Applying Information Technologies to Architectural User Programming

by

Charles M. Dalsass

S.B., Civil Engineering (1995)
University of Connecticut

Submitted to the
Department of Civil and Environmental Engineering in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE IN CIVIL AND ENVIRONMENTAL ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May, 1997

Signature of the Author.....
Department of Civil and Environmental Engineering
May 9, 1997

Certified by.....
Jerome J. Connor
Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by.....
Joseph M. Sussman
Chairman, Departmental Committee on Graduate Studies

Applying Information Technologies to Architectural User Programming

by

Charles M. Dalsass

Submitted to the Department of Civil and Environmental Engineering
on May 9, 1997 in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Civil and Environmental Engineering

Abstract

This thesis presents analysis, propositions, and solutions for improving the quality and enhancing the performance of an existing business process, User Programming. User Programming is an architectural pre-design methodology used by Henn Architects (Munich, Germany). Information technologies will be strategically applied to different areas of this complicated and subtle process. The effort involves the creation of prototype software applications, using groupware and WWW technologies. The thesis represents phase one of Design Technology's work on the "CardWall" project for Henn Architects.

The study starts with a detailed investigation of User Programming, with consideration of schedule, documentation, roles and process. A description of the areas where improvements can be made follows. This information is the result of interviews with Henn programmers, and work from the Design Technology group. Next, this paper will investigate software concepts that follow from the targeted areas for improvement. These suggestions and concepts are clarified with the creation of two prototype software systems. A chapter on architecture outlines the major facets of the HennApplet information model. Finally, a technical section provides detailed explanation for design decisions and technical specifications concerning these prototypes.

Thesis Supervisor: Jerome J. Connor
Title: Professor of Civil and Environmental Engineering

Acknowledgements

I would like to thank Bill Porter and Paul Keel for providing encouragement, enlightenment and support throughout this project. Although I may have complained about technical areas of this work throughout the year, I am sure you both know that I have been extremely happy working in Design Technology. I've asked alot of people about their labs and their research work at MIT, and I honestly don't know of anyone else here who has such an interesting and enriching project as this. On account of my involvement with this project, MIT has been a blast for me.

This paper is dedicated to my terrific family: Eggs, Johnson, Munny, Noni, Di, Dan and Pop. Love!

Contents

List of Figures

Chapter 1 - Programming

User programming is a general term describing the acquisition of information that is required for a design at the initialization stages of a project. Often applied to architecture, programming is a methodology that is used to clarify and simplify the way that design is approached. Programming does not attempt to replace design with an algorithm. Rather, it provides a sequence and guidelines as well as a philosophy for information retrieval and client relations, which result in well thought out design decisions. The first part of this chapter will focus on specific variant of programming originally proposed by William Pena in *Problem Seeking, An Architectural Programming Primer.* The second part of this chapter will describe in detail how Henn Architekten Ingenieure practices programming.

1.1 William Pena's Programming from *Problem Seeking*

Like other forms of programming, the methodology detailed in *Problem Seeking* brings to the project the mandate that no design decisions are made before the problem is found and clearly stated. It provides mechanisms to avoid attempts at solving problems without making methodical endeavors to find out what those problems really are. The methods proposed in *Problem Seeking* intend to circumvent the common architectural error of attempting to find the problem and to solve it at the same time. To avoid this mistake, guidelines are proposed for the creation of a clearly defined design problem on a project. The result of programming (and pre-design in general) is a clear statement of what the problem is.

Problem Seeking splits the architectural endeavor into two parts. The first is programming and the second is design. "If programming is problem seeking, then design is problem solving" (Pena 1977). In the first stage, the programming, the goal is to arrive at the design problem statement. Having realized the problem statement, the designers may use that problem to create a design solution that solves the problem. This programming method distinctly separates out the two sections, and de-couples the programming (pre-design) from the design. What connects the two stages is the problem statement. Programming forces architects to make clear problem statements - a step which is universally supported, but rarely practiced in architectural projects. *Problem Seeking* also demands that the client be recognized as the main source of information for design decisions. To support this, programming supports intensive information retrieval from the client. This is done by a series of group meetings where members of the client organization are interviewed about certain aspects of the organization. They are encouraged to anonymously submit suggestions, complaints and facts about the present condition of their workspace.

The information that will dictate decisions about the design is in widespread and disparate forms within the client organization. Although it is very clear that the information about the design of a new building belongs to and is held by the client, the onerous task of retrieving and making sense of that information is the real obstacle in client relations. The programming process suggested in *Problem Seeking* outlines specific steps for

gathering and organizing this information. This is accomplished by retrieving the ideas found from the group sessions and categorizing them into a matrix. The categorization allows the designers to begin the next step of creating design suggestions for new or improved facilities.

"Programming is an organized process based on standard procedures which can be used on large and small projects, simple and complex building types and with single or multiple clients. It is particularly useful with large, complex and unfamiliar projects for large client groups"(Pena, 1977).

Problem Seeking splits programming into five clear steps. These are:

1. Establish Goals - clear, concise statements about the long term, strategic goals of the client.
2. Collect and Analyze Facts - find out facts that will govern the design, these are things that exist now.
3. Uncover and test Concepts - does the client have ideas about how to achieve these goals?
4. Determine Needs - what level of quality, room space, and quantity of production are required?

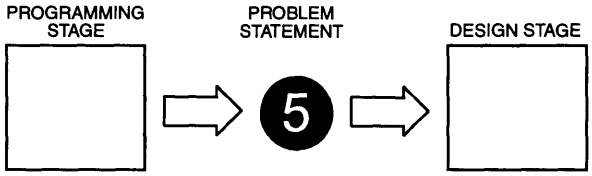


Figure 1-1: Separation of design from pre-design (Adapted from Pena, 1977).

5. State the Problem – what is the general aim of the project, where should it be going?

Steps one through three are the steps that are used to find out information from the client. The fourth step - determine needs, imposes and yields constraints on the problem by trading off specific needs with the budget. The needs are derived from the information found within the first three steps. The final step produces the main artifact of the programming process, the problem statement. Although classifying the information does not have to be done in any specific order, reaching the problem should be the last step.

For each of the five steps above, *Problem Seeking* suggests that the following four subclassifications are made. The goals, facts, concepts and needs are categorized by Function, Form, Economy and Time. Function is essentially what the building is going to be used for, and what needs the building must provide. Function focuses on what types of work are to be done within and by the building. Form, relates to the physical shape of the

building and how it fits into the physical and cultural surroundings. Economy is the quantity and distribution of money associated with the project. Time deals with the state of the building/site in the past, present and future.

This step-by-step classification procedure creates a matrix of information for the synthesis of the project into a cohesive and well thought out design problem. Performing this classification allows programmers to remember all the necessary components of each one of the steps. Within facts, goals, concepts, needs and the problem statement, the further classes can be used as a checklist for information that may have been forgotten or left out. This method also serves the purpose of organizing information into clear, concise and relevant pieces of information.

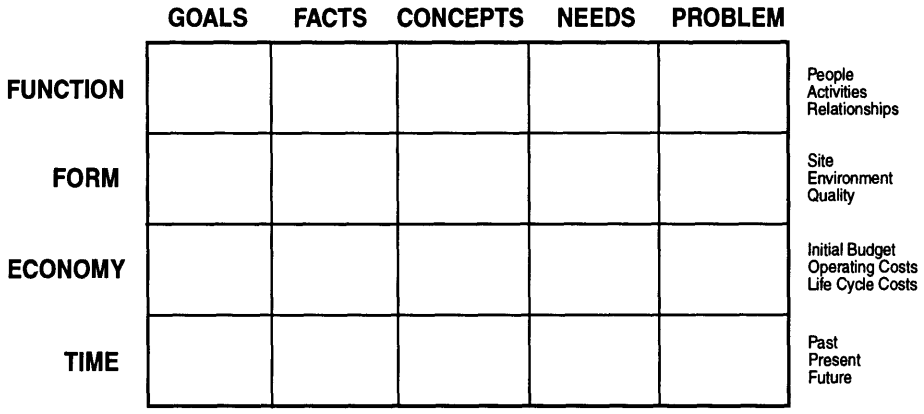


Figure 1-2: Programming matrix. (Adapted from Pena, 1977).

Problem Seeking also provides a philosophy for client/architect relations. The premise is that the client knows the goals, facts, concepts and needs better than anyone else does. Allowing the client to make decisions is another premise behind client/architect relations. For this reason, the user must take an intimate part in the programming process. The client must be coached to make decisions about what his needs are and what problem needs to be solved (this also reduces input from the architect). *Problem Seeking* suggests that the programming effort require two teams, one to represent the viewpoint of the architect and the other to represent the client's needs.

Problem Seeking also suggests making project cost estimates at an early time in the programming process. This is used as part of the data classification and gathering occurring in the first three steps of the process. This is done using a "BrownSheet", which is a large board outlining the types of space and cost requirements. The brown sheet contains visual representations of the types of rooms that are required for function and their

subsequent sizes. It is used to visually trade-off space requirements with cost.

Programming gets clients to make suggestions to help organize the disparate information regarding the project and to make preliminary estimates of area requirements. To do this, the client's people come together in a group where all participants may participate. As the client's members make these suggestions, their ideas are graphically represented using paper cards that are pasted upon a large board, (in this paper) called the CardWall. The cards usually contain a few words, characterizing the idea that the card represents. Each card should only represent one clear idea. As a group, the programmers and the client's people make decisions about the goals, facts, concepts, needs and the design problem. When the design problem is decided upon by the programmers, it is passed to the designers who will come up with a detailed design to solve the problem.

1.2 Henn Architekten Ingenieure

Henn architects in Munich Germany has taken many of the concepts of the programming process and used them in their everyday practice. In addition to following suggestions from the Pena book on format, style and procedure, the Henn group has made changes and additions to the user programming methodology. They have expanded the use of Pena's concepts as well. Where the programming process was originally proposed as a system to communicate with the client, organize data and make initial estimates of costs, it has become a marketing tool, an archiving tool to keep the history of the project, and a product for the client. Henn's clients such as VW of Wolfsberg and Skoda manufacturing of Poland consider the programming "CardWalls" to be valuable deliverables from Henn. The documentation produced from the cards and BrownSheets of the programming process creates a log of the entire design process in tangible form. Finally, the Henn group has codified the process and created sophisticated methods for using programming to help their clients achieve their goals.

Programming provides solutions for Henn Architects in the following areas:
1. As a way to design buildings
2. As a public relations and marketing tool
3. As a knowledge base
4. As a means of communication with the client (transfer of information)
5. As a facilities management tool
6. As an information system

1.2.1 Business Philosophy

The business model of the Henn Architects is that acting as the architect (a complete stranger), they have has no ability to make decisions regarding what the client needs at the onset of the project. Instead, the focus is to promote a high level of collaboration with the client. Henn Architects demand this type of collaboration because much of the decision making power is "passed back" to the client when Henn Architects obtains a contract. Rather than having most of the design decisions made by the architect, it is in both the architect and the client's best interest to have the client make decisions based on

what they need. This philosophy is carried out using the user programming sessions that involve many members in the organization. At all stages of the design, the client is asked to participate in the decisions affecting the design. When the design has been completed, the user is given copies of the documentation summarizing the decisions they have made.

The information that the client has about their needs, exists in various and disparate forms within the company. User programming and the Henn philosophy focus on collect-

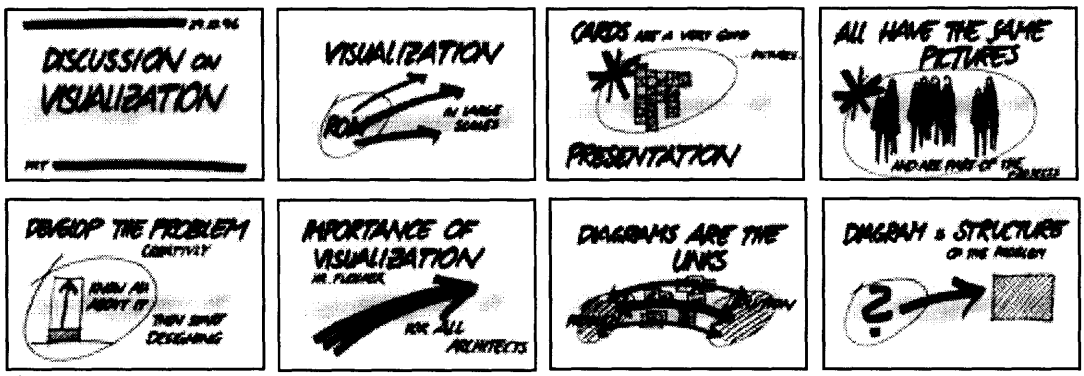


Figure 1-3: Cards from a discussion on visualization in architecture. Drawn by Christine Kohlert.

ing that information from within the ranks and applying it to the design. It does this with a highly egalitarian system, by accepting suggestions during programming from members of the organization at all levels with equal acceptance. The ideas that come from employees are not mapped to a specific author, and their look is consistent throughout. In the programming all ideas are given equal scrutiny through anonymity.

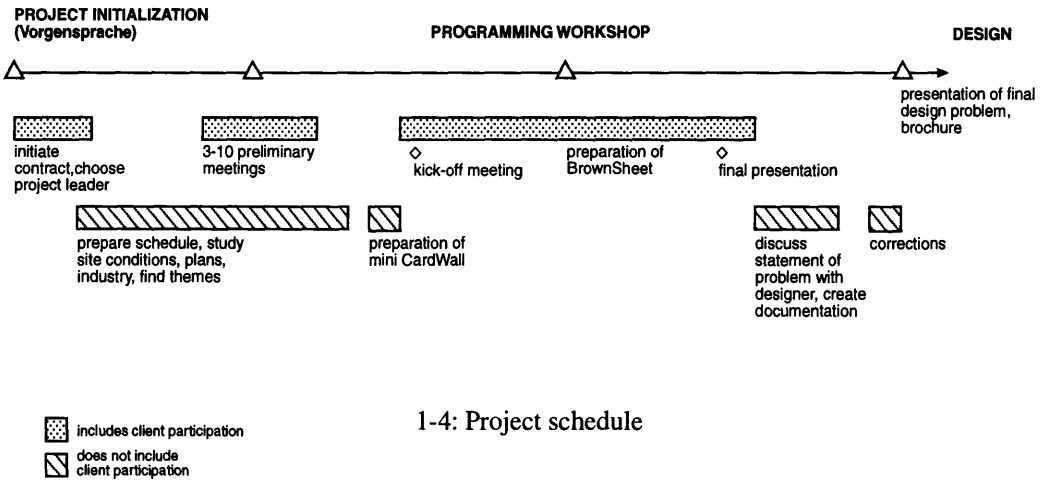
Having the client make decisions about the design changes the role and types of activities of the architecture firm. Henn Architects have moved away from the traditional roles of the designer. The Henn philosophy makes the architect's role more like a management consultant in the initial stages of the design. In later stages of the design, Henn assumes the traditional role of the designer. However, the way projects are handled with programming reduces Henn Architects' autonomy to make decisions. Accepting client's decisions throughout the project means that there is less room for the architect's input. This does not mean that Henn approaches the project without any pre-conceived ideas about how the design will flow. Notions of group involvement, communication, flattening of organizational structures, and shared equity, and making little allowance for private space, influence Henn Architects' designs. These philosophies are presuppositions that are carried into the programming practice and greatly influence final designs.

Although it is disputable that Henn Architects might turn down a contract after doing an

initial analysis of the user's design problem, the methodology of programming and its follow-up design process suggest that the user may not need a new building at all. Perhaps the architect realizes that the real problem with the design is that the management needs to be reorganized, or that there needs to be a different assembly process within a factory. Henn Architects offers User Programming as a service to clients who need to figure out if they need a building at all or are suffering internal problems that stem from a different source.

Much of the value of programming and the Henn Business philosophy arrive less from the process itself, than from the other intangibles that it provides. User programming gives the client a sense of being involved in the project and having the power to make decisions about the final design. It provides representation for the client, who might otherwise have no say about the final product. It gives members of the organization the ability to participate, regardless of their position in the organization, and it provides a solution based on the clients needs rather that the architect's inspiration. The methodology inspires team-spirit. It forces the client to clearly list the goals, needs and problem of the project. Finally, programming prevents the architect from making design decisions before all information is collected from the client.

