC.CLP
;*********************************************************************


;*********************************************************************
; WATCH-COSTS-NORMAL - If the tower cost ratio is greater than 4 and
;      the importance of minimizing costs is high or extreme, then
;      inform the user that he has chosen an expensive tower style.
;*********************************************************************

;*********************************************************************
; WATCH-COSTS-EXTREME - If the tower cost ratio is greater than 6 and
;      the importance of minimizing costs is average then inform the
;      user that the chosen tower style is expensive.
;*********************************************************************

;*********************************************************************
; TOWER-COST - If the tower has been created, then look up its cost
;      ratio.
;*********************************************************************
```

```
;**********************************************************************
; COST_EVALUATOR.CLP
;**********************************************************************


;**********************************************************************
; MAXIMUM-COST-VALUE1 - If the importance of minimizing cost is none,
;      then set the maximum cost rating to 100.
;**********************************************************************

;**********************************************************************
; MAXIMUM-COST-VALUE2 - If the importance of minimizing cost is low,
;      then set the maximum cost rating to 90.
;**********************************************************************

;**********************************************************************
; MAXIMUM-COST-VALUE3 - If the importance of minimizing cost is
;      average, then set the maximum cost rating to 80.
;**********************************************************************

;**********************************************************************
; MAXIMUM-COST-VALUE4 - If the importance of minimizing cost is high,
;      then set the maximum cost rating to 65.
;**********************************************************************

;**********************************************************************
; MAXIMUM-COST-VALUE5 - If the importance of minimizing cost is
;      extreme, then set the maximum cost rating to 50.
;**********************************************************************

;**********************************************************************
; MEETS-MAX-COST - If the total cost rating is less than or equal to
;      the maximum cost rating, then inform user that the tower meets
;      the cost constraints.
;**********************************************************************

;**********************************************************************
; FAILS-MAX-COST - If the total cost rating greater than the maximum
;      cost rating, then inform user that the tower fails to meet the
;      cost constraints.
;**********************************************************************
```

```
;*********************************************************************
; COST_SUGGESTOR.CLP
;*********************************************************************


;*********************************************************************
; CUT-COSTS - If the chosen tower style is expensive, then suggest
;      changing to an I style tower.
;*********************************************************************




;*********************************************************************
; DESIGN_ADVISOR.CLP
;*********************************************************************


;*********************************************************************
; UNDO-FACTS - If an undo command is given, remove all facts that were
;      associated with the last user action prior to the undo.
;*********************************************************************

;*********************************************************************
; STARTUP-RULE - If the system is started, inform the user to start a
;      new design session.
;*********************************************************************

;*********************************************************************
; NEW-DESIGN-STARTED - If a value is given for required weight, then a
;      new design has been started.
;*********************************************************************

;*********************************************************************
; HEAVY-LOAD - If the required weight is greater than 5000 lbs, then
;      advise the user to use and A or X style tower.
;*********************************************************************

;*********************************************************************
; WATCH-ACID-RAIN - If the tower dimensions are set and there is acid
;      rain, then advise the user to use wood for the supports.
;*********************************************************************

;*********************************************************************
; TOO-MUCH-RAIN-FOR-WOOD - If the tower dimensions are set, there is no
;      acid rain, and the rainfall is greater than or equal to 48
;      inches/year, then advise the user not to use wood.
;*********************************************************************

;*********************************************************************
; GET-OVER-THE-SMOG - If the tower has been created and there is smog,
;      then advise the user to make a tower over 15 feet high.
;*********************************************************************




;*********************************************************************
; DESIGN_ANALYST.CLP
;*********************************************************************


;*********************************************************************
; TIME-TO-CONSTRUCT - Add assembly time and manufacturing time.
;*********************************************************************

;*********************************************************************
; FIND-STRENGTH - determine the strength ratio of the tower from the
;      strengths of the pieces and the type of tower.
;*********************************************************************

;*********************************************************************
; BUCKLE-LOAD - Calculate the maximum weight that can be supported by
;      the tower without buckling.
;*********************************************************************

;*********************************************************************
; CHECK-BUCKLING - If the design is complete, then inform the user of
;      the maximum weight that can be supported by the tower without
;      buckling.
;*********************************************************************

;*********************************************************************
; WIND-LOAD - Convert wind load to wind speed.
;*********************************************************************

;*********************************************************************
; FIND-MAX-WIND - Calculate the maximum wind speed that can be
;      withstood by the tower without toppling.
;*********************************************************************
```

```
;*********************************************************************
; CHECK-TOPPLING - If the design is complete, then inform the user of
;      the maximum wind speed that can be withstood by the tower
;      without toppling.
;*********************************************************************