1.2.2 Project Schedule



1-4: Project schedule

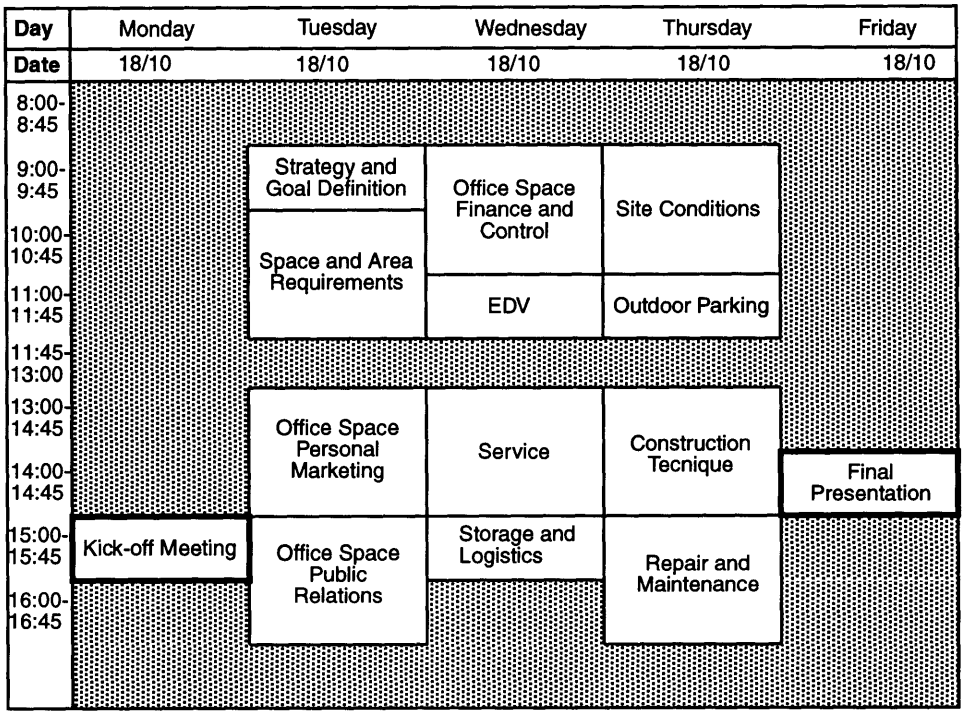


Figure 1-5: Typical Programming schedule (translated from Henn, 1992).

Having initiated a contract for the programming part of the project, then Henn team begins the *Vorgesprache* or research part of programming. This involves a series of informal meetings with different, individual members of the organization. The number of meetings depends upon the size of the project and the level of efficacy the owner wishes to carry out the pre-design. If the project concerns an existing workplace, the Henn team holds these meetings in the workplace of each of the workers to observe the person in their domain. Careful attention must be paid to the processes within the organization and the types of changes that need to be made there. For example, in a project involving a research center, the team would be introduced to the concept of a clean-room, shown how the employees use the room and be described the future goals of the new clean-room (better air quality limits etc). As these meetings are carried out the programmers create abstract categorizations of ideas or *themes*. The themes are the categories of information that are most important to the future design.

A project leader is chosen from the company to work closely with the Henn team. The project leader is typically someone within the client organization who has an interest in the new architectural work and who has an overall knowledge of the people and departments within the organization. This person will act as the closest connection from the

client organization to the Henn programmers. The project leader accompanies the Henn programmers throughout the organization, introducing them to the employees who work there and making them familiar with needs that they may have in the future. These meetings also allow the Henn programmers to study the manner and environment that the members of the organization work in, and to ask them about the features of the place that they like or dislike.

After going through the client organization and becoming familiar with its members and departments, the team prepares an initial analysis of the major themes of the project. These themes represent a classification of the types of information that needs to be gained from the client. The typical project involves a grouping of major concepts that will affect the design. In these listings, there is always a section for the goals and needs. The goals are what the team currently thinks the long-term goals of the organization are at this time (this may change over time). The needs are the specific square meters of area in the building, machinery, equipment and resources that will be required to fulfill those goals. Typically, there are 7 categories that that will be used as themes for the project:

Goals: What are the long-term goals of the organization? What are its reasons for expansion/improvement/building? Does it wish to improve production, employee relations or become a better university?

Needs: The needs are the result of the following design classifications. These are the actual spaces required and their associated costs. This also includes any other quantifiable resources that are needed.

Communication: This is an analysis of the flow of communication within the organization. The Netgraph is used in this analysis to see which members of the organization meet and speak together to share expertise. The communication may also represent a sharing of values or company identity as well.

Workplaces for the People: This facet of information represents the places in which the people will work. Although some of the information obtained in this section may contain information about the area required, eventually that knowledge will be transferred to the Needs section. This theme organizes all information about the way people work within their offices, labs, shops or classrooms. An example of this is the fact that people working in a chemistry lab need to have access to eye-showers.

Special Service Functions: This area of information represents knowledge about social areas and those spaces not directly related to the organization's purpose. Examples of this are cafeterias, cafes, medical facilities, entrance rooms or day-care.

Site Conditions: This body of knowledge is what makes up the site conditions. Information in this section may be traffic conditions, area, topography, soil conditions, people, parking needs, environmental, landscape architecture, pollution.

Infrastructure: The infrastructure pertains to auxiliary utilities and functions that support operation of the building and its inhabitants. These include, telephone, computer networks, electrical, heating, sewage system, fire escape routes and security.

Functions: In this stage, we need to know what specific items are needed for functions within the organization. This area of information is particularly important for production. In regards to assembly production, this may be a listing of all the machines that are needed along the assembly line. This section may also include offices, facilities, storage, logistics and restrooms. Typically these functions are in regard to production for a factory.

Processes: Information regarding processes concerns the connections between functions. Information in this section might show the flow of goods between different stages of a product on a production line, as it passes between machines. This is not simply the functions of the previous paragraph with connections among them, but rather the abstraction, separation, reorganization, selection, and division of these functions. Processes are important to Henn Architects because of their philosophy of "form follows flow", meaning that the building should be designed around the processes involved within it. After making classifications of the information needed from the client for a successful design, the Henn team prepares a schedule for the user programming interviews. The sessions are divided up according to the previous categories or themes of knowledge. Not all employees are chosen to participate in the programming sessions. The project leader and the Henn team choose employees from the company based upon overall knowledge and experience within a certain knowledge theme. Except for the first group interview, there is no requirement of rank or position to be chosen for the programming interviews, but rather broad demonstrated knowledge in the areas. For the first group meeting, top management and strategic members of the organization are asked to participate. This is done because the first meeting is used to clarify the goals of the organization at the beginning of the programming. It is assumed that those in the highest positions of the company are in the best position to have clear goals of the long-term strategy for the organization. It is important that the goals are clearly stated at the beginning so that the rest of the participants in later sessions will see them and be able to make judgements about how to accomplish them. Invitees may be asked to join more than one session.

Sometimes the people chosen for a specific area of knowledge are not able to answer the questions that come up in the process. In these cases, new people may be asked to join the effort. This may mean adding additional programming sessions to the schedule. Other times, the knowledge about a specific area will not be available from within the organization. In this case, specialists will be brought in from outside of the organization to participate in the sessions.

If the goals change or are found to be unclear or unrealistic during the programming, members of the initial meeting will be asked to reconvene.

A week before the programming begins, letters are sent out to the participants inviting them to the workshop. The schedule is reordered based on their availability.

Programming meetings are typically held over a one-week period. The week starts with a 45-minute Monday "kick-off" meeting that introduces the client to the programming process. From Tuesday to Thursday, the group interview/meetings are held. These usually take 2 hours, except for the first, which takes one hour (the top management have less time to meet). On Friday, the final workshop presentation is a 45-minute presentation of the ideas that were found during the group interviews.

The Henn team prepares cards for the kick-off meeting with the goals and facts already known from the preliminary research and obtained from the project leader. These are presented to the members of the organization during the meeting.

The kick-off meeting is held in a large meeting room. Cards from the initial research outlining the organization's goals and facts are presented. The Henn employees also explain the schedule that will be going on later in the week, how programming works, and what the main topics will be. Henn employees also make the clients feel comfortable with speaking during the session and ask invitees to prepare for the meetings by getting them to bring ideas about their knowledge theme.

In the group interview sessions, which are held in the same meeting room as the kick-off meeting, programming cards are conceptualized and created by an architect, as users give suggestions about the goals, facts, concepts and needs of the project. Within each knowledge area, the interviewees make suggestions about what information might be needed to achieve the previously established goals. These are posted on the large wall among others from previous meetings. A moderator keeps the discussion on track by asking appropriate questions (specific questions have been decided upon before which pertain to the theme of the meeting) and clarifying the objectives for the discussion. The structure is somewhat formal, as all members of the group are not allowed to speak up on ideas that they think would help the design. Rather, respondents answer questions from the moderator so that the conversation does not wander from the theme of the workshop session.

Each idea card is drawn in the same way so that the cards have graphical uniformity. However, there is an emphasis on making the cards look different enough so that they are easily recognizable. Typically, the cards are marked with thick magic marker, in bright colors. Bold, clear strokes are used to represent solid, simple ideas. Almost all cards contain two to ten words explaining the content. Cards are left anonymous and there is no other information associated with them once they are placed upon the idea board. The Facts are always orange and the Concepts violet.

As the discussion goes on, cards are used as place markers so that others don't forget limitations that may previously have been brought up. Ideas that were previously brought

up are not forgotten because they can be referred back to from the CardWall. The cards also have the effect of forcing the members of the client organization to provide clear ideas that will be remembered. Underneath the goals, facts, concepts and needs, the cards are also categorized based on "heading cards" which roughly correspond to the major themes, or categories of information, suggested at the beginning of the project. There is typically a heading card for each of the themes of the project, in addition to any others categories that come up. These cards are created during the programming session. Heading cards usually repeat under different super-categories.

At the end of each workshop meeting, the Henn team quickly rearranges the CardWall into the goals, facts, concepts, and needs sections to prepare for the next meeting. If necessary, they will construct charts that focus on a specific scenario brought up in the previous programming session. In the subsequent programming session, members will be asked for expertise supporting or refuting these ideas. Before the final meeting, as many as 10 charts are prepared. These will be presented with the CardWall at that time. The entire CardWall is entirely resorted and corrected for mistakes to prepare for this.

Charts are used to demonstrate specific examples. They are created from a combination of cards that were previously created and arrows, diagrams and text. They are usually compiled and pasted on large 1-m. by 1-m. boards. They are used to show relationships between ideas, connections and possible solutions. An example of a chart might be three scenarios for the site location. It would show each of the locations for the site: inner-city, countryside, or a specific location by a river. For each of the alternatives, the pros and cons of each are displayed on the chart. Another common type of chart is a bubble chart of possible connections between rooms in a building. These charts are based upon the Netgraph communication matrix and show heavier lines connecting rooms that sponsor much communication within them. A final type of chart is the pie chart of cost vs. area, cost vs. material, cost vs. quality etc.

During the final meeting, the entire group checks the CardWall and the week's progress is discussed. Comments are made regarding accuracy and categorization of the cards. The space vs. cost is compared and adjusted if necessary.

After the final presentation, the programmers solidify the design problem by conferring with the designers. A few pages of text are sufficient to represent the problem statement. It is intentionally meant to be succinct. The final problem statement (in addition to all cards) will go into the brochure that will be given to the client for analysis and corrections. All other content generated from the programming also goes into the brochure. The client goes over the brochure and makes corrections if necessary. They may also make remarks or have suggestions about the problem statement, since this is what the design solution will be based upon. The programmers may also have a separate meeting for the client to display the final brochure and explain the completed problem statement. In the meeting the client gives the final word for the design work to proceed from the problem statement.

1.2.3 BrownSheet

The BrownSheet is used as a visual tool to represent the area and cost requirements based upon the needs of the clients. This comes directly from the needs section of the CardWall. It is used to iterate and delegate space among different competing groups within the organization. This stage is done concurrently with the "needs" section of the programming methodology. The BrownSheet is repeatedly edited and changed over the period of the week.

Most of the time the Henn team does not actually use a BrownSheet in the final meeting because it is too difficult to make one from cardboard and paper. As a replacement, two carefully marked spreadsheets are always prepared. The first spreadsheet lists the area requirements in general terms, usually categorized by department area or sub-department. The second spreadsheet is more specific than the first one. It further breaks down the departments into individual rooms, and categorizes them. These categories are typically hallways, social rooms, office rooms, exercise rooms, warehouse space, labs and work-places. The spreadsheet divides the overall area into gross area and net area. There are also three columns that further categorize the overall area. These columns are for human service areas (toilets etc.), technical areas (shafts for pipes, wire-housing etc.), traffic areas (hallways, entryways etc.), and outside spaces (atriums, gardens etc). Instead of using the BrownSheet, a combination of charts and the spreadsheets are displayed to the client in the final meeting.

Catalog of Rooms

No.	Employees			Description of Room			Area/ Main		Area/ Service	Area/ Utility	Area/ Traffic	Area/Outside	Special Material Requ.
	Temp.	Equ. Full Time			Category	Quantity	Usable	Total					
		regular	lab.										

Figure 1-6: Spreadsheet of needs used to created the BrownSheet (translated from Henn, 1991).

The BrownSheet contains square boxes organized into categories of the organization (departments). Groups of similar size and use may be grouped together to show their relations. Each box is drawn in proportion to the amount of area required. A few words describing the use of the room is placed at the bottom of each space or group of spaces. In addition, the areas of the space/s are shown at the bottom of the BrownSheet.

The section of the organization that they belong to categorizes the spaces (department). These groups often coincide with the programming sections from which the needs are derived. The total and net areas of these groups are displayed on the BrownSheet.

The BrownSheet is used to "haggle" with the client over potential areas and how these

areas should be distributed over the period of the programming. A typical BrownSheet then, necessarily contains numerous additions, remarks, and revisions. Since pasting small, white areas representing spaces for the clients to view it quickly becomes covered with tape, glue and ripped off paper squares.

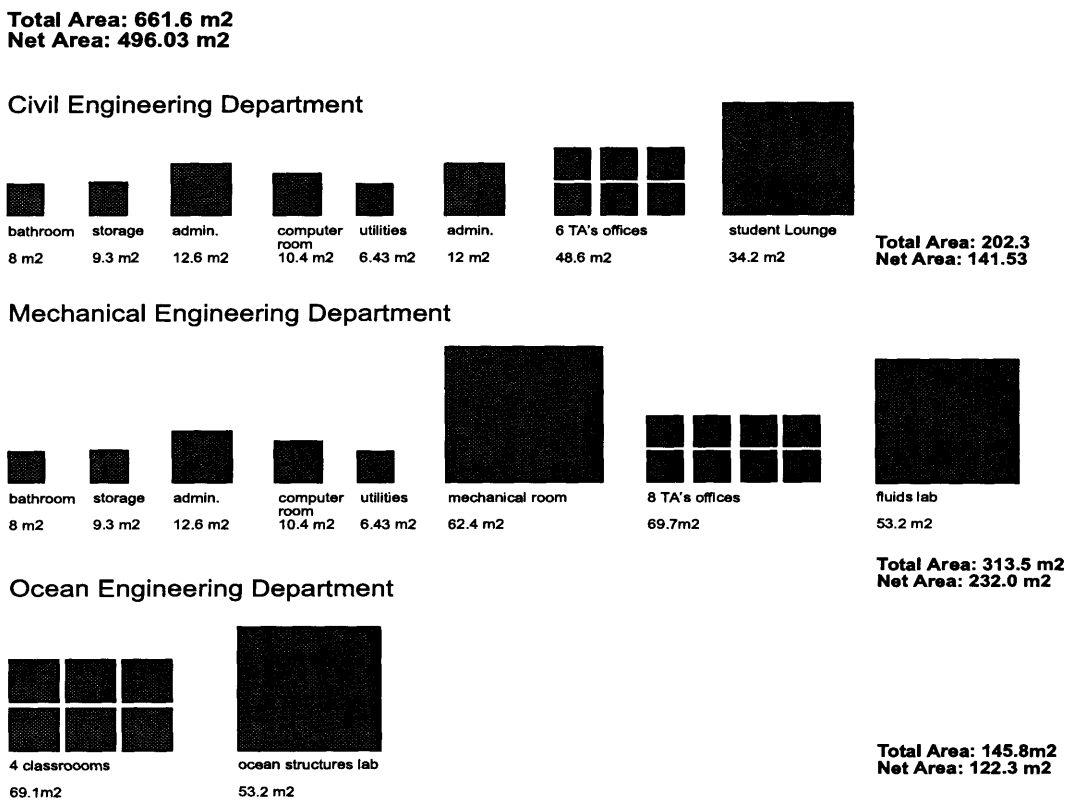


Figure 1-7: The BrownSheet.

1.2.4 Documentation

One by-product of the Henn Architects' programming process is a large book that is a compilation of the ideas formulated in the programming sessions. It is divided by tabs with the categories which make up the programming matrix (ziele:goals, fakten:facts, konzepte:concepts, bedarf:needs, aufgabenstellung: problem-statement, anhang:appendix). A large part of this documentation is the cards that come from the CardWall, copied in color, seven or eight to a page. The employees use color-laser-copying machines to do this. There is no additional text provided with the cards (except for the text on the cards themselves). The content of the remaining sections of the documentation is as follows:

Preliminary Materials:

Cover Page - Project Title, content description, date, Henn Logo.

Intro Page - Project title, description of documentation contents.

Names page - markers for each of the main programming categories.

Einleitung (Introduction) - Intro to the project, scope and how the project was initiated, who owners are, use of the building, where the funding is from.

What is programming? - Contains a description of programming.

Information Page - Has dates about the start of the project, details about the beginning of the project, List of Henn Architects' employees, A list of all the client participants.

ziele:goals

Contains brief descriptions of the function, form, cost and time required in the goals part of the CardWall.

fakten:facts

This contains the cards from the facts section of the CardWall. The programmers organize the cards by the categories of facts that were found during the programming sessions. These categories are not the same categories as the themes that correlate with the workshop sessions. For example, the documentation for the Technical University of Munich arrived at the following categories to sort cards: the identity of the school, the activities which occurred there, the nature of the students, current teaching style etc. Next, there are cards listed in the categories of the individual divisions of the organization. No cards are duplicated in two categories. Cards that are introduced and that pertain to general information about the project are presented. Next, cards, seven per page are displayed under the categories of the grouping that the programming sessions were done. In addition to the cards, organizational charts are drawn showing which connections must be made between different departments. These are drawn as bubble diagrams with connections between the departments drawn in different thickness lines. The heaviest lines have the highest priority. Bubble diagrams are also presented here showing rooms within the department, and the connections between them.

konzepte:concepts

The initial page contains a table of contents describing the concept chapter. Similar to the facts sections, categories are shown describing the categories that have been chosen to organize the ideas. These vary from abstract (i.e. communication) to very practical (an idea about having a social room, suggestions about the structure of the building).

bedarf:needs

Initial Page contains a description of the needs section, followed by a table of contents. There are two main parts to this section, the first is a direct transposition into table form, the needs of individuals as they are expressed upon the brown-sheet. This document does not contain all the rooms that are projected for the project. The second part contains a large and very detailed proposal for the areas which are to be included in the final design. These are organized by type of room (utility, office, social etc).

aufgabenstellung:problem-statement

This section contains a few pages that contain the description of the design statement. For each of the sections of function, form, cost and time, there is a short description of the problem statement.

anhang:appendix

Contains laser copies of all the brown-sheets. Includes a description of the site, including maps, aerial photos, access, vegetation, schedule of the programming sessions.

1.2.5 Business Organizational Diagrams (Netgraph)

The Henn team also uses a technique called the Netgraph to analyze the communication within the organization. (Allen, in press) It uses the information it gathers from the Netgraph surveys and analysis to derive information about how to connect and organize spaces within the organization. The Netgraph allows a designer to analyze the type of communication going on within an organization. The software itself simply organizes the information retrieved from the surveys into matrices representing the communication between different categories of the organization.

The Netgraph is created from extensive surveys that are given to the members of the client organization. These are handed out to the employees who fill them out. Members of the organization are asked who, over a period of the day they conversed with, either by telephone, email or face-to-face. They are also asked who they would *like* to converse with within the organization given better availability. These surveys also provide age, department, sex, salary and other personal information about the respondents.

Participants are asked to complete the surveys a number of times. The data is then analyzed for invalid information based on statistical variation from the norm. Invalid survey data is removed from the data set. The more surveys that are done, the closer the information is to yielding accurate results. The files are consolidated into two single files.

The resulting files are submitted as text files and run through a software program that

creates a colored, two-dimensional matrix representing the connections between the members. The hallmark of this system is that the data can be organized into different categories for comparison. For example, the data could be viewed to show a representation of who communicated in an organization by department. Another way to analyze the data would be to see if age had to do with the communication, or if the distance between the departments was interfering with their ability to communicate.

This information is tested during the programming where the architects ask the participants to describe the type and extent of communication within certain divisions of the company. It is then compared with the Netgraphs for validity.

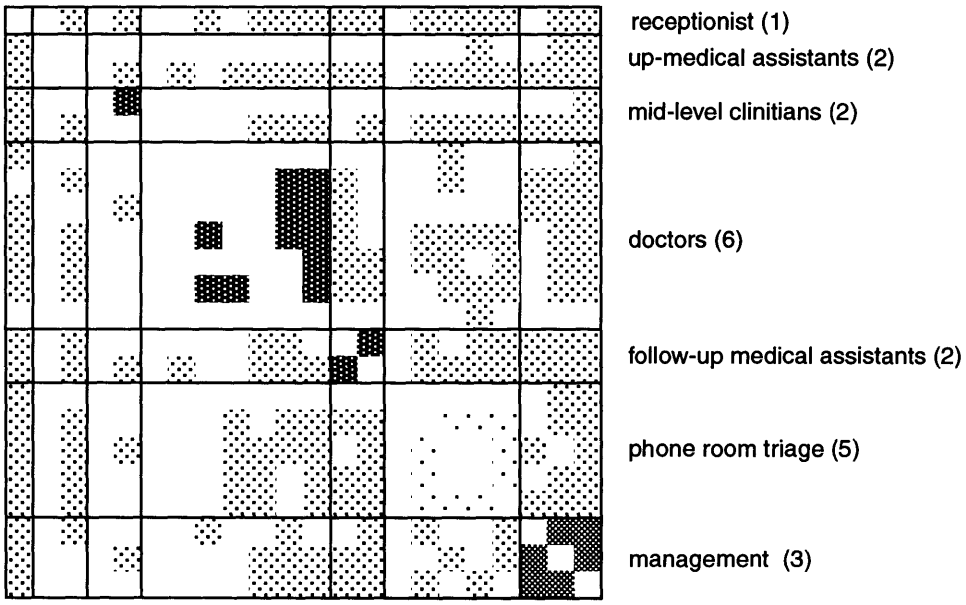


Figure 1-8: Netgraph organized by position, from a small health clinic (Adapted from Kisiel, 1997).

The Netgraphs also play a large part in explaining the communication part of the programming. They are analyzed frequently and used to create cards clarifying the communication within the organization. They are also tested against the CardWall, because programmers ask specific questions to verify the Netgraph data.

1.2.6 Posters

Posters, distinct from Charts are rarely being used because they are difficult to create. They are simply mini-versions of the CardWall. Posters are given to the client to hang on the wall for display or to document the process. The use of posters is to get the company to identify itself with the programming process and the decisions they made during the project.

Chapter 2 - Scope

This chapter follows from the definition and description of the programming practiced by Henn Architekten Ingenieure in the previous chapter. The purpose of this chapter is to describe and analyze suggestions to improve the programming process with electronic media. These suggestions come from both Design Technology and Henn Architects. The chapter starts with a brief historical introduction to the project, and goes into the suggestions for improving the CardWall. Finally, an analysis is made about how and where to apply these technologies to programming.

2.1 Brief History

During the spring of 1996, the Henn Architecture group made contact with members of Design Technology at MIT with an interest in improving and automating the Programming methodology through information technologies. The Henn group envisioned improvements over the traditional methods and technologies that are currently being used. This project has culminated in the development of prototype software and applications development of systems described in this paper.

In the fall of 1996 Paul Keel, Bill Porter and I began work on the project by looking far into the future of the CardWall and the programming process. Rather than simply automating and computerizing the system, the goal was to create a system that would provide additional knowledge to the programming methodology. The CardWall would become more than an idea board of contributor's suggestions, but rather, an intelligent way to classify and organize information. Much like a knowledge-base system, the CardWall would contain intelligent information that could be connected to ideas. In addition, the ideas themselves could be connected together and reorganized to suit particular needs. The group also envisioned the BrownSheet being closely connected to the CardWall, so that the flow of the ideas that exist on the CardWall could easily be connected to the BrownSheet. Furthermore, the BrownSheet could be extended to the preliminary design stage and the MasterPlan stage.

In December of 1996, the Design Technology group presented a series of concepts and a prototype that represented these ideas. Based on this, the Henn group supported the financial proposition for the project to commence. In initial meetings they provided input addressing the important aspects of the project.

2.2 Suggestions for Change

The following paragraphs list the objectives of the Henn group and the Design Technology group for the progression of the project.

2.2.1 Grouping Strategies

Since the matrix structure used during programming is one way of organizing the data found within the CardWall, Henn Architects found it important to focus on other ways to group and classify data. For example, the CardWall could contain different layers that could be used to sort cards. Cards could be placed different on different layers to empha-

size categorization within certain groups. Another suggestion from the programmers is that each card contains a time stamp, and that the cards could be grouped by that category. Similarly this can be done by connecting the ideas (cards), placing different priorities on them, marking them with different colors, or organizing them according to the header card they are associated with.

In addition to analyzing different possible grouping strategies for software prototypes, other commercial software will be analyzed. This includes "brain-storming" software, graphing software, and whiteboard software for group collaboration. The Henn group concluded that the current offerings on the market be scrutinized, in case off-the-shelf software could be used. In addition, different programming methodologies are to be studied within the field of architecture. This would yield a benchmarking comparison with systems of a similar nature.

2.2.2 Creating a Database

To contain all the data contained by the many cards created during Programming (>300 cards for most projects), the Henn group needs to store the data in a consistent, accessible and secure way. Henn Architects also needs to alleviate the logistical problems associated with having to move the large board of cards off site at the beginning of the design stage. They need to prevent the order (hence the location in the matrix) of the cards being forgotten when they are taken off of the wall. Finally, they would like to store much more information for each card than is currently stored. For this, Design Technology suggests storing cards within a commercial database. This would allow the card to be much more than an ordinary graphical representation of an idea with a picture and some text, but possibly a container or link to a rich supply of information.

The Henn programmers would like an archive of cards from other projects that would be searchable based upon different criteria. For example, the cards could be sorted according to the date that they were created or by the theme of the programming session in which they were created. Cards could be sorted according to the project that they were created in. It would not be necessary to sort by the person who created them, since it is important to retain anonymity during programming.

Most commercial databases provide a graphical interface to enter and update data. This could be used to create cards and input card data. It could be used to access the data from a remote site and to archive older projects, storing a large amount of useful information for future projects and historical purposes. For these reasons, the database was considered one of the first things to work on in this project.

2.2.3 Creating Marketing Tools

A large part of the value of the Henn Programming process lies in its intangibles. In order for the Henn group to obtain clients, they must be able to successfully market and promote their products and design philosophies. For this reason, promotional materials are extremely important to Henn Architects. In the past, the group has created well-designed

and appealing pamphlets and materials outlining programming and its virtues. The pamphlets are distributed to Henn Architects' potential clients to introduce them to programming. They would like to expand these capabilities through information technology. For example, Dr. Henn suggests that for an exhibit in a museum of a previous project (Audi-VW) the programming cards be displayed via a computer terminal for others to see how the museum was built. This would lend credence to Programming as well as the design of the building itself.

Another use of promotional materials is in meetings for potential clients. An archive of old CardWalls might be brought up on the computer to demonstrate the basic programming concepts from previous jobs. The ability to show animation and movement would also be appealing for displaying demos to clients.

The Henn group has focused on the WWW as a medium to for distributing promotional materials. The reason for this is the "free" and public nature of the WWW, where large amounts of information can be distributed to large groups of people. In addition, the promotional nature of the WWW itself, being a symbol of high technology and the future is an image Henn Architects is happy to embrace.

2.2.4 Knowledge Base of Information

One of the most powerful concepts of the CardWall is its ability to contain and organize information in a graphical way. Extending this concept of a graphical data system has led the Henn group to suggest the development of a knowledge base of information that can be connected to other sources of information. In this scheme, information added to the system can be monitored for connections to other information that has previously been added to the system. When a new piece of data is entered into the system, a flag will show up signifying some connection to other information that may be related.

For example, suppose a member of a group interview session on Monday noted that there is only a 3-foot clearance for pipe within a certain building duct. This information would be stored in the database. On the following Tuesday, if another member of the team in a different department suggested that the pipes be made large than their current 2' diameter, a flag would raise telling the programmer at the time the idea was suggested that this issue had been raised before.

Intelligent "case-based" or learning systems are the subject of sophisticated artificial intelligence research. Although this project does not intend to cover this field in depth, the Henn group has suggested this area of research in the long-term strategy of the project.

A more readily applicable feature is the creation of direct links between cards and engineering drawings such as floor plans and the HVAC layout. This feature will be closely connected to the task of creating the database to hold the information contained within the CardWall.

2.2.5 Quick Scanning of Cards

Some of the most practical issues can delay the most sophisticated idea. In order for the programming process to be successful in a computerized form, there must be a way for the cards to be scanned in quickly and without effort. The Henn group puts a strong emphasis on the quality of the cards it uses during programming, and has formed an identity for itself with the bold, colorful and uniform look of the cards it produces. The identity of these cards is very much synonymous with the identity of the company and the work that they produce. The artistry involved with the creation of the cards is highly developed and is clearly not reproducible with modern software tools.

Another limitation to using software to draw the cards is trying to display the cards as they are being drawn. During the group interviews, the cards are very rapidly drawn and displayed upon a large poster board as a reminder of the ideas that have been suggested during the session. It would severely slow down programming if the employees had to create and save files. The employees would also have to have some way of placing them on a very large monitor that would have enough resolution to see the details of each one.

The Henn group and Design Technology recognized from the beginning of the project that the "look and feel" of the cards and the CardWall holds high value for the company and that modern computerized drawing tools are inadequate for the extemporaneous type of drawing done during programming. A brief analysis of the available software and hardware available revealed that without better tools for drawing free-hand pictures on a computer, the alternative is to find an easy way to digitize the cards that exist.

The suggestion for scanning cards implies making design decisions in the "problem seeking" stage. Ironically this defies the methodologies of Henn and *Problem Seeking*. The Design Technology group considered scanning to be a less important facet at the early stage of the project.

2.2.6 Representation of Flows within CardWall

Another aspect of the project is finding a way to represent flows of concepts and ideas in a visual way. Inherent in much of the way that programming is done are relationships among the cards on the CardWall. These relationships may be precedence, communication, physical movement, entrances and exits of buildings etc. If the relationships among the data that comes from the client during the group interviews are effectively linked and displayed in intelligent ways, there is much value gained to the display.

One way that relationships among data can be displayed is the use of precedence relationships among activities that are known to be related. These small networks could be displayed as PERT charts within the CardWall. For example, a group of people describing facts concerning the way they work over the period of a day could be linked together in this manner to express the order in which they work. Different types of flows or connections may also represent where the different people worked together to create a common goal. The CardWall contains a much higher value if two ideas may be linked in

this way.

The Netgraph, which was mentioned before, is another way to represent flows in a graphical, flexible way. The Netgraph shows the flow of communication among people within an organization. These results could be formulated as an integrated part of the CardWall. Since the Netgraph exists outside of the rest of the Programming that Henn does, it is one objective of the company to combine the information contained within the Netgraph with the ideas within the CardWall and the BrownSheet.

For example, a simple version of the Netgraph could be part of the CardWall interface, showing connections within the Netgraph in other representational ways. The current version of the Netgraph shows connection between department by a matrix of dots. Another way to represent the data might be to automatically draw a bubble diagram based upon the Netgraph that would loosely demonstrate connections between the departments. It could show the connections between them as very thick lines to demonstrate that there is a high level of communication between them.

2.2.7 Creating Charts

Charts are used to demonstrate ideas to clients after the group interviews have been performed. The charts are typically 1mX1m and are displayed on large boards similar to the CardWall. An electronic CardWall may help to create these charts by using digital cards from the CardWall as parts of the charts.

2.2.8 Creating the BrownSheet

There are a large amount of improvements that can be made to the existing BrownSheet. The Henn group would like to see the BrownSheet have extended capabilities, somewhat like a spreadsheet which could be used to compute areas automatically and sum up the areas with their projected unit costs to reach preliminary cost estimates. In addition, the ability to quickly make changes and store the information easily would be of great benefit. One the most obvious potential improvements is the elimination of the cutting and pasting of the white pieces of paper to resize the spaces.

Currently, the Henn programmers use spreadsheets to calculate the costs of the spaces which they extract from the needs section of the Netgraph. As a tool for clients, it would be helpful to see a graphical demonstration of the trade-off of area vs. cost for different types of spaces (office vs. labs vs. social). This would take the form of a pie or bar chart at the top of the BrownSheet.

2.2.9 Creating Documentation

Much of the documentation that is given to the client is created via a laborious process of pasting the cards from the project into a large book. As mentioned in the description of the documentation from the first chapter, a large part of the part of the programming book that is given to the client is simply all of the cards from the programming sessions pasted into the book, seven to a page, in laser-copied color formats. These are organized by

categories of ideas that the programmers decide are relevant to the design. This clearly consumes an extremely large amount of time for the employees of the company.