;*********************************************************************
; DESIGN_CRITIC.CLP
;*********************************************************************


;*********************************************************************
; SQRDISTANCE - Determine the square of the distance between two
;      points.
;*********************************************************************

;*********************************************************************
; LEG-DONE - Calculate whether or not a support completes a leg.
;*********************************************************************

;*********************************************************************
; OLD-LEG - Determine whether or not the leg being worked on is the
;      last one that was not completed.
;*********************************************************************

;*********************************************************************
; LAST-ADDED-SUPPORT - If a support is added, then mark it as the
;      newest support in the design.
;*********************************************************************

;*********************************************************************
; FINISH-A-LEG - If a new support is added and it is not part of the
;      last incomplete leg, then inform user not to start a new leg
;      until the other is finished.
;*********************************************************************

;*********************************************************************
; RETRACT-SUPPORT-FACT - If there is a newest support, retract that
;      fact after all other rules have fired.
;*********************************************************************
```

```
;*********************************************************************
; DESIGN_EVALUATOR.CLP
;*********************************************************************


;*********************************************************************
; ENOUGH-TIME - If the time to construct is less than or equal to
;      the required time, then inform that the user the tower can be
;      constructed on time.
;*********************************************************************

;*********************************************************************
; NOT-ENOUGH-TIME - If the time to construct is greater than the
;      required time, then inform the user that the tower cannot be
;      constructed on time.
;*********************************************************************

;*********************************************************************
; TOO-MUCH-WEIGHT - If the maximum load is less than the required load,
;      then inform the user that the tower will buckle.
;*********************************************************************

;*********************************************************************
; NO-WEIGHT-PROBLEM - If the maximum load is greater than or equal to
;      the required load, then inform the user that the tower will not
;      buckle.
;*********************************************************************

;*********************************************************************
; TOO-MUCH-WIND - If the maximum wind speed is less than the required
;      wind speed, then inform the user that the tower will topple.
;*********************************************************************

;*********************************************************************
; NO-WIND-PROBLEM - If the maximum wind speed is greater than or equal
;      to the required wind speed , then inform the user that the tower
;      will not topple.
;*********************************************************************




;*********************************************************************
; DESIGN_SUGGESTOR.CLP
```

```
;***********************************************************************

;***********************************************************************
; CONTINUE-FIRST-LEG - If a leg was started with another unfinished,
;      then suggest that the user back up and finish the other leg.
;***********************************************************************



;***********************************************************************
; DISPOSAL_ADVISOR.CLP
;***********************************************************************

;***********************************************************************
; BEST-MATERIAL - If tower dimensions are set, then suggest the tower
;      be made out of wood because that is the best material for
;      disposal.
;***********************************************************************



;***********************************************************************
; DISPOSAL_ANALYST.CLP
;***********************************************************************

;***********************************************************************
; FIND-DISPOSAL-COST - Calculates the cost of disposal of all the
;      parts of the tower.
;***********************************************************************

;***********************************************************************
; DISPOSAL-COST-NEEDED - If cost information is needed, then
;      calculate the disposal cost and inform the user.
;***********************************************************************



;***********************************************************************
```

```
; DISPOSAL_CRITIC.CLP
;*********************************************************************


;*********************************************************************
; NOT-ONLY-MATERIAL - Determine if there are any rods of different
;      material than the one passed to the function.
;*********************************************************************

;*********************************************************************
; SAME-MATERIAL - If support added and it is not of the same material
;      as the other supports, then there is more than 1 material.
;*********************************************************************

;*********************************************************************
; NO-MULTI-MATERIALS - If there is more than 1 material, then inform
;      the user that the last material choice was bad, because it was
;      different from the other materials.
;*********************************************************************

;*********************************************************************
; CANT-DISPOSE-WELDS - If the connector chosen is a weld, then inform
;      the user that welds are a bad choice because they are hard to
;      dispose of.
;*********************************************************************




;*********************************************************************
; DISPOSAL_SUGGESTOR.CLP
;*********************************************************************


;*********************************************************************
; USE-ONE-MATERIAL - If the last material added was a bad choice, then
;      suggest that the user use only 1 material in the design.
;*********************************************************************

;*********************************************************************
; USE-SNAP-OR-BOLT-IN-DISPOSAL - If a weld was used in the design, then
;      suggest that the user change to a snap or bolt.
;*********************************************************************
```

```
;*************************************************************************
; MANUFACTURING_ANALYST.CLP
;*************************************************************************


;*************************************************************************
; CALC-MANUFACT-TIME - Calculate the time to manufacture the components
;      of the tower design.
;*************************************************************************