Another problem with the documentation is its static form. The text cannot be searched or rearranged easily. If the employees wish to insert a card at the beginning of the document, it is not easy to remove them without replacing the entire document. It is also valuable to arrange the cards into different sorting strategies. For example, to arrange the cards by the date that they were created in addition to session that they were created in.

Finally, an electronic CardWall might be a form of documentation itself. In this form, it could continue its useful life. The clients could expand it by adding to the knowledge-base capability of the CardWall. As the project progresses, project managers could link the cards to shop drawings and specifications. Even after the project, the results of the goals could be linked to the goals of the CardWall.

2.2.10 Creating Graphical Displays

Graphical Displays are an extension and addition to the application of the CardWall and BrownSheet as promotional material. The creation of graphical displays involves the use of interactivity, animation, and user interfaces to achieve a medium for presentations and displays. Particularly in the case of museum exhibits and information sessions, this type of interface is appealing.

This aspect of the project can also serve as an interface to load, save and edit different projects, or to switch between the BrownSheet and the CardWall. This is connected to the knowledge base and the database aspects of the project as well.

2.2.11 Other Features

In later interviews, programmers (Kohlert; Ziriakus, 1997) were asked to outline the areas where information technologies would help them the most. The programmers would like to see hierarchies of information available to them based upon the level of detail of the cards. For example they could link a card stating the goal of "we need more light in the offices" with another card "add more lights" and finally a card labeled "need 100 Watts per desk". This could appear in a hierarchical form, where different levels of information could be displayed outlining different details of information.

Another suggestion was made for different linkages to represent different connections between cards. In this manner, each card could have a series of comments associated with it. The comments would contain keywords that could be searched and categorized. Based on the keywords, cards could automatically be linked together to form groups of connections among them. This would be similar to the system the programmers currently use, of placing small yellow markers on the cards to indicate a higher level of importance. Using this method, they would not have to explain hundreds of cards to clients, but could explain only the important ones.

Links between the cards could also show connections indicating communication on the CardWall. This might mean that cards showing the hierarchy of a business might link together different employee roles. This would be important because a common theme for the Henn Programmers is documenting communication flow within the client organization via the Netgraph.

If cards could be linked to other cards, then it would be possible to show connections between the goals of a project and the concepts that support that goal, and the facts that provide explanation. This would allow different scenarios to be represented in a presentation format that could be shown to clients. Each of these sets of connections could be isolated to show different ways that each goal could be achieved. For example, a user might click on a goal and see those concepts that relate to that goal isolated from the rest of the CardWall. There could also be different types of links, so that heavier links would indicate that the achievement of the connected goal is heavily based upon success of the associated concept. The opposite case, that a goal could never be achieved if a concept is undertaken, could be represented by a dotted line.

Another idea suggested by the programmers is the ability to rearrange the cards without losing the original position. This would allow them to organize the cards chronologically, without losing the previous "header" method of viewing them. This might be useful in between programming sessions to help programmers find cards that are known to be in a particular order. Views of cards could also be customized for the individual, so that each person has their own particular view of the CardWall.

The programmers also recognize that with all of these connections being visible all the time between the cards, it would be impossible to make sense of the links between them. There would be far too many links on the board at one time. This would necessitate sophisticated methods of organization and the ability to hide links when they are not necessary to the understanding of the project.

Cards that connect to other CardWalls is another valuable feature suggested by the Henn programmers. Perhaps it would be possible for a user to click on a card and see another CardWall pop up.

2.3 Analysis

In interviews with the employees who actually do the programming, the areas where programming can be improved or changed becomes less obvious than the above sections suggest. As mentioned before, some of the above improvements and changes imply a solution before the problem is clearly defined. They assume a graphical display of cards upon a computer screen where each card is a bitmap, being manipulated, sorted and stored within a software program. This assumption was based upon an initial prototype developed by the Design Technology group that has these characteristics. This is not necessarily a valid solution to the problem, and should not be considered in an abstract without solid grounding in the realities of programming.

In order to add value and efficiency to the programming process in terms of creating valid software and applications improvements, it is necessary to face the practical limitations of programming. The most important of these is that there will never be an automated replacement for the visually appealing cards created by the programmers. Furthermore, the technological limitations of creating the Cards and then scanning them in as they are being made are a significant technological barrier. This is because of limitations in display size and clarity, lack of ability to effectively draw cards with contemporary hardware interfaces, and the failure to effectively enter unknown data into the computer during the time of their creation.

In order to maintain the procedure and philosophy of the user programming sessions without allowing the technology to drive changes, electronic CardWall use requires the following: A digital monitor 10mX2m with very high resolution and bright color. A pen and pad hardware device, which would allow the drawing of cards in the same, high quality manner as they are on the cardboard cards today. A sophisticated data entry and software interface to accept the cards and load the images into the database along with other relevant information (this would probably require another person to enter data).

Of these three technical requirements, the data entry software and interface is definitely possible. The free-hand drawing tools are also foreseeable as an available product in the next few years, but would have to have different pen-tip thickness and a variety of similar qualities to the magic-markers used today. This would make it a very costly custom piece of hardware. The first requirement, the enormous monitor or projection screen, would be expensive and a logistical nightmare at any time in the future. This is because the sheer size of the monitor would not change in the foreseeable future. The benefits of spending this significant amount of money far outweigh the benefits received from the electronic CardWall itself. This problem will certainly be the largest barrier to the prototype solution with bitmapped images on a display screen, except for promotional or educational uses.

A possible solution would be to draw the cards in the traditional manner, and then scan them into the computer either at the time they are drawn or at the end of the session. It would not be possible to achieve a graphical display of the cards for the interviewees at

that time. However, it would be possible to archive the CardWall as it is being created. This could be used during programming for the client and later on for the designers, promotion, archiving, education, or presentations covering different alternatives. This would require an advanced form of the prototype mentioned before, as well as a rapid scanner and software to interface with scanner and CardWall prototype.

Yet a third solution would be to wait until the programming is finished to scan in a subset of the cards from the CardWall. This may be passed to the designers, but would probably rather be used for promotional materials and presentations only. Of course, once the programming is over, the record of which cards came from which interview are lost, as well as other details concerning facts about the card. Although this information is not gathered anyway as programming is currently practiced, it would be a significant improvement to gather this information and have it stored into the computer. This would increase the value of the cards on the CardWall and extend their useful life.

Another strategy would be to continue to develop small versions of the CardWall such as the prototype mentioned before, and to wait until significant technological difficulties are overcome. In the mean time, other information technologies could be applied to other areas of the project, which are more readily applicable.

For all of the above solutions, it is true that the longer it takes to scan in and enter associated data, the more difficult it becomes to keep track of the information associated with each card. One of the problems with the way that programming is currently practiced, is that the cards are dead after the programming sessions are held. The cards currently hold such limited information that it is hard to make sense of them without being present when they were created. In addition to the cards' text and drawings, a few additional phrases or contextual information can help to make them more self-explanatory. The way programming is practiced today leaves too much information forgotten. The best time for the cards and additional data to be entered is at the time they are created. If enough additional information could be associated with each card, then the finished CardWall could be passed to a different designer with almost no explanation necessary. It would also be useful in the creation of the brochure to document additional information for each of the cards.

There is another thing to take into account when choosing a basic design solution. It is that the process of programming should never have to be sacrificed for the perceived benefit of technologically sophisticated solutions. Although the goal of this project is to streamline and improve the process, technology should only add benefits and not take away from the basic premises of programming. Since many of these benefits also come from the non-tangible elements of programming, certain efforts must be made not to over-automate or interfere negatively with the system.

This does not mean that the process will not be changed. In fact, technological improvements will bring about changes. As one programmer suggested, that if there could be a

greater connection made between the programming and the Netgraph, there might be sufficient thrust for an additional meeting for the client to show and explain the results. This could be a results based meeting, showing the technical efficiencies of this.

The Design Technology group agreed that the solution should be to leave open the possibility that the cards could be displayed at the time they are created within the programming sessions. More realistically, and in the near future of the project, it would be better to assume that the cards will be scanned in during of after the programming sessions. This would create the documentation and archiving as the sessions go on. In between sessions, others (such as the client) would be able to add cards to the CardWall remotely and view the results of the sessions so far (this is actually done now, but the programmers must announce the new additions at the beginning of the next session). The designers would be able to sort out the cards that they think are important for their work, creating a summarized version of the larger CardWall. The CardWall would be available between programming sessions, for the rest of the company to view the process. An electronic CardWall could more easily manipulated, searched and sorted at this time to enhance the marketable value of the programming sessions. Managers could use the CardWall between sessions to evaluate design possibilities, show connections and weigh out alternatives.

Since the information about the cards tends to be lost as the time between when the card was created and the time it is recorded, it is better to input the cards as they are being made, rather than at the end of each workshop session. However, due to constraints with scanning and inputting data, it may be necessary to input the data and scan cards at the end of the programming sessions. Although design solutions should take into account limitations in drawing and inputting cards, the solution should leave open the consideration that someday the technology will change.

Using this strategy, the CardWall would act much like a communication device. The interface must be distributed and multi-user. It should be available for access to anyone with the appropriate permission.

Based on interviews with programmers, computer technology would only be helpful in certain areas of programming. This implies that information technology solutions should be applied carefully and only in places where it is needed. The following diagram shows the area where solutions will be applied.

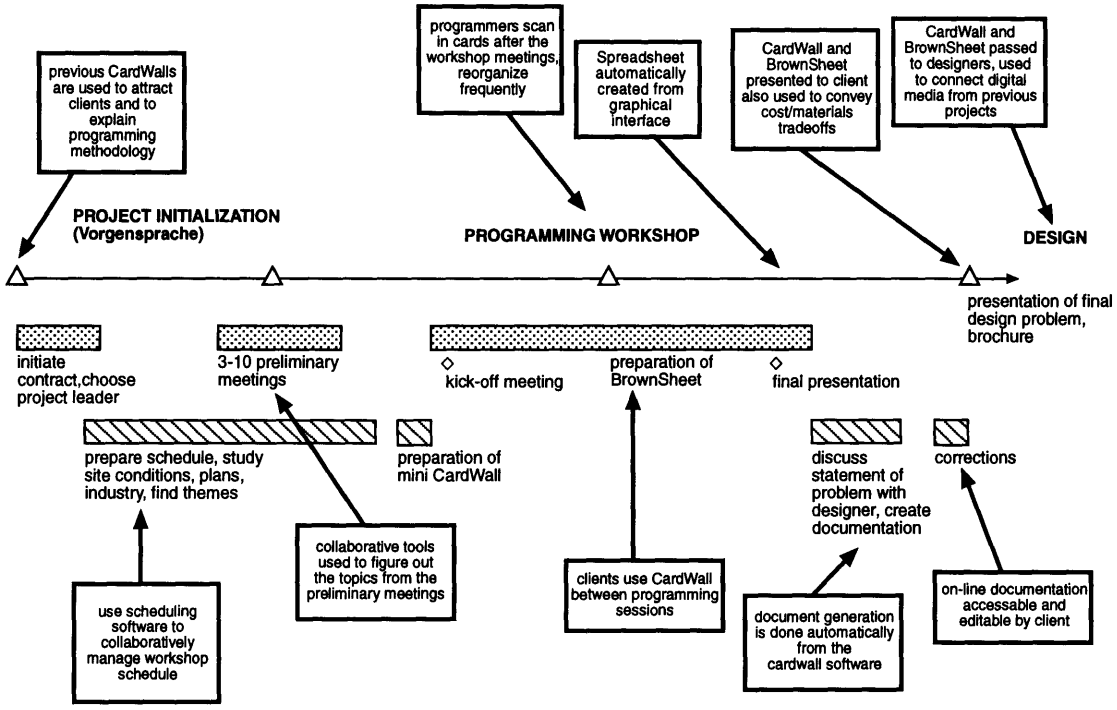


Figure 2-1: Areas within the programming schedule slated for improvements.

Chapter 3 - Design Concepts

This chapter uses the suggestions from the previous chapter to propose software design elements for a programming software application. Although the intention of these improvements is not to change the process used by the Henn programmers, change will be inevitable result of these improvements. Many of the topics presented in this chapter represent ideas generated by Paul Keel and myself during the fall of 1996. Most of the ideas presented in this chapter are independent from implementation. Not all have been implemented, or have been completely analyzed in detail. Some of the ideas imply directions that the software will lead, as the solution becomes more complex. Architecture and technical information are presented in the following chapters.

3.1 Combining the CardWall and BrownSheet

The initial stages of analysis led to the problem of combining the many parts of programming into a unified application. If possible, the CardWall module should be connected with the BrownSheet and Netgraph. In the future, the application may be used in areas of the design as well. Figure 3-1 shows a possible configuration to make the connection between the pre-design and design parts of the project. The diagram labels the pre-design stage as the **concept elaboration** stage, and the design steps as the **concept transformation** and **formation** stages.

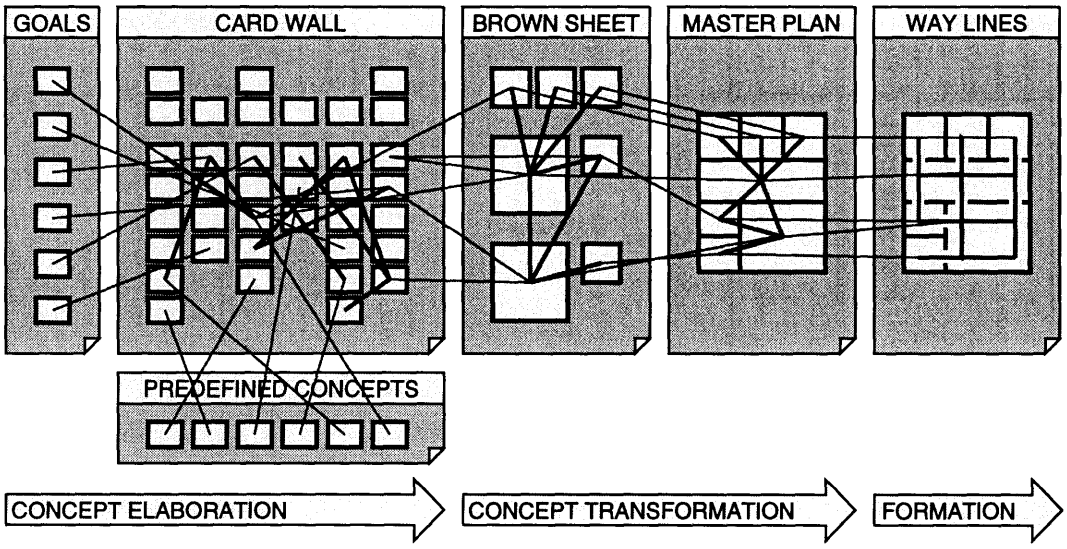


Figure 3-1: Connection between CardWall and the BrownSheet and design (drawn by Paul Keel).

In Figure 3-1, a separation is made between the goals of the project and the rest of the programming. This is to accentuate the clarity of the goals at the beginning of the programming as defined by the management of the organization. In the actual application, this may be a separate frame to contain the cards that are in the goals section of the matrix. An alternative to using a frame, is to mark the cards and order them so that the goals would be separate from the rest of the CardWall. Another category of cards is the **predefined concepts**, which are the ideas that the Henn programmers implicitly bring into the project through their experience and knowledge of successful design. The predefined concepts should not be shared with the client, since the client does not create these ideas and they will be the same for all clients.

The application should be able to combine the CardWall with the BrownSheet and show the connections from the needs section of the CardWall directly to the BrownSheet. Suppose each of the square boxes on the BrownSheet is resizable, moveable and can be linked together. For each card in the needs section (meaning that it specifies the area requirements), a certain square is created. This box goes onto the BrownSheet canvas. In the master plan area, the squares from the BrownSheet could be replicated. There, they could be dragged and moved to create different dimensions (while keeping the area constant) that would better approximate the constraints imposed by the site. The different rectangular and square areas could be arranged in a suitable way to optimize space within the limitations set by the site conditions. These rooms could also be connected with lines to indicate connections between them. The lines indicate the locations of doors connecting the rooms. Taking this a step further, the hallways, windows and doors could be optimized for light and livability using a method such as WayLines developed by Paul Keel (Keel, 1995). The result leads an architect well towards the final design of the building.

The combination of these different elements sets a starting point for the project. The application will consist of a canvas that allows the manipulation of miniature "cards". The cards will have certain properties associated with them that may be modified by the user, depending upon access restrictions that the person has. The cards will be able to be categorized and organized in a number of different ways. This will reduce the complexity of a large number of cards, and allow users to focus on a small subset of cards that pertain to a particular topic. The CardWall section of the application will also have a link to the BrownSheet, where information coming from the CardWall can be directly linked into the BrownSheet. This will translate into resizable boxes, which represent the area requirements. The BrownSheet section will be able to display statistics about the percent use of space, total area, net area, and cost requirements, as well as categorizing the areas on the BrownSheet.

3.2 CardWall as a Distributed Medium

Henn programmers suggest that different people access the CardWall during the promotional sessions, during workshops, and after the workshops (for the designers). This

indicates that the system should be developed so anyone can access the CardWall from anywhere a computer terminal is available. It also means that different people will simultaneously access the CardWall, each with different interests and permissions to change the information contained within it.

A typical CardWall involves 300 or more cards. Such a large layout makes it difficult to make sense of so many ideas. For clients and designers who access the database, a large number of those cards will be irrelevant to their particular needs. Some users may wish to display only a subset of those cards because it takes too long to download 300 cards. This means that the solution should focus on allowing the user to customize the number of cards that are downloaded and displayed. One way to do this would be to provide a card arrangement file for each user that specifies the cards that will be downloaded and the way that they will be displayed. Figure 3-2 demonstrates this. Each user has a specific way that they may view the information, while all information comes from a semi-public database of cards. The CardWall is centralized, but may be accessed by a large many users.

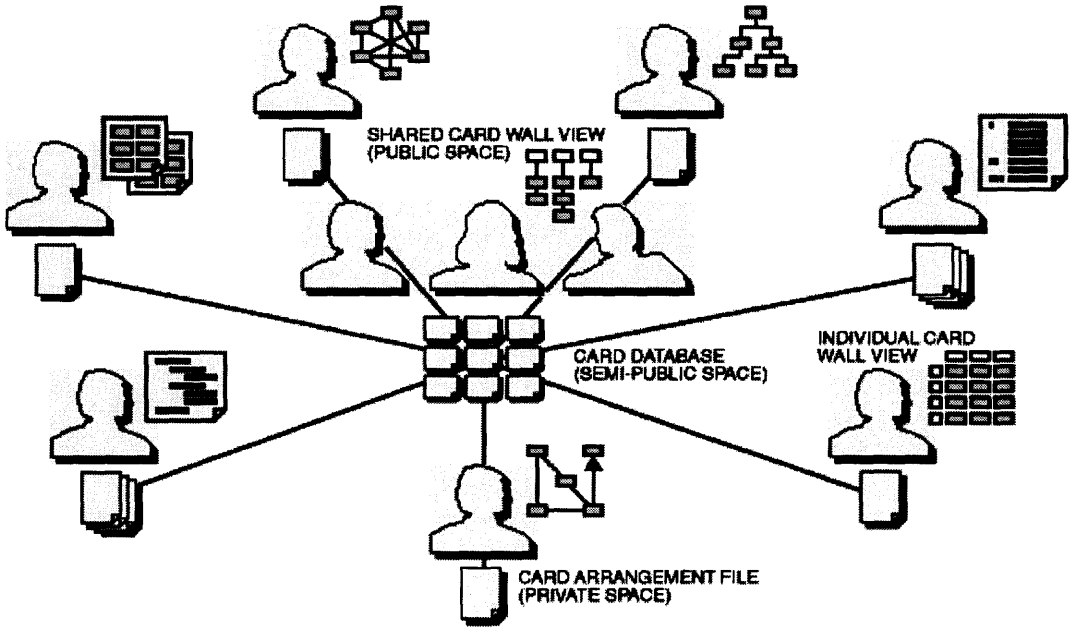


Figure 3-2: Within the CardWall module, each user has a preference file to determine how cards are arranged and displayed. Cards are downloaded from a centralized database (drawn by Paul Keel).

3.3 Link Types

One of the main features that the programmers mentioned is the ability to connect cards together via links. For this reason, the application should have different ways to connect cards and to make relationships between them. Therefore, the application should have the ability to make different types of links between the cards. Five types of links have are proposed. The types of links are the hierarchy, the network, the process, the group, and the progress links. The hierarchy links show direction from one card to another and is used to show a tree relationship between cards. The network link shows only a connection relationship between cards without implying any other organizational groupings. The process link, shows a dependence based on order. The group link combines cards within a single level of hierarchy.

Since there are a variety of different ways to display the cards, we might save each different card arrangement as a special view and save the name and location of each card in a list. In this case, there could be a CardWall view, a "needs" view or a hierarchical view. Users could switch between the different views or save new ones, as they are needed.

A different way to arrange the cards might be by using the links rather than saving the previous locations of the cards. The links and not the actual locations of the cards on the screen determine the way that the cards are displayed. That means that users may link cards together using hierarchical links and later press a button that would display the cards linked as a hierarchy. If a few cards are linked using process links, then another button would show an outline of cards connected to display a PERT-like chart.

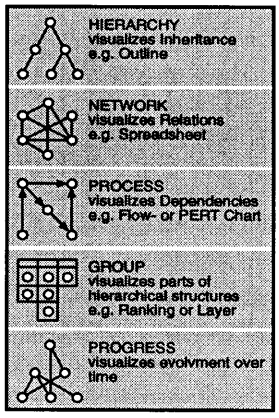


Figure 3-3: Cards on the CardWall contains 5 different types of links (drawn by Paul Keel).

3.4 Saving Cards of the CardWall

Saving different versions of the CardWall and the BrownSheet is another issue the Design Technology group tackled at this conceptual design stage. This issue comes up because one intention of the programmers is to be able to look at the CardWall at different stages of its progression. This requires some way to go back to previous data stored at periodic times during the programming. The most obvious way to do this is to save all the information about the CardWall, including the locations of the cards, and the links connecting the cards. One problem with having multiple back-ups is that it requires redundant copies of the information. These copies often get confused when the user wishes to see a previous version.

A different way to save previous versions of the CardWall is to keep only one version of the CardWall at all times. Rather than saving the entire CardWall for every backup that is made, all edits may be saved. Since it is true that the number of deletions from the CardWall is minimal (unless an unwanted card is accidentally made), this may be a valid solution. Similar to the wastebasket familiar to Apple Macintosh users, each card that is deleted can be placed in a separate category. To resurrect cards that have been placed in the trash, it may be possible to drag them out of that area and place them on the regular CardWall screen. This ensures that nothing is ever actually deleted over the period of the project.

Another way to delete cards and links without having to save a backup of all the information of the CardWall is to make it impossible to delete cards and links, but only possible to mark them as useless or invalid. If cards are not "used" by being linked for a period of time they are automatically grayed out. Fading out the card to show that it is less useful without deleting it prevents the user from having many different backups of data. This helps to resolve conflicts between different users, who may disagree about which cards belong on the CardWall.

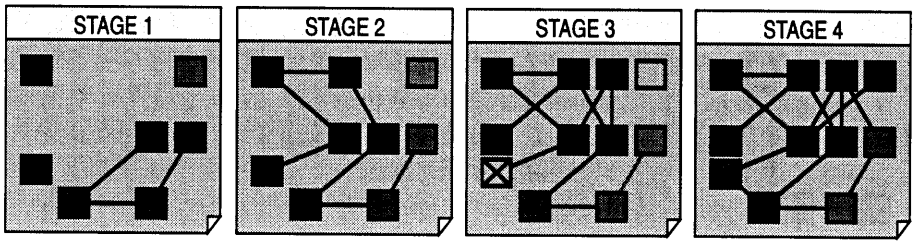


Figure 3-4: Saving the CardWall information is done by reducing the priority of cards and links (drawn by Paul Keel).

By combining the concept of cards having priority with the idea of graying out cards, it is easy to see how the cards may be deleted by lowering their priorities. If person double-clicks a card to reveal an information dialog box about the properties of the card, they may choose to lower the priority of the card by clicking a button. The newest priority will show up on the dialog box. At the same time, the card will become slightly more grayed out. The person may choose to reduce the priority of the card, but does not have the ability to remove the card completely.

During programming, it is necessary for different people to have different roles. This means that there should be different permissions for different people. The programmers would like to be able to delete cards, while the client should only have the ability to add cards or lower the card's priority. One way to implement security without developing a proprietary password encryption and authorization system is to rely on a previously developed system from a standard database.

3.5 Arrangements

It is important for this application to arrange information in a variety of different ways. At this conceptual stage, all of the cards on the CardWall may be arranged by pressing a button which uses the support of the different types of links to display certain subsets of the cards. The arrangements are the different algorithms that represent data to the user.

The CardWall arrangement is the typical Pena style of organization. This uses the information stored in hierarchical links to display the cards on top of one another. The algorithm is implemented by searching for all cards which have no links attached as parent cards (no incoming links, only outgoing links). Since the Henn matrix structure has a consistent layout, (goals, facts, needs, concepts and problem statement at the top level, subcategories of each of these at the second level, and at the third level no subcategories), it is easy for the algorithm to predict the layout. The algorithm recursively goes through the tree structure, counting up the greatest number of leaves of the tree that are parallel and at the second level. Based on this number, the top level of cards can be drawn out. Cards that are linked to different second level (or heading cards) may be replicated in two different places. A different colored border indicates any cards that have been replicated. This layout implies that all cards are correctly linked. If they are not (e.g. no cards are without parents), it becomes significantly harder to draw this arrangement. It is possible to automatically draw the links according to the way a user has set up the cards on a grid. Another way to guarantee the correctness of the links is to have the computer check each time the user creates a hierarchical link. If there is an error, a dialog box can pop up explaining the problem.

The hierarchical arrangement uses the network links to organize the CardWall into a hierarchy tree. The cards that do not belong to the tree are not displayed on the canvas. This is useful for finding a hidden hierarchy of cards. In order for the algorithm to accomplish this, it must have a top card defined. This is the last card that has been chosen by the user. The algorithm starts by recursively drawing out the branches of the tree in a

counter-clockwise or clockwise fashion. As it draws each card, it leaves a pointer to that card in a separate array. The algorithm checks the array before drawing each card, whether the card has already been drawn. This resolves circular links among cards during the algorithm. The final layout depends upon whether the rotation of the links goes counter-clockwise or clockwise. It is displayed with a tree structure as the final result.

The progress arrangement is used to show the order that the cards have been created. This is a feature that the Henn programmers mentioned in interviews as a desirable for use between programming sessions. The programmers want to see the order that the cards were created without losing the original organization of the cards. The progress arrangement combines the order that the cards were drawn with the way that they were connected. The cards are spaced out along a timeline in the y-direction. Cards that result as direct connections from the original idea are drawn along the x-direction. Cards that are a result of other cards are those which are connected via network links and have the next latest date. The cards must be scanned in as they are created, or in the proper order that they were created at the end of the programming sessions if this algorithm is to be implemented correctly.

There are a number of other arrangements that are explained in detail in **Process and Relational Analysis, Keel.** These are:

Outline - An arrangement for viewing hierarchies similar to the CardWall arrangement.
Center - Organizes cards in a hierarchy with the top card in the center.
Groups - Uses group links to separate groups among the cards and to isolate them.
Priority - Shows cards with a weaker priority grayed into the background.
Highway - Uses a combination of link priority and card priorities to show connections between cards as heavier or lighter.
Matrix – A view similar to the CardWall, but with categorization in the y-axis.
Layer - Divides cards into the different layers they have been assigned to via a separate window.
Circle - Draws all cards in a circle fitting the size of the screen so that all the links are visible (no two cards are adjacent which are linked).
Process - Isolates all processes within the CardWall and displays them.

3.6 Netgraph

The Henn programmers suggested that the Netgraph program (AGNI) be moved from the Unix and OS/2 platforms it now resides on, to a more distributed and common platform. In addition, they would like to see it become a part of the CardWall and BrownSheet application. In more advanced stages of this application, a module will be provided support the Netgraph. Although all of the analytical features provided by the AGNI software (Allen, in press) are not necessary to access from this distributed platform, a thin client version will be available as part of the package. The additional capabilities that the Netgraph will provide are alternative ways to view information provided by AGNI files. Figure 3-5 gives an example of this. In this figure, three teams have been

surveyed and analyzed for communication patterns. The fist part shows the typical Netgraph layout as it looks with the current AGNI program. The second stage shows each of the teams broken up with connections between them represented by lines of different thickness. The teams that have communicated the most are connected with heavy lines. Another way to view the Netgraph information is to break up one of the teams and look at the communication among the teams. Similar to the CardWall, the Netgraph will have different arrangements that provide algorithms to display information differently. The Netgraph will have three arrangements that can be used to display the information with the push of a button.

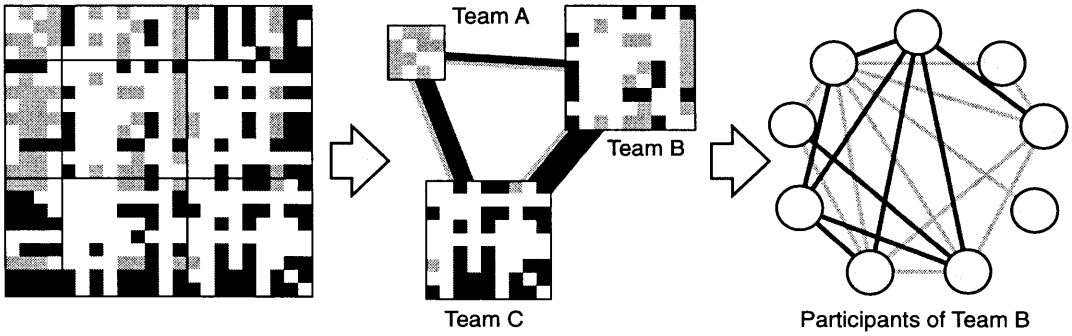


Figure 3-5: Lightweight version of the Netgraph animates data with three different types of representations (drawn by Paul Keel).

Because the CardWall, BrownSheet, and Netgraph modules are closely linked together, it will be easy to compare information contained within the CardWall to information within the Netgraph. The Netgraph will also make suggestions about how spaces in the building should be connected, since the "teams" may be correspond to departments within the organization. The connections automatically provided by the Netgraph will be used to make suggestions at the master plan stage. The master plan stage is where rooms are connected together and rearranged to reflect site constraints.

3.7 Card and Link Attributes

The computer medium provides support for specifying attributes for both cards and links. This is not something that is effectively supported with the physical CardWall in use today. Attributing information to these objects on the CardWall provides greater richness and value than having only images attributed to cards. Figure 3-6 shows the different attributes that can be connected to cards and links. The first column, support, indicates whether the computer will provide values for the specific attribute. If support is partial, the user is required to enter values. If support is automatic, the computer automatically computes values for this at the time of creation. Time and Date are automatic because

they may be computed when the card is created. Since cards will be scanned in at the end of the workshop, the card's times are not the actual times they were created, and so the date and time fields should be editable. Some of the fields in Figure 3-6 are not required for the actual application. For example, the Henn programmers disagreed with using a name attribute because all of the cards should remain anonymous. It is important to realize that during and after programming sessions, it is difficult to re-create the information that went with every card, therefore some of the information on the table may not be entered or used in the actual application. A further description of these attributes is provided in Process and Relational Analysis (Keel, 1997).

Figure 3-6: Attributes associated with cards. This data will be used to design the database of cards (drawn by Paul Keel).

3.8 BrownSheet Module

Henn programmers would like to be able generate a BrownSheet from the spreadsheets that they compile from the required areas. Rather than taking this approach, it may be better to start with the graphical BrownSheet and create a spreadsheet from that. The BrownSheet module of this application will circumvent the use of the spreadsheet completely and allow all required areas to be created visually. The user can create new area requirements on the electronic BrownSheet from a menu. The creation of a new BrownSheet area will place a representative white square on the canvas. The user can drag the corner of this box to modify the size of the area, or can specify an explicit value for the area. The user can categorize the BrownSheet boxes into usage of space (office, exercise, warehouse, labs, social etc) and departments, as these boxes are created. A dialog box indicates the following properties: explicit area, usage of space, department, percentage of net area to gross area, description, number of employees temporary and permanent, and comments on material requirements. When the user saves the BrownSheet, the information is filed into a remote database or spreadsheet.

The BrownSheet Module will be intimately connected to the projected cost of the project. The BrownSheet will allow the categorization of space in two different ways. The first is by department, and the second is by type of space. For each of these two categories, there should be two sets of charts giving statistics about the spaces. The first, is the percent of total area by department and percentage of total cost by department. Second, is the percent of total area by category and percentage of total cost by category. Next, there should be a table of costs for each of the different types of spaces. The BrownSheet also requires configurable ways to categorize the different types of spaces.