;*************************************************************************
; GET-TIME-WHEN-DONE - If tower design finished, then calculate the
;      manufacturing time and inform the user.
;*************************************************************************

;*************************************************************************
; FIND-MANUFACT-COST - Calculate the cost of manufacturing.
;*************************************************************************

;*************************************************************************
; MANUFACT-COST-NEEDED - If cost information is needed, then find
; the manufacturing cost and inform the user.
;*************************************************************************




;*************************************************************************
; MANUFACTURING_CRITIC.CLP
;*************************************************************************


;*************************************************************************
; INFORM-ILLEGAL-CONNECTOR - Inform the user if a connector is illegal
;      for the support it connects.
;*************************************************************************

;*************************************************************************
; NO-ILLEGAL-CONNECTORS - If connector added, then inform the user
;      if the connector is illegal for the support it connects.
;*************************************************************************

;*************************************************************************
; INFORM-ILLEGAL-SUPPORT1 - Inform the user if a support is illegal
;      for the connector to which it attaches its start point.
;*********************************************************************

;*********************************************************************
; INFORM-ILLEGAL-SUPPORT2 - Inform the user if a support is illegal
;      for the connector to which it attaches its end point.
;*********************************************************************

;*********************************************************************
; NO-ILLEGAL-SUPPORTS - If support added, then inform the user if the
;      support is illegal for the connector to which it attaches.
;*********************************************************************




;*********************************************************************
; MANUFACTURING_SUGGESTOR.CLP
;*********************************************************************


;*********************************************************************
; REMOVE-ILLEGAL-CONNECTOR - If an illegal connector is added, then
;      suggest that it be removed.
;*********************************************************************

;*********************************************************************
; REMOVE-ILLEGAL-SUPPORT - If an illegal support is added, then
;      suggest that it be removed.
;*********************************************************************




;*********************************************************************
; MARKETING_ADVISOR.CLP
;*********************************************************************


;*********************************************************************
; SLEEK-TOWER - If design session started, then advise the user to
;      use an X or I style tower.
;*********************************************************************
```

```
;**********************************************************************
; PACKAGING_CRITIC.CLP
;**********************************************************************


;**********************************************************************
; DONT-PACK-WELDS - If a weld is used, then inform the user not to use
;      welds.
;**********************************************************************




;**********************************************************************
; PACKAGING_SUGGESTOR.CLP
;**********************************************************************


;**********************************************************************
; PACK-SNAP-OR-BOLT - If a weld is used, then suggest that the user
;      switch to snaps or bolts.
;**********************************************************************




;**********************************************************************
; SAFETY_ANALYST.CLP
;**********************************************************************


;**********************************************************************
; SAFETY-VALUE - Determine the safety value of the tower.
;**********************************************************************

;**********************************************************************
; SAFETY-ANALYSIS - If tower design is finished, then determine the
;      safety value and inform the user.
;**********************************************************************
```

```
;**********************************************************************
; SAFETY_CRITIC.CLP
;**********************************************************************


;**********************************************************************
; TOWER-HEIGHT - If tower dimensions are set, then look up the tower
;      height.
;**********************************************************************

;**********************************************************************
; TOWER-PLATFORM-WIDTH - If tower dimensions are set, then look up the
;      tower platform size..
;**********************************************************************

;**********************************************************************
; WATCH-HEIGHT1 - If tower height is greater than or equal to 12 feet
;      and the platform is less than or equal to 3 feet, then inform
;      the user that the tower is too high for the platform size.
;**********************************************************************

;**********************************************************************
; WATCH-HEIGHT2 - If tower height is greater than 15 feet, then inform
;      the user that the tower is much too high.
;**********************************************************************




;**********************************************************************
; SAFETY_EVALUATOR.CLP
;**********************************************************************


;**********************************************************************
; SAFETY-EVALUATION1 - If safety rating is less than 0, then inform the
;      user that the tower is unsafe.
;**********************************************************************

;**********************************************************************
; SAFETY-EVALUATION2 - If safety rating is greater than or equal to 0,
;      then inform the user that the tower is safe.
;**********************************************************************
```

```
;**********************************************************************
; SAFETY_SUGGESTOR.CLP
;**********************************************************************


;**********************************************************************
; LOWER-TOWER - If the tower is much too high then suggest that the
;      user lower the tower to under 16 feet.
;**********************************************************************

;**********************************************************************
; INCREASE-OR-LOWER - If the tower is too high for the platform size,
;      then suggest that the user either increase the platform to
;      more than 3 feet, or decrease the tower height to less than 12
;      feet.
;**********************************************************************