It is important that the dialog box that displays the properties of each BrownSheet area be configurable for the user. For example the usage of the space (office, exercise, warehouse, labs, social etc) may change for certain projects. Having the user enter the category manually is a way to configure this. After the category has been entered, it will appear as an item on a keyword list of categories. At the end of a session, the settings will have to be saved to an initialization file for subsequent use.

The BrownSheet has three different views. The first view arranges the areas categorized by type of space. The second view shows the different areas categorized by department. For each category, the net and the gross areas are displayed, as well as the costs associated with the category. The third view saves the last arrangement as the user left it. This view would be used to recall an arrangement of spaces into a rudimentary master plan, showing the connections among rooms and their relative positions.

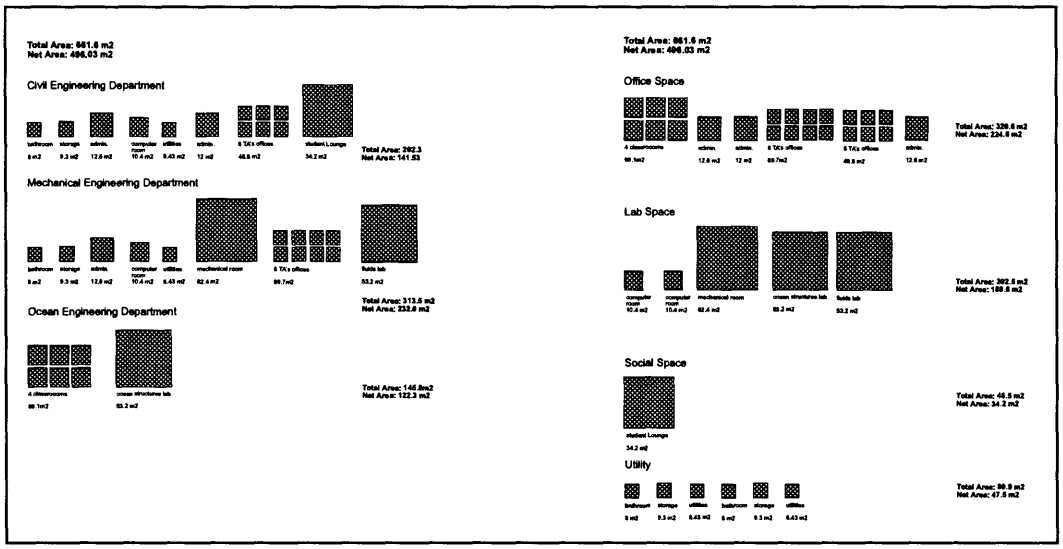


Figure 3-7: Two of the proposed BrownSheet arrangements: by department, by type of space.

Chapter 4 - Architecture

This chapter focuses on the architecture of the information systems that have been applied to the programming process. Because this thesis describes the beginning of a larger project, much of the prototyping and development is unfinished at this time. Of the different aspects of project, the HennApplet and the database of cards will be covered the most thoroughly, since it has been the focus of the development during this thesis research. The applet provides a graphical user interface from which to view the cards, select the cards from other projects, manipulate the information contained within the cards, and create the BrownSheet. The database provides features to store, organize and sort information.

This chapter follows the principles outlined in *Object Oriented Systems Analysis* which recommends the following steps for software development: Analysis of the Problem, External Specification, System Design, Implementation and Integration.

4.1 Distributed Medium

Because of the requirements given by the Henn programmers, it is important that the CardWall be highly distributed and platform independent. For this reason, the CardWall and BrownSheet applet (HennApplet) is written in Sun's Java®. This provides the ability of the application to be accessible from any location that has a network connection and a web browser. Although the application is distributed, the data must be centralized. This is because there can not be multiple copies of the card information for every computer which uses the application. It would be far too complicated to keep multiple copies in sync with one another.

The disadvantages of using Java for the HennApplet are lengthy download times, security restrictions, and speed. In the near future, each of these limitations seems to be surmountable. For example, using JAR file formats to zip and compress multiple class files makes download times shorter by reducing the number of connections to the host to one. The future also promises greater bandwidth to decrease download times.

Security restrictions prevent unknown applets from planting viruses or reading private information on the host machine. The SecurityManager class provided in the java.lang package manages these restrictions. Unfortunately, they prevent the applet from writing file (i.e. arrangement file) to the host machine. The security restrictions on the applet may be lifted if the origin of the applet can be correctly verified. Using public and private keys to sign an applet with a digital signature can verify the applets' origins. This feature is already available on Microsoft Internet Explorer and is found in the package java.security package in the jdk1.1 release.

As a compiled language, java code is as much as 20 times slower than native C++ code. Since java's introduction about 2 years ago, this has been a significant problem for java developers. The JIT (just in time) compiler provided by SGI in its Cosmo-Code 2.0 has overcome this (Shiffman, 1996). The compiler first type checks the code for security and

translates it into native code. This provides significant performance gains from interpreted java bytecodes. Because of this technology, as well as the digital signatures and compressed zip files mentioned before, java remains a good choice for development in the future.

4.2 Event Model

Shlaer and Mellor suggest the use of a state model to determine the different states that the cards may take over the period of their lifetime. Figure 4-1 demonstrates this model. This particular model is intended for use with a distributed architecture because updates to the database are made at the end of the session, rather than while attributes of the cards are modified. If updates to the database are made as cards are moved and manipulated on the screen, it would slow down the applet considerably.

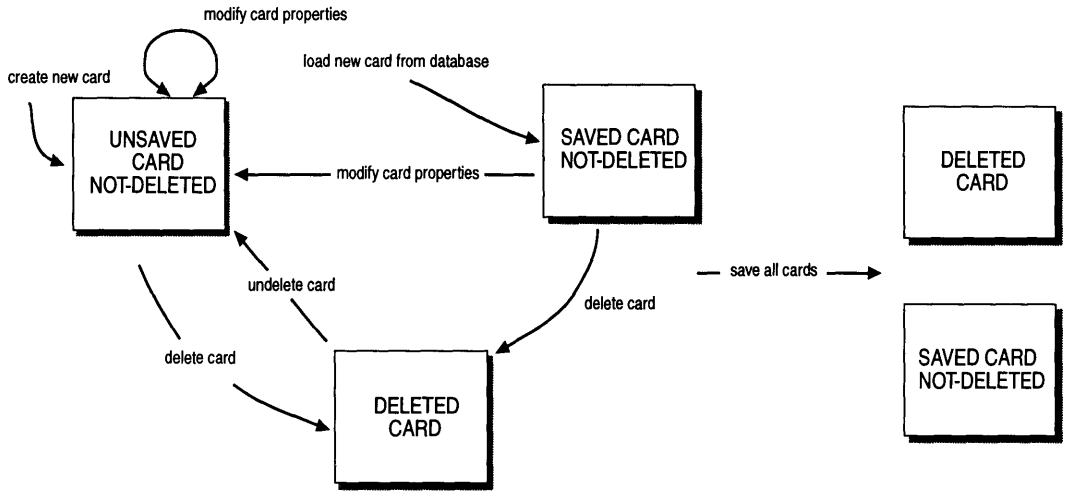


Figure 4-1: State diagram for the HennApplet. This provides periodic updates to the database.

4.3 Schemes for Saving the Information in the CardWall and BrownSheet

There have been two different schemes that have been used to store cards and BrownSheet areas into a centralized location. The first scheme uses a simple Java server that accepts calls from the HennApplet and saves the information into a certain file format (this format is described in the following section). This scheme is outlined in Figure 4-2 below.

The second scheme involves a database and becomes significantly more complicated. Instead of making a connection to a server that writes a stream of data to a file, the applet makes HTTP post and requests to the Lotus Domino® server. Images from the "Images" database are also downloaded directly from the applet. When the information about the

cards is read in, the applet requests the database for a specific view of the data. The view can be set up to isolate certain cards (for one project for example) so that not every card within the database is read. Using the views for input data provides rapid retrieval of information, but is dependent upon the way the data comes from the HTML data the server sends back to the client. When the HennApplet saves data, it does so using HTTP 1.0 'POST' methods to the server. This requires a single connection to the server for each card that has been modified and is quite time consuming. Figure 4-2 below demonstrates

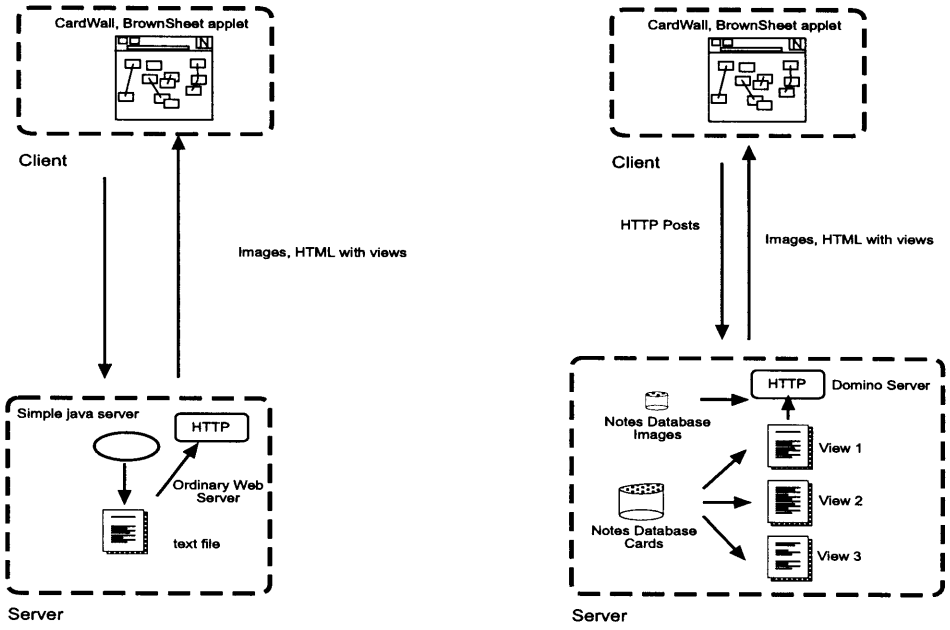


Figure 4-2: 2 ways that data may be saved from the HennApplet.

this.A
A third model for saving the data to a central database involves the Notes database, but this time using RMI (remote method invocation) to save and read in the cards. This is described in the following chapter.

The information contained within the BrownSheet is much less document-centered than the CardWall data and is best stored directly to a spreadsheet. Although this has not been completed yet, early versions of the applet will have all BrownSheet areas saved (like the cards) to a Notes database and then transferred to a spreadsheet with a Lotus Script® program.

4.4 Object Model

At an early stage in the design of the HennApplet, it was decided that the behavior of the

CardWall and the BrownSheet be similar. Both have the following attributes:

A canvas to contain a number of rectangular, graphical objects

- The objects (BrownSheet box or CardWall card) within the canvas may be moved and resized
- Each object on the canvas may have it's attributes reflected and updated by a properties menu
- Contains a grid with snap, resize grid etc.
- The objects on the canvas may be linked together
- Layer support
- The state of the either the BrownSheet or the CardWall may be saved at any time to a central, remote data source
- Support for animation, includes a double-buffered display canvas

Because of these similarities, both the CardWall and the BrownSheet derive from a common ancestor, the AbstractCanvas. Many of the functions within this class are abstract, or are over-ridden by the derived classes, BrownSheetCanvas and CardWallCanvas. Figure 4-3 shows this relationship. The class structure of the cards and BrownSheet areas are discussed in the following chapter.

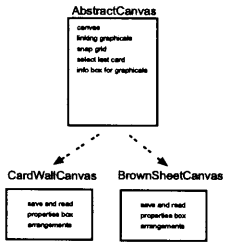


Figure 4-3: Both the CardWall and the BrownSheet exhibit similar behaviors, these are abstracted from the AbstractCanvas class.

4-5 Database Design

Although the database provides storage for cards that the HennApplet accesses, it also provides additional features for other information technology solutions. The database provides security and password authentication. Currently, this is done over the web with a basic authentication scheme (64-bit encoding) which does not present a high level of security. The database also provides support for documentation creation and workflow. This means that by applying certain scripts to the database of cards, it is possible to arrange them into the format of the documentation. This can be done to automatically produce the documents that will be given to the client. Finally, the database allows the cards and images within the database to be searched and sorted.

Because of the features listed in the previous chapter, a Lotus Notes database was chosen. This provides the advantages of a form-based database that can be used to create forms for cards and brown sheet boxes. The Domino web server allows the applet to communi-

cate with the database and to submit and request information.

The cards are stored in a database form called Card, which has the properties referenced in Figure 3-6 of the previous chapter. There is also a links field that is a multiple-choice keyword box that contains the keywords referring to the rest of the cards. This allows the user to select the other cards that may be linked to the card. The database provides a number of different views to represent the data. Although not all of these are necessary, these are "by Date" which organizes the cards according to when they were created, "by Author" that organizes the cards in alphabetical order according to the author. Two necessary views, are "by Keyword" and "input". These are the views that the HennApplet to uses cross-reference cards when it needs to read in data and submit it.

The image location of the card is stored in the field, *image_location*. This is not a rich text keyword field, but rather a string that contains the URL of the image. Images are stored in a different Notes database. Over the web, the images are downloaded directly from this database based on this URL string. In the future, the actual image will have to be stored within a rich-text field of the Documentation database. Referencing the image, from the Images database within the Documents database is a possible way to accomplish this.

The Henn programmers prefer to use one large database for all projects rather than having multiple databases of self contained databases. This allows them to search and sort the database according to the project they are working on. It also keeps the data more centralized. This implies that each card have a field containing the project name as well as the rest of the card attributes.

In the future development of this project, the keyword will not be used to refer to a card, since this may be changed, is not unique, and causes unnecessary complication when saving. The current state of the database is that it contains the forms that represent the cards, but does not save the BrownSheet areas currently. In the future, there will be a new form called a "BrownSheet Box" that will be used for this purpose. Also, some fields within the card form (such as the author) will be removed to simplify the design.

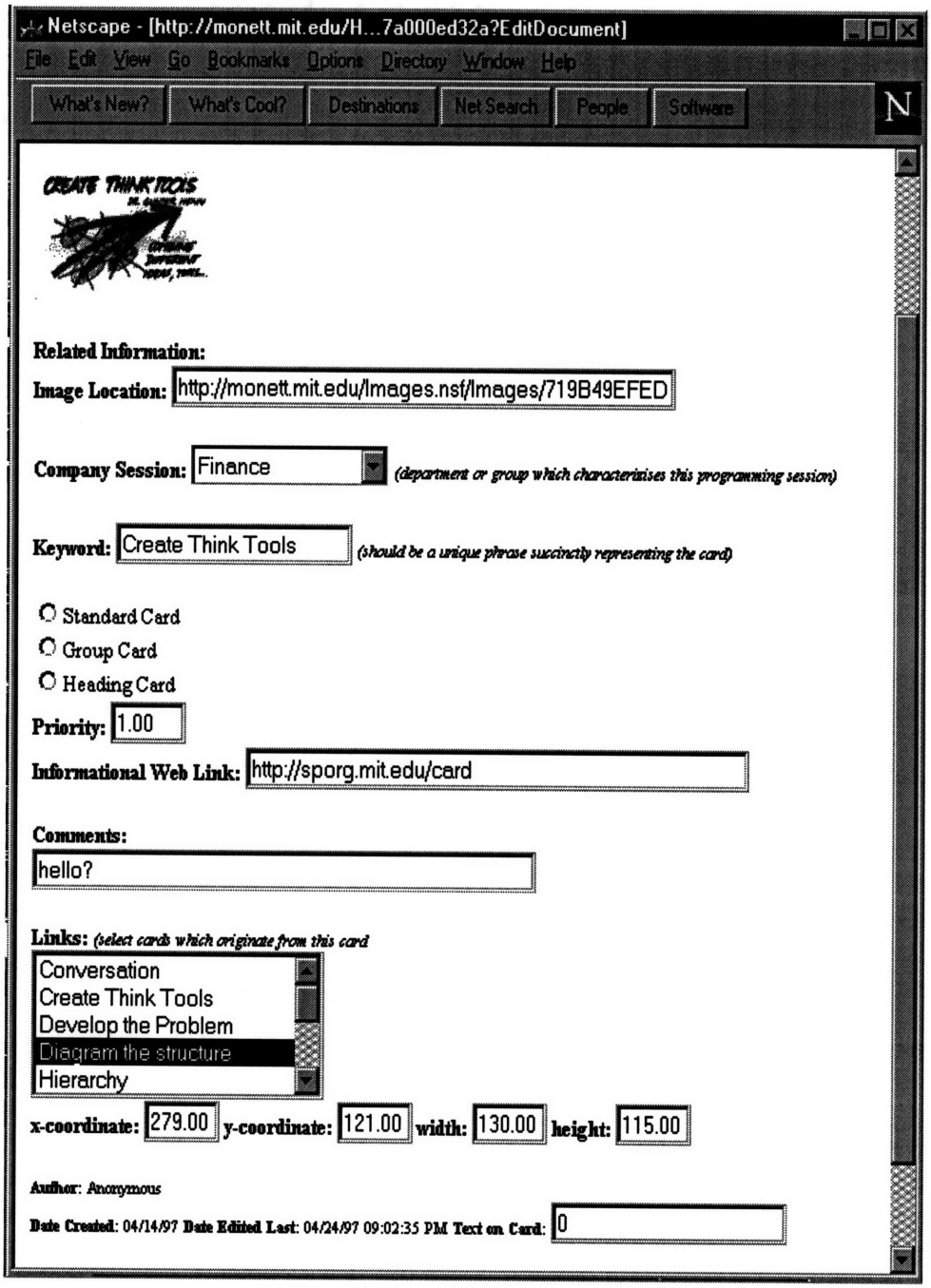


Figure 4-4: Fields within the cards' database can be edited via the web.