;**********************************************************************
; USER_RULES.CLP - rules which are directly affected by user
;                  action and contain no domain knowledge
;**********************************************************************

;**********************************************************************
; INFORM-ACTION - If the user does something, then echo the action to
;      the screen.
;**********************************************************************

;**********************************************************************
; PRINT-REPORT - Prints out a prts list and assembly sequence to the
;      file "sneakers.rpt".
;**********************************************************************

;**********************************************************************
; GENERATE-REPORT - If design session is finished, then print report.
;**********************************************************************

;**********************************************************************
; PRINT-NEXT-ASSEMBLED - Print the object that gets assembled at the
;      given time.
```

```
;**********************************************************************

;**********************************************************************
; PRINT-ASSEMBLY - Rule which loops, printing the assembly sequence.
;**********************************************************************

;**********************************************************************
; END-PRINT-ASSEMBLY - If finished printing, close file.
;**********************************************************************
```

Appendix E

Program Modules

The following is a listing of the headers for the C functions used in **SNEAKERS**. They are separated by module and included for those interested in modifying the source code.

```
/*******************************************************************
 * ADD_PARTS.C - this module contains functions which are used in   *
 * adding user selected parts to the design of the tower.          *
 *                                                                 *
 * Written by : Rob Douglas                                        *
 *******************************************************************/


/*******************************************************************
 * void add_connector(PointStr, char *)                            *
 *     adds a connector of a specific type to the tower at the     *
 *     chosen point.                                               *
 *******************************************************************/

/*******************************************************************
 * void add_rod(PointStr, int, char *)                             *
 *     adds a rod of given length and material to tower at point   *
 *     given.                                                      *
 *******************************************************************/

/******************************************************************
 * void add_bracing(PointStr, PointStr)                           *
 *     adds bracing between the two given points.                 *
 ****************************************************************/

/****************************************************************
 * void find_connector(PointStr, DATA_OBJECT **)                *
 *     finds the connector which is located at the point given, *
 *     returning a pointer to the connector in the DATA_OBJECT. *
 ****************************************************************/

/****************************************************************
 * void find_rod(PointStr, DATA_OBJECT **)                      *
 *     the find the rod that ends at the point given, and return *
 *     it in DATA_OBJECT.                                       *
 ****************************************************************/

/****************************************************************
 * void include_rod(PointStr, PointStr, PointStr, int, char *)  *
 *     fill in slots for a rod and create an instance for it, then *
 *     include it in the tower.                                 *
 ****************************************************************/

/****************************************************************
 * void get_destination(DATA_OBJECT *, PointStr, PointStr *)    *
 *     find the ultimate destination of the rod given, based on *
 *     tower style and other rods in the tower, and their       *
 *     destinations.  This destination is usually out of reach of *
 *     a single rod, but is the guide for the combination of rods. *
 *     If NULL is given as the rod, generate a new destination. *
 ****************************************************************/

/****************************************************************
 * VOID *get_point_instance(PointStr)                           *
 *     returns a pointer to the instance of a point at the      *
 *     location given, returns NULL if there is no such point   *
 *     instance.                                                *
 ****************************************************************/

/****************************************************************
 * PointStr get_endpoint(PointStr,PointStr,DATA_OBJECT *,       *
 *                       DATA_OBJECT *,int)                     *
 *     finds where the end point of a newly placed rod should be, *
 *     including the angle at which it should be placed.        *
 ****************************************************************/

/****************************************************************
 * PointStr find_point(float,float,PointStr,PointStr,PointStr,int) *
 *     uses geometry and given angles to determine the numeric  *
 *     values of the point to be found.                         *
 ******************************************************************/

/******************************************************************
 * PointStr find_easy_point(PointStr, PointStr, int)              *
 *     uses simple geometry for non-complicated angles to find a   *
 *     point.                                                     *
 ******************************************************************/

/******************************************************************
 * void add_platform_point(PointStr)                              *
 *     adds one of the corners of the platform to that instance    *
 *     necessary.                                                 *
 ******************************************************************/




/******************************************************************
 * CLIPS_FUN.C - this module contains functions which access CLIPS *
 * objects, and are used by CLIPS.                                *
 *                                                                *
 * Written by : Rob Douglas                                       *
 ******************************************************************/


/******************************************************************
 * void inform_user()                                             *
 *     this is a function called by CLIPS expert systems to output *
 *     a message to the screen, and thereby the user.             *
 ******************************************************************/

/******************************************************************
 * void add_start_points()                                        *
 *     adds the first rod placement points to the list of valid    *
 *     rod points.                                                *
 ******************************************************************/