4-6 Use of the HennApplet

From the HennApplet the user has the ability to link other databases and to edit the information within them. Because of security reasons, these databases must reside on the server that the applet came from. The ability to modify information comes from within the interface of the Henn applet itself, and also to URL links to the database through the Domino web server. This means that a user may open the HennApplet and create a number of cards, modify those cards and then save them to the database. The user can also use a web browser to access the database of cards and modify them via HTML forms. From the web browser the user may also delete or create new cards. The user can change images, edit text, make comments or modify the links. The next time that the user loads the HennApplet, the changes to the web browser are revealed within the canvas.

Security restrictions on applets also prevent images placed on cards from coming from other machines than the machine the applet came from. The current version of the HennApplet has a link to the Images database that allows users to upload image files from their local machine. Once the images are added to database, they may be referenced by the cards. This may not be as easy as adding the image to the card directly from the local machine that the applet is running on.

The following paragraphs list the features of the CardWall and BrownSheet that are currently operational.

The user interface section is the Java part of the Card Wall. This section provides an interactive area for users to brainstorm ideas collectively in an ad-hoc manner. In this region, users may add, modify and link cards. When they are finished with organizing their cards and connecting them appropriately with the cards of others, they may save them to the database. Users do not have the ability to delete cards in this section of the program.

The database stores all the cards. At any time, users may enter the database and modify cards within. This provides an alternative method to access cards and links other than the User Interface section. If users want to delete cards, they may do it in this area. The database is organized with a number of different views such as "by Author", or "by Session".

The Image library will be used to store the images that are embedded in the cards. This means that images that were used for previous cards may be reused and easily accessed. All cards that contain images must have their images in this library (Java has a technical restriction on programs retrieving any picture on the network).

Using the CardWall

To add a new card: In the User Interface, press the new card button on the Tools panel. A new card will appear in the upper-left hand corner of the CardWall. It will automatically have a name like "new card created: Sun Jan 21. 19971.003" this is a default name for the card, which you may change. Do not give two cards the same name.

To edit card information: In the card property box, edit the text you wish to change. Press update when you are finished if you would like to change the information. Check to see if the information has been updated by revisiting the card. Press the save button to save the changes to the database.

To add a picture to a card from the card library: Enter the image library. Click one of the images and click the left mouse button while the mouse is over the image. Click "Copy Image Location". Go back to the CardWall and click the card you would like to have the image pasted into. Go to the field marked "img" and copy the URL into the field either by pressing CNTRL-V or clicking the left mouse button and pressing "paste".

To delete a card or links: You may not delete cards or links within the Card Wall User Interface. Go to the database and select a view. From there, click on the card you would like to delete. At the top of the card document you will see a marker which says "delete this page".

To pop up a window referring to a web site: Hold down the shift button and click the card. A new web browser will pop up containing the referred to link.

To reload the CardWall: Hold down the shift button and simultaneously press reload.

Toolbar

 Save to Database: Save all cards to the database, including their locations.

 Move Button:_Use this button when you would like to stop linking together cards when you click on them and would like to just move them instead.

 New Card_Use this button when you would like create a new card (idea).

 Link Button: Use this button when you would like to link together cards to show some relationship.

 Links on Top: This toggles the links between being drawn on the top or the bottom of the cards.

The CardInfo Dialog Box is used to inspect the different fields of the cards. As each card is touched on the screen, the information is updated in the dialog box. The information within the cards may also be edited with the CardInfo dialog box. Not all of the attributes are shown on the CardInfo dialog. Although it is not complete yet, there will be a BrownSheet area information box that will perform similar tasks.

The Menu applies to both the CardWall and the BrownSheet. This provides functions such as changing the shape of the grid or linking on/off. The features are described below. Some of the features found on the tool bar are also found in this menu.

In the Beta 1.0 version of the HennApplet, there is another tool bar for the different arrangements of the cards. Each of the arrangements is based upon an algorithm that arranges the cards depending on how the cards are linked. This tool bar has not been moved into the newer version yet.

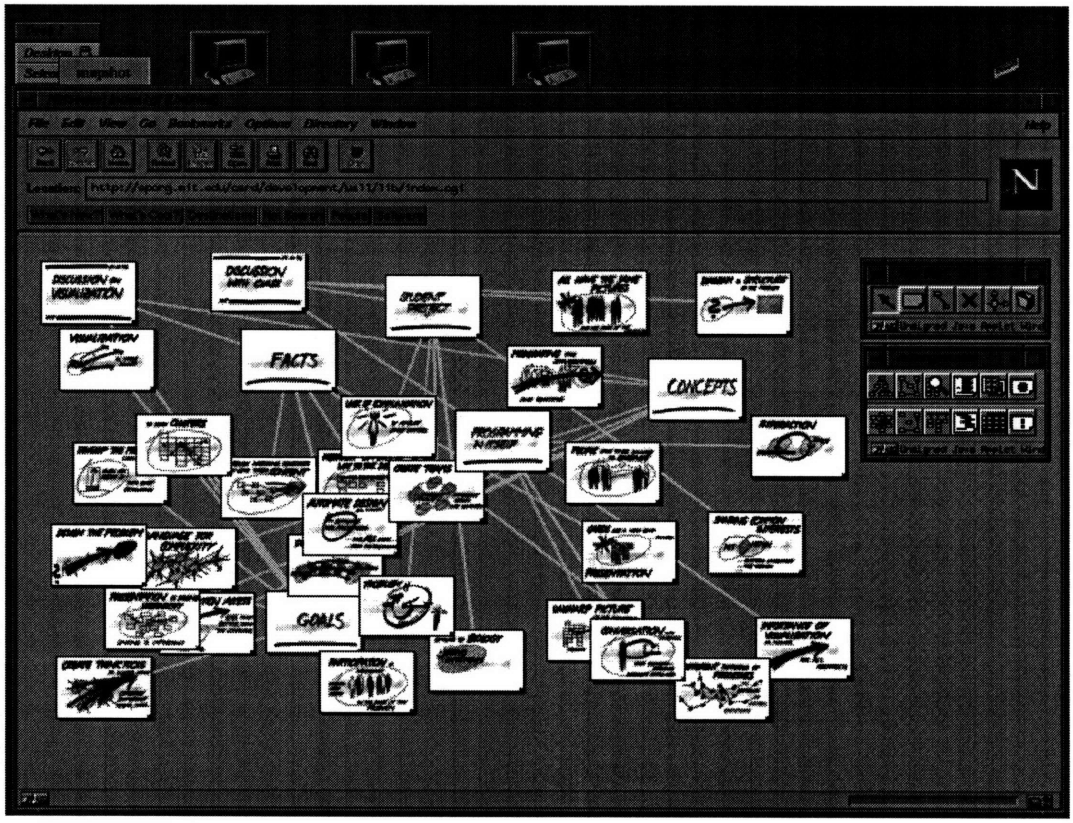


Figure 4-5: Beta 1.0 version of the HennApplet. This applet was presented to Henn in December 1996 as an initial prototype. This screen-shot shows un-arranged cards with connections between them.

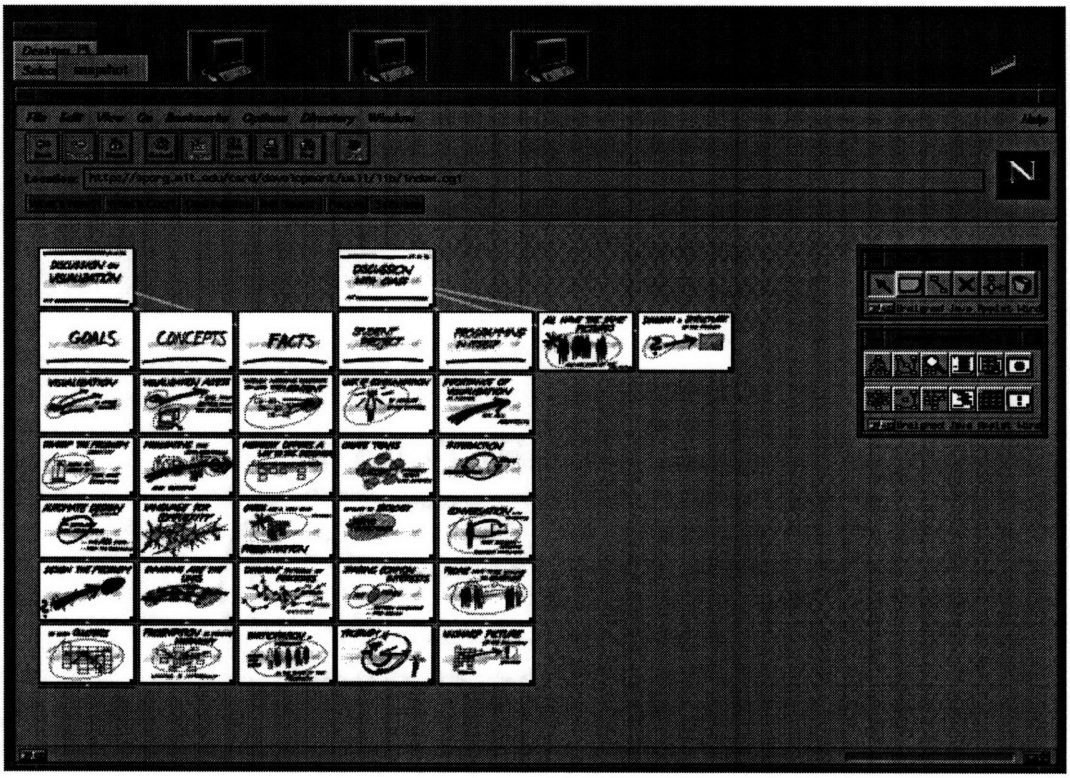


Figure 4-6: Beta 1.0 version of the HennApplet. This screen-shot shows the CardWall after pressing the CardWall arrangement button.

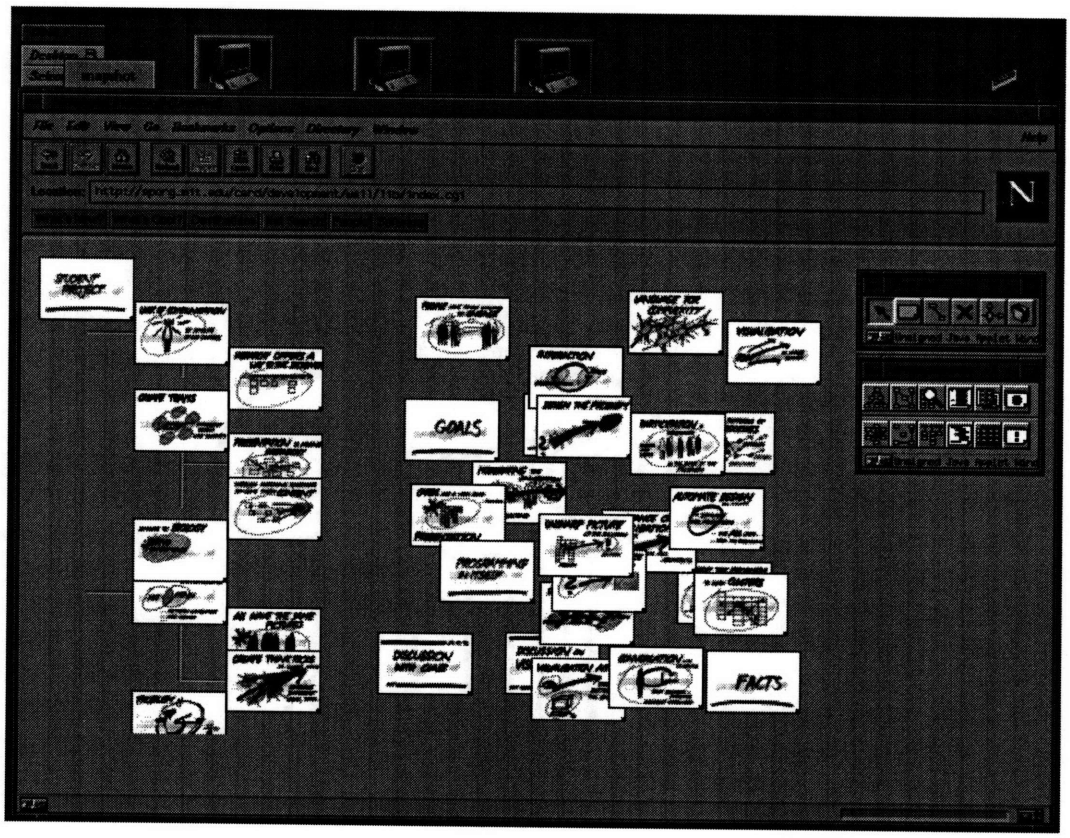


Figure 4-7: Beta 1.0 version of the HennApplet. This screen-shot shows the CardWall after selecting a top card and then pressing the hierarchy arrangement button. The algorithm leaves the unconnected cards to the right of the hierarchy.

Figure 4-7: Beta 1.1 (latest) version of the HennApplet. This screen-shot shows the different modules that connect to the cards database and image library. The editable dialog box and menu are both visible.The brown sheet is still largely undeveloped.

Chapter 5 -Technical

This chapter covers technical information concerning the structure of the CardWall program not described in chapter 4. The issues are covered in sufficient detail that this resource may be used as documentation for future development. In order to explain why certain design decisions were made in the development process, this chapter contains reference to the original prototype CardWall presented to Henn Architects in December of 1996. This version will be called Beta 1.0. The most recent version (the version that exists at the time this paper is being written) will be called Beta 1.1.

5.1 Component and Event Models

The Beta 1.0 version of the CardWall prototype used Java AWT (abstract window toolkit) components for each of the cards. This yielded the necessary functionality of Cards that have images pasted into them and the ability to move around via drag and drop. These components follow the Sun philosophy of using native components and windows (Win32, Motif etc.). The event model used was the standard, hierarchical event model from jdk1.01, which propagates events up the class hierarchy until some component finally catches them.

5.1.1 Beta 1.0 Version

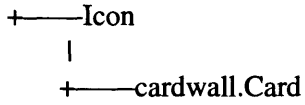
The model proposed for the Beta 1.0 version was adopted from a set of classes written by Marty Hall (Hall, 1996): (http://www.apl.jhu.edu/~hall). The class hierarchy was of the following structure:

ImageLabel: A class that creates a Component that contains an image.(http://www.apl.jhu.edu/~hall/java/ImageLabel/ImageLabel.html). This class downloads an image from a URL and paints an image within the component. It also resizes the component so that it is the same size.

Icon: A class built on ImageLabel that supports drag and drop for ImageLabels i.e. draggable pictures/icons. This part adds the moveable functionality of the card. Does not support "empty-box" dragging, the entire component moves along with the mouse.

Card: Built upon the previous two classes with additional support for information contained within. This class also needs the ability to resize cards, have links attached to them through drag and drop, and the ability to add text to the Card.

```
java.lang.Object
  |
  +----java.awt.Component
       |
       +----java.awt.Canvas
            |
            +----ImageLabel
                 |
```



These classes can be used in two different ways. The first is by capturing events within the canvas that holds the Cards. This method requires some modifications to the handleEvent method of the canvas. The other way is to allow the Cards themselves to pick up the events. For Beta 1.0, the Applet was sub-classed to create a new class called the GraphPanel. This class contains the cards and deals with the interaction, by using the first method, in which the GraphPanel fields the events. The events were handled in the following manner:

When an event is captured within the GraphPanel, it is passed to the handle event method of the GraphPanel. This method calls a static member of Icon and checks if the event can be handled there. If not, the event is returned. Otherwise, inside the Icon's static IconHandleEvent method, the Icon has access to a reference to the Container (GraphPanel) class. Within the IconHandleEvent method, a call is be made to the GraphPanel's handleEvent method that goes through all the components of the container, searching for the one that has the coordinates of the event that occurred. If the Card is found, it calls a normal (public) method of the particular Icon, and labels that Card the targetIcon. It passes the event to the Card, but this time, it has to translate the coordinates into the local coordinates. The function called is handleEvent of the Icon. This method takes care of moving the Icon. Finally, the event object goes to MouseUp, MouseDown and the other events of the Icon class.