/******************************************************************
 * PointStr return_3D(int, int, int *)                            *
 *     returns a valid 3D point from the design and increases i    *
 *     if a point exists.  Otherwise, returns an ignored value and *
 *     does not increase i.                                       *
 ******************************************************************/

/******************************************************************
 * short connector_at(VOID *)                                     *
 *     returns 1 if there exists a connector at the given          *
 *     position.                                                   *
 *****************************************************************/




/*****************************************************************
 * DRAW_FUN.C - this module contains most of the functions needed  *
 * to draw the pieces of the tower during graphics rendering.      *
 *                                                                 *
 * Written by : Rob Douglas                                        *
 *****************************************************************/


/*****************************************************************
 * int DrawFilledCircle(int, int, int, int)                        *
 *     draws a filled circle of given color, of a given diameter,  *
 *     centered a given point.                                     *
 *****************************************************************/

/*****************************************************************
 * int DrawThickLine(int, int, int, int, int, int)                 *
 *     draws a line of given color and thickness from one given    *
 *     point to another.                                           *
 *****************************************************************/

/*****************************************************************
 * void cross(PointStr, PointStr, PointStr *)                      *
 *     returns the cross product of two three-dimensional vectors. *
 *****************************************************************/

/*****************************************************************
 * float dot(PointStr, PointStr)                                   *
 *    returns the dot product of two three-dimensional vectors.    *
 *****************************************************************/

/*****************************************************************
 * void Normalize (PointStr *)                                     *
 *     normalizes a given vector, giving it a length of 1.         *
 *****************************************************************/

/*****************************************************************
 * void BuildMatrices(PointStr, PointStr, PointStr, PointStr)      *
 *     builds transformation matrices from vectors describing      *
 *      viewing position and arrangements.                         *
 *****************************************************************/

/*****************************************************************
 * void InitViewVectors(int)                                     *
 *     sets up the viewing vectors based on the viewing angle    *
 *     selected.                                                 *
 *****************************************************************/

/*****************************************************************
 * void transfer(PointStr)                                       *
 *     changes coordinates from world values to device values.   *
 *****************************************************************/

/*****************************************************************
 * void Cnvt3DTo2D(PointStr *, XPoint *)                         *
 *     converts a three dimensional point to a two dimensional   *
 *     projection.                                               *
 *****************************************************************/

/*****************************************************************
 * void DrawROD(DATA_OBJECT)                                     *
 *     draws the rod passed onto the screen.                     *
 *****************************************************************/

/*****************************************************************
 * void DrawConnector(DATA_OBJECT)                               *
 *     draws the connector passed onto the screen.               *
 *****************************************************************/

/*****************************************************************
 * void DrawPlatform()                                           *
 *     draw the platform on top of the completed tower.          *
 *****************************************************************/

/*****************************************************************
 * void Draw_A(int, int, int)                                    *
 *     draws an abstract version of an A-tower of given dimensions.*
 *****************************************************************/

/*****************************************************************
 * void Draw_I(int, int, int)                                    *
 *     draws an abstract version of an I-tower of given dimensions.*
 *****************************************************************/

/*****************************************************************
 * void Draw_X(int, int, int)                                    *
 *     draws an abstract version of an X-tower of given dimensions.*
 ******************************************************************/

/******************************************************************
 * void Render()                                                  *
 *     calls the drawing functions to draw the various pieces of  *
 *     the tower.                                                 *
 ******************************************************************/

/******************************************************************
 * void DrawGrid()                                                *
 *     Draws the lines that make a drawing area grid which is the *
 *     floor.                                                     *
 ******************************************************************/




/******************************************************************
 * MATH_FUN.C - this module contains several useful math functions *
 * used in other modules.                                         *
 *                                                                *
 * Written by : Rob Douglas                                       *
 ******************************************************************/


/******************************************************************
 * float sqr(float)                                               *
 *     squares a given number.                                    *
 ******************************************************************/

/******************************************************************
 * float find_angle(PointStr, PointStr)                           *
 *     returns the angle between two vectors.                     *
 ******************************************************************/

/******************************************************************
 * float vector_length(PointStr *)                                *
 *     returns the length of a vector.                            *
 ******************************************************************/

/******************************************************************
 * int swap_row(float **, int, int, int, int)                     *
 *     a useful utility function which is used by the inversion   *
 *     and Gaussian functions.  It simply swaps the rows whose    *
 *     indexes are passed to it in the vector passed in.  Out of  *
 *     bounds is checked, and will return an error.               *
 *     Written by Jeff Choate.  Modified by Rob Douglas.          *
 *****************************************************************/

/*****************************************************************
 * void gaussian_elimination(float **, int, int, float **)        *
 *     This function does Gaussian elimination.  It is written by  *
 *     Jeff Choate with no help from the books.  Therefore, this   *
 *     one is the one to follow if you do not understand the       *
 *     process.  The algorithm goes right down the diagonal        *
 *     elimination all entries below it, and making itself 1 for   *
 *     possible later solutions.  It can be used to quickly check  *
 *     linear dependence, for if any diagonal entry is 0 after the *
 *     algorithm finishes, the matrix has some linear dependance.  *
 *     This algorithm is not as numerically precise as the previous*
 *     inverter, for it does no row swapping unless the diagonal   *
 *     entry is zero.  And only then because you have to.  This is *
 *     a big no no, for it leads to big roundoff problems.  Also   *
 *     the row is divided by the pivoting column first, which only *
 *     leads to numeric problems.  Therefore I have left this in   *
 *     not so much for use as for reference.  It could be easily   *
 *     changed to be just as precise as the previous inverter was. *
 *     This is a non-destructive function, so the original matrix  *
 *     will be preserved. However, the result of the inversion will*
 *     be stored in the result array which must be previously      *
 *     allocated. It has to be at least as big as                  *
 *     xdim * ydim * sizeof(float).                                *
 *     Written by Jeff Choate.  Modified by Rob Douglas.          *
 *****************************************************************/

/*****************************************************************
 * void back_substitute(float **, int, float **)                  *
 *     performs back substitution on a matrix that has been        *
 *     diagonalized by Gaussian elimination.                       *
 *****************************************************************/

/*****************************************************************
 * int vecmult(float *, float **, float *)                        *
 *  This routine multiplies a (4X1) vector by a (4X4) matrix and   *
 *  puts the result in the (4X1) vector specified.                 *
 *****************************************************************/

/*****************************************************************
 * float pointdist(PointStr, PointStr)                            *
 *     returns the distance between two points in 3 space.        *
 *****************************************************************/
```

```
/*****************************************************************
 * MEMORY.C - this module contains functions which are used for    *
 * managing memory in the whole program.                           *
 *                                                                 *
 * Written by : Rob Douglas                                        *
 *****************************************************************/

/*****************************************************************
 * void *newitem(int)                                              *
 *     returns a generic pointer to a block of memory of the       *
 *     required size.                                              *
 *****************************************************************/

/*****************************************************************
 * void *olditem(void *,int)                                       *
 *     returns a generic pointer to a block of memory of the       *
 *     required size, reallocating for the old item.               *
 *****************************************************************/




/*****************************************************************
 * POINT.C - this module contains functions used in keeping track  *
 * of the points that the user might wish to place various objects.*
 *                                                                 *
 * Written by : Rob Douglas                                        *
 *****************************************************************/


/*****************************************************************
 * short pointlist_empty(int)                                      *
 *     checks whether or not one of the global points lists are    *
 *     empty.                                                      *
 *****************************************************************/

/*****************************************************************
 * void clear_pointlists()                                         *
 *     empties all of the global points lists.                     *
 *****************************************************************/

/*****************************************************************
 * void get_point(int)                                             *
 *     controls the selection of a point on the screen, checking   *
 *     to see if it is on the proper list.                         *
 *****************************************************************/

/*****************************************************************
 * void validate(int, int, int, PointStr **)                       *
 *    checks whether or not a certain point is on one of the global*
 *     lists,   *and removes it if it is, returning the 3D value   *
 *     of the point.                                               *
 *****************************************************************/

/*****************************************************************
 * void add_valid_point(int, PointStr *)                           *
 *    adds a point to the proper list.                             *
 *****************************************************************/

/*****************************************************************
 * void add_A_points(int, int, int)                                *
 *     adds the initial points for an A-tower of given dimensions  *
 *     to the global rod points list.                              *
 *****************************************************************/

/*****************************************************************
 * void add_I_points(int, int, int)                                *
 *     adds the initial points for an I-tower of given dimensions  *
 *     to the global rod points list.                              *
 *****************************************************************/

/*****************************************************************
 * void add_X_points(int, int, int)                                *
 *     adds the initial points for an X-tower of given dimensions  *
 *     to the global rod points list.                              *
 *****************************************************************/

/*****************************************************************
 * int get_two_points(PointStr *)                                  *
 *     get two points which could be used to place bracing.        *
 *****************************************************************/




/*****************************************************************
 * SCREEN.C - contains all of the functions which control the user *
 * interface including all callbacks and any functions which access*
 * widgets directly.                                               *
 *                                                                 *
 * Written by : Rob Douglas and VUIT                               *
 *******************************************************************/


/*******************************************************************
 * UserFunctions()                                                 *
 *      necessary function when including CLIPS in another C       *
 *      program.  Describes the user defined functions that may be *
 *      CLIPS expert systems.                                      *
 *******************************************************************/

/*******************************************************************
 * void init_datastructs()                                         *
 *     initializes some of the global variables.                   *
 *******************************************************************/

/*******************************************************************
 * void create_procedure(Widget, int *, unsigned long *)           *
 *     called by most widgets when they are created to add them to *
 *     the widget array so they can be easily accessed.            *
 *******************************************************************/

/*******************************************************************
 * void quit_button_press(Widget, int *, unsigned long *)          *
 *     callback for exiting the program when quit button is        *
 *     pressed.                                                    *
 *******************************************************************/

/*******************************************************************
 * void DisplayTower(Widget, int *, unsigned long *)               *
 *     callback to display the tower when the drawing area is in   *
 *     view.                                                       *
 *******************************************************************/

/*******************************************************************
 * void InitGraphics(Widget, int *, unsigned long *)               *
 *     opens display and sets up drawing area to act as graphics   *
 *     window.                                                     *
 *******************************************************************/

/*******************************************************************
 * void new_button_press(Widget, int *, unsigned long *)           *
 *     callback for when the new button is pressed.                *
 *******************************************************************/

/*******************************************************************
```

```
 * void requirements_reset(Widget, int *, unsigned long *)        *
 *     callback to reset the requirements selections to their     *
 *     default values.                                             *
 ******************************************************************/

/*****************************************************************
 * void requirements_ok(Widget, int *, unsigned long *)           *
 *      callback to accept selected requirements.                  *
 ******************************************************************/

/*****************************************************************
 * void requirements_cancel(Widget, int *, unsigned long *)       *
 *     callback to cancel selection of requirements and return to  *
 *     previous point in the program.                              *
 ******************************************************************/

/*****************************************************************
 * void colormap_ok(Widget, int *, unsigned long *)               *
 *     callback to acknowledge that the black and white version    *
 *     does not have all of the features of the color version.     *
 ******************************************************************/

/*****************************************************************
 * void colormap_cancel(Widget, int *, unsigned long *)           *
 *     exit program if black and white screen is not acceptable.   *
 ******************************************************************/

/*****************************************************************
 * void activate_aspect(int)                                       *
 *     turns on an aspect or agent button, and highlights it.      *
 ******************************************************************/

/*****************************************************************
 * void aspect_pressed(Widget, int *, unsigned long *)            *
 *     callback for when one of the aspect buttons is pressed.     *
 ******************************************************************/

/*****************************************************************
 * void agent_pressed(Widget, int *, unsigned long *)             *
 *     callback for when one of the agent buttons is pressed.      *
 ******************************************************************/

/*****************************************************************
 * void return_pressed(Widget, int *, unsigned long *)            *
 *     callback to return from the agent specific information      *
 *     window.                                                     *
 ******************************************************************/
```

```
/*****************************************************************
 * void deactivate_aspect(int)                                   *
 *     unhighlight an aspect or agent button when new information *
 *     is no longer available.                                   *
 *****************************************************************/

/*****************************************************************
 * void tower_style_select(Widget, int *, unsigned long *)       *
 *     callback for abstract tower type selections from palette.  *
 *****************************************************************/

/*****************************************************************
 * void redraw()                                                 *
 *     force a refresh of the screen.                            *
 *****************************************************************/

/*****************************************************************
 * void change_int_scale(Widget, int *, XmScaleCallbackStruct *)   *
 *     callback assigned to handle changes in the values of the   *
 *     intermediate level scales, which determine the dimensions  *
 *     of the tower.                                             *
 *****************************************************************/

/*****************************************************************
 * void manage_scales()                                          *
 *     set the initial values of the intermediate level, dimension *
 *     selecting scales, before they are managed.                *
 *****************************************************************/

/*****************************************************************
 * void intermediate_accept(Widget, int *, unsigned long *)      *
 *     accept intermediate values chosen, and propagate the      *
 *     values chosen.                                            *
 *****************************************************************/

/*****************************************************************
 * void Undo(Widget, int *, unsigned long *)                     *
 *     callback for when undo is selected, starts all of the     *
 *     necessary functions for removing pieces and resetting     *
 *     values.                                                   *
 *****************************************************************/

/*****************************************************************
 * void desense_place_buttons(Widget, int *, unsigned long *)    *
 *     turn sensitivity off for palette buttons during detailed  *
 *     design.                                                   *
 ******************************************************************/

/******************************************************************
 * void place_rod(Widget, int *, unsigned long *)                *
 *     perform functions necessary for placing a rod into the tower*
 *      design.                                                  *
 ******************************************************************/

/******************************************************************
 * void place_bracing(Widget, int *, unsigned long *)            *
 *     perform functions necessary for placing bracing into the  *
 *     tower design.                                             *
 ******************************************************************/

/******************************************************************
 * void place_connector(Widget, int *, unsigned long *)          *
 *     perform functions necessary for placing a connector into  *
 *     the tower design.                                         *
 ******************************************************************/

/******************************************************************
 * void place_platform(Widget, int *, unsigned long *)           *
 *     perform functions necessary for placing the platform at the *
 *     top of the tower.                                         *
 ******************************************************************/

/******************************************************************
 * void set_placing_buttons()                                    *
 *     set the sensitivity of the palette buttons during detailed *
 *     design on.                                                *
 ******************************************************************/

/******************************************************************
 * void rod_select_ok(Widget, int *, unsigned long *)            *
 *     accept the selected parameters for the rod to be placed.  *
 ******************************************************************/

/******************************************************************
 * void rod_select_cancel(Widget, int *, unsigned long *)        *
 *     cancel placing of a rod.                                  *
 ******************************************************************/

/******************************************************************
 * void connector_select_ok(Widget, int *, unsigned long *)      *
 *     accept the selected parameters for the connector to be    *
 *     placed.                                                   *
 ******************************************************************/
```

```
/**************************************************************
 * void connector_select_cancel(Widget, int *, unsigned long *)    *
 *     cancel placing of a connector.                              *
 **************************************************************/

/**************************************************************
 * void out_to_screen(char *, char *, char *, char *)              *
 *     outputs recommendations to themessages window, and stores   *
 *     information for the individual agents' output.              *
 **************************************************************/

/**************************************************************
 * void show_aspects()                                             *
 *     activate those aspect buttons that have new information.    *
 **************************************************************/

/**************************************************************
 * void show_agents()                                              *
 *     activate the agent buttons that have new information.       *
 **************************************************************/

/**************************************************************
 * void list_selection(Widget, int *, XmListCallbackStruct  *)     *
 *     callback for changing values of selections for attributes   *
 *     of objects added to the design.                             *
 **************************************************************/

/**************************************************************
 * void change_view_scale(Widget, int *, XmScaleCallbackStruct *)  *
 *     rotate the view of the tower as selected by the rotation    *
 *     bar.                                                        *
 **************************************************************/

/**************************************************************
 * void change_information()                                       *
 *     changes the information in the information text window.     *
 **************************************************************/

/**************************************************************
 * void DeleteAgents()                                             *
 *     Removes all data from agent lists.                          *
 **************************************************************/
```

```
/*****************************************************************
 * TOYS.C - contains functions which affect various objects in     *
 * COOL, the CLIPS Object Oriented Language.                       *
 *                                                                 *
 * Written by : Rob Douglas                                        *
 *****************************************************************/


/*****************************************************************
 * void change_height(int)                                         *
 *     changes the height of the tower.                            *
 *****************************************************************/

/*****************************************************************
 * void change_base(int)                                           *
 *     changes the base of the tower.                              *
 *****************************************************************/

/*****************************************************************
 * void change_platform(int)                                       *
 *     changes the platform size of the tower.                     *
 *****************************************************************/

/*****************************************************************
 * float choose_valid_angle(float, DATA_OBJECT)                    *
 *     returns the closest angle to the given angle, allowed by    *
 *     the given connector.                                        *
 *****************************************************************/

/*****************************************************************
 * VOID *get_last_instance()                                       *
 *     gets a pointer to the last instance created, for use in     *
 *     undoing.                                                    *
 *****************************************************************/

/*****************************************************************
 * short undo_last_added()                                         *
 *     removes the last object added to the tower.                 *
 *****************************************************************/

/*****************************************************************
 * short member_of_class(char *, VOID *)                           *
 *     checks whether or not an instance is a member of a given    *
 *     class.  This had to be written because of a bug in the      *
 *     CLIPS code which normally would perform this function.      *
 *****************************************************************/
```

```
/******************************************************************
 * void remove_from_tower(VOID *)                                 *
 *     takes care of superficial details concerned with removing a *
 *     piece from the tower.                                      *
 ******************************************************************/

/******************************************************************
 * void replace_valid_point(VOID *)                               *
 *     replaces the points taken off of the global point lists by *
 *     adding a piece to the tower, when that piece is removed.   *
 ******************************************************************/
```