Some problems that began to arise with this complex system arise from the fact that the Card class is a subclass of the Icon class. Also, it is bad design to have behavior passed through the inheritance hierarchy. The Card needs to have it's own behavior not shared by the Icon (i.e. the Icon provides the ability to move, while the card should be able to be resized). How do the events get passed to the Card, without having to move the behavior to a different class? One way might be to move the HandleEvents method higher into the higher abstraction level. Another would be to have separate functions to take the mouse events that would be declared "abstract". Then the events could be passed as a Card or some other subclass. However, those events would still have to be screened for behaviors before they are passed on. It is also possible to subclass the handleEvent methods of the Icon, but this replicates code and moves code meant to be within one class to another.

This complicated event model was making Java development more difficult as the specific needs became more complex. The model was forcing changes to code inside the Icon class to get the needed behaviors. The three classes thus become less distinct and independent. Maintenance became more difficult since code is required in multiple classes, and behavior could not be isolated. Since events must go through the two super classes, it was becoming necessary to mix up the behavior between the classes. This was

especially a problem when trying to subclass the Cards into StandardCards and GroupCards, which have different behaviors than Cards.

It is important to distribute the behavior in an object-oriented fashion throughout the class hierarchy, but to leave more specific behavior in lower classes. A class that is higher in the hierarchy should never know about the behavior of the classes which derive from it (Although it is perfectly good style for an object in a lower class to know and use the methods that come to it through the inheritance process). Other behavior should be easily over-ridden in lower classes to avoid complication. Ideally, the events would be completely separate from the code describing the Cards, and the behavior could be added or taken away with ease.

There were also other problems with the AWT itself. One was that the components themselves were too "heavy". The Java 1.01 AWT uses the native components through peers that communicate with the Windows, UNIX or Macintosh windows systems. A CardWall having 100 Cards would take up almost much memory as a program having 100 dialog boxes open. This was causing problems with all but the most robust systems, particularly for the Macintosh. The AWT components are also opaque. This means that there is no way to draw lines on top of the component without drawing it inside the Component itself. The Components are platform dependent, forcing the look to be different for different systems. Finally, it is sometimes difficult to gain access to Components that come precompiled. Often private fields within the superclass from the AWT need to be accessed and do not have pre-written public accessesor methods.

5.1.2 Beta 1.1 Version

For the Beta 1.1 version, a different toolkit was used for the Cards. The toolkit used is called LTK or Light ToolKit (Laffra, 1997). It provides a number of features that are better tailored to graphical interfaces requiring complex interactions and moveable objects within a canvas. The features included a callback methodology for receiving events, abstracted double buffering for all Graphical objects, the use of light widgets, and abstract paint methods.

The LTK uses Graphical objects instead of native window resources for Components. This means that all of the components are "faked" (drawn) using the drawing resources of a single native Component, the DisplayListCanvas. Each of the Graphical objects contained within the canvas are contained within a linked list. Whenever one of the Graphical objects is moved, the screen is automatically repaired by the DisplayListCanvas, which loops through the linked list, painting the Graphical objects in the area from which the object was moved. This allows movement of Graphical items without having to redraw the entire screen. This is a particularly desirable quality for the CardWall.

Another feature of the LTK that is useful for the CardWall is the way that the Graphical objects can be rearranged in the linked list so that the objects may overlap one-another.

By rearranging the order of the linked list, objects may appear on top or below each other. This is necessary for the layers concept which was presented in the chapter, Design Concepts.

The event scheme works by adding a "client" area to the DisplayListCanvas. The client is an area that waits to accept events within the canvas. A client can be specified by alerting the canvas that a certain area will be set up to wait for specific events. This is done with the addClient() method of the DisplayListCanvas. The addClient() method also specifies a connection between the client and any Object which implements the CallBackable Interface. This makes considerable progress from the AWT in connecting events to the appropriate object to receive them.

Based on this model, the following class structure is used for a Card:

```
java.lang.Object
  |
  +----ltk.Graphical
       |
       +----ltk.FilledGraphical
            |
            +----cardwall.MoveableDrag
                 |
                 +----cardwall.Card implements
```

CallBackable, Serializable (not used currently), DataObjectContainer (also not currently used)

Interactions Needed:
Move Button: MouseDown (Card), MouseDrag (anywhere), MouseUp(anywhere)
newCard: MouseDown(Anywhere)
linkButton: MouseDown(Card), MouseDrag(anywhere), MouseUp(Anywhere or card)
Delete CardButton: MouseDown(Card)
Links on Top: None
Garbage: mouseDown(Card)

Ownership of Behavior:
Card : moving, stretching, linking
cardwall: new card, delete card, link on top, mark as unused

Buttons:
Exclusive: moveButton,newCardButton,linkButton,deleteButton, garbageButton
Push-Down-Pop-Up: linksOnTop (wayLineButton)

5.1.2.1 Issues with the LTK for Beta 1.1

Is it possible to sub-class for behavior? Is it possible to implement additional behavior to

a single object, without having to encapsulate it all within one class? This may be difficult because of the specific class structure. For example, I have added a tab to the corner of the Card so that a user can click and pull to resize the Card. I want the Card class to also implement CallBackable because I want the parts that are added by the card to contribute those additional behaviors. But if I implement this class here, then my activateCallback(int method_nr) method will override the previous one and it will take the Moveable behavior away. One way to do this is to actually override the CallBackable function in the Card class, and to move the parts contained inside the Moveable class (from that function) into my new overriding function (it would be nice to have a function that inherits parts of other functions). This way, I could keep the code in all its respective places and no code is repeated throughout the class structure. The code brought down into the subclass is minimal:

```
switch (method_nr) {
case _mouseDown:  return mouseDown();
case _mouseUp:    return mouseUp();
case _mouseDrag:  return mouseDrag();
}
```

It would be ideal to have a switch statement that inherits switch statements from the other switch statements. Since the latest "clients" will be added to the beginning of the event queue, the areas that are specified as clients will be the first ones to receive that event. This means that even though the tab overlaps the main area of the card, the tab should be the part that receives the event first. The first problem with this is getting the small tab area to move along with the rest of the icon as it is dragged. If this issue is not addressed, the tab area will stay in one place, waiting for the mouse events even though the card is somewhere else. This requires certain logic in the Card class to ensure that area which the canvas is looking for moves with the rest of the canvas. The solution is to simply over-ride the MouseUp function from moveable. Inside this new function, it is necessary to remove the old client and add a new one in the new location. This ensures that every time a card is moved, the area that catches the mouse events for resizing are also moved.

To follow up on events that occupy the same client, but take different behavior, it is necessary to over-ride the function in a lower class, calling the super version of the function. This is the same as the problem with the AWT, where long switch statements were making the code very complex and difficult to debug. Inside that function, an if-statement checks for more specific behavior. This is also how the buttons in the ToolBar screen the events.
In order to update Graphicals manually, it is neccessary to pass the method reset() through the inheritance heirarchy. This way, it is possible to call the reset method on the lower classes and to add some additional behavior for the object. This is why the reset method was included in ltk.FilledGraphical and ltk.Moveable which simply calls the super class' reset method. The LTK toolkit is a bit more low level than the AWT was in terms of laying out panels. With the AWT you specify an abstract layout manager to do

this for you, while in the LTK, much of the laying out of sub-panels is done within the (Component) parent class itself, via calls to the reset method of specific Graphical. For example the Card contains an image with a text box below it and a WrappingTextBox behind the image (in case text should be displayed on the Card). Each time the Card moves, the Image, the text box and the WrappingTextBox must explicitly be moved so that they remain together.

Another problem that arose was gathering events while linking together Cards. Cards are linked together by clicking the first Card and holding down the mouse to drag an "elastic band" link to another Card. It is easy to have a client prepared for each of the Graphicals (added in the constructor of the graphical) ready to pick up mouse down events. Using "and" and "if" statements, it is possible to screen the mouse down events to signify that I want to link the Graphicals (as opposed to moving it). When I stretch the link, I need to have a mouseUP client which knows which graphical has just been accessed. But when do I add that client to wait for the mouseUP anywhere in the screen? If I add the client at the constructor of the Card, it may accidentally replace other clients. However, it will put a new client in for every card on the wall. They will also exist for the duration of the program. If I add the client at the time the mouse is pressed on the Card, there will be no way for the second Card to have a handle on the first client that was added. The solution is to add a static pointer to the card that was previously clicked. With access to this pointer available to the entire class it is possible to remove the same client which was added to receive the final mouseUp Event. Then, the link has access to the Card that the link originated from as well as the second card that was clicked. The Link object that is currently being used is made static within the Card class.

5.1.3 Additions and Changes Made to the LTK

1. put a reset() method in FilledGraphical and Moveable which calls super.reset()
2. changed the access of the variable focus_handler from private to protected
3. added some new functions to the drawing capabiites of the DisplayListCanvas (drawImage() can now take a location and a heght and width)
4. changed the following variable in the EntryField protected rather than private in order to subclass InvisibleEntryField: value, blink, font, client, fm_blinker, callback_method_nr, text_width, text_height, borderwidth, allowedChars, _keyPress,_keyRelease, _selected
5. in class Moveable, made the following variables public: applet, _mouseDown,_mouseUp, _mouseDrag
6. made applet in Display List Canvas public with an accessor function called getApplet(), so that I am able to launch a web browser as a response to someone clicking a card from inside the card class
7. made the constructor in the class Moveable public
8. separated out the code in the line class, put the draw function separate as a new function called drawThickLine()
9. add a draw function to Moveable which just calls the superclass' draw method
10. added a finalize() function to the Moveable class which takes away the display list

canvas

11. still need to fix bug in the lowerGraphical method of DisplayListCanvas

Even though the new LTK is lightweight, it must be downloaded during a remote session, rather than using the classes provided with the web browser at the client side. This may have the effect of causing slower speeds in the initialization of the Applet that counteracts the positive effects of using the light tool kit. Everything from the toolkit that is used must be brought over the web. The main problem with Java initialization speed is that separate connections must be made for each of the classes that are downloaded. The solution is to archive the necessary classes within a single zip file. Although earlier versions of Java do not support compression, the benefits of making a single connection are impressive. In Java 1.1 the JAR archive file format is supported which allows compressed class files. In either case, it is important that the original directory structure of the classes are contained within the zip or JAR file, particularly if the contained classes are in packages (as is the LTK).

5.2 Interaction Model

Another issue concerns passing messages from one class to another if they are not in the same package. For example if I subclass Frame, and from this new class I have a dialog box pop up. How is it possible to get the dialog box to call functions in the parent class without knowing what those functions are? As a rule, it is a good idea not to have definitions of functions from one class embedded in functions from another class. This is bad coding style. One way to do it is similar to the callback method, where all of the button events are defined a static final integers. This integer is passed to the CardWall that contains a hash table between the numbers that are coming in and the functions that should be called. This is nice because it makes the classes more autonomous and hence more easily reused. The other way is to use the Observer and Observable classes from the Java class libraries.

How is it possible to get messages from the dialog box back to the frame that owns it? The Frame has a reference to the dialog box, but the dialog box cannot have a reference to the Frame unless is is a superclass reference. In this case, the functions defined in the class that inherits from Frame are invalid (anyway, it is not good coding style to put references to functions in another class that may be prone to change later on). You should not have to put all classes in the same package to do this. One way might be to call some function that you know the Frame has, such as handle event. This can be over-ridden in the subclass of Frame. An option is to use the Controller-Model-View paradigm for object oriented development in Java.

As any program becomes more complex, there is a need to de-couple the objects in the program from the program itself. In Object Oriented terminology, this is known as the Controller-Model-View paradigm and becomes very useful for graphic user interface design. One can imagine a card wall with various entities, which work together independently of one another. Responding to events and interactions from the user, the object's

"do their own thing" without having to know the behavior of the other objects.
The Controller-Model-View paradigm separates the application into three parts, the controller, the model and view. The model refers to the actual body of the program, which may do computations, retrieve data or contain the application logic. The view and the controller parts refer to the user interface. The controllers make connections to the model, and the view represents the state of the model in a particular way. (Wutka, 1997) The ideal application will separate the three completely, particularly in the case of the model vs. the view and the controllers. In the CardWall program, the "deck" of cards within the model should contain all information about the cards, and the user interface should made up of the DisplayListCanvas and any other dialog boxes which affect the program. The parts that make up the user interface should only be "dumb" elements that send data to the main model. In the latest version, this is not the case, the data is contained within the Cards rather than the deck in the model.

For example, in the CardWall Beta 1.1, there is a window that displays information about the Cards. The window displays the URL that the card may refer to, the creator (author) of the card, the date the card was made, and the URL of the image the card contains. Each time one of the cards in the CardWall is clicked, the dialog box updates its information according to that card. This means that there must be a message passed from the card (which receives the mouse event) to the dialog box. The most obvious way to handle this is to implement a static method in the dialog box that is called when the mouse event is received from the card. Doing this also means is that the Card is now dependent upon the dialog box.

If the name of the dialog box is ever changed, or a new type of dialog box is set up, or the function name changes, the Card will be invalid. As the program increases in size, the classes eventually become one large glob of dependant items. A card is not separate at all from a link or a cardwall. Static links refer to other objects and destroy the object model. Ideally, the card should have no knowledge of the dialog box, what its functions are or how it implements them. In fact, cards should never make mention of them. This can be applied to all objects in an Object Oriented schema.

Using Controller-Model-View means setting up a layer of software abstraction which enables the card to send messages to the dialog box (or vice-versa) without having to know who receives that message or what is done with it. In Java, the Observable class and the Observer interface are used to do this. An object which inherits from Observable notifies any Observers (it doesn't know who) any time relevant data has changed. It does this first by setting a flag with the setChanged() method and then calling notifyObservers(). The objects that extend the Observable class, are usually simple groupings of related data. Whenever the data in the class changes, the calls to the observers are automatically made. The observer's call is a method called update() which can contain the reaction to the event that the data has changed. In the case of the dialog box, the changing of the variable last_mouse_clicked within the class CardWallApplet should give rise to the fields in the dialog box updating to represent the appropriate card. This

means that the integer last_mouse_clicked can be an ObservableInt, a class deriving from the observable class and holding the value of an integer. Likewise, the variable for each of the strings that represent each of the input fields are ObservableStrings, a class which inherits Observable and notifies the cards any time the input fields hav been modified. This method is much cleaner and separate than using static variables. The program makes the CardWallPanel class an Observer, and puts a field "selected" in each one of the cards. This field is an Observable and hence creates the ObservableBoolean class. Whenever this field changes, the CardWallPanel is notified and the field last_applet_clicked is reset to the new card. The only complication is that a reference to the applet must be passed to the card in the card constructor. Passing in a generic Observer instead of the Applet alleviates this problem. Since the CardWallPanel implements the Observer class, it is an Observer and hence can be passed into the function without the dialog knowing what type it really is.

One problem with passing an instance of a CardWallPanel as an argument of a function to the constructor of the Card is that it completely reduces the modularity of a card. Although it may seem that the occasion to use a card in other programs may never arise, it sometimes will. For example when I attempted to write a test program that wrote cards to Notes databases, I was unable to make a temporary card without having a CardWallPanel object to pass into the constructor. It is impossible to pass in the CardWallPanel super class, the HennApplet or the LTKApplet because you can not add the Applet as an Observer to the selected field. The update that was called when selected changed had no meaning to the applet at all. The solution is to use another class CardRegistry (Wutka, 1997) as an intermediate.

The CardRegistry is also an observable and can call update to its Observers when cards have been added to it. Instead of passing a CardWallPanel into the constructor of the card, I can simply pass in a CardRegistry and add the card to the registry upon its creation. This makes testing easy, for now if I want to create a bogus card in a test program, I simply have to create a registry object and pass that into the constructor.

5.3 Two Schemes for Saving Cards

As the complexity of the CardWall has increased, the need for more organized information management is unavoidable. At the early stages of the CardWall, the CardWall applet makes a connection to a URL and reads information from a text file that specifies information about the card. The information which comes from the text file is parsed out and used as arguments for the constructor of the cards. The cards are built, and then placed upon the CardWall.

5.3.1 A Basic Java Server and File-Format

When the user hass finished modifying the CardWall, the Cards are saved to a file via a basic server that simply accepts the data and writes it out to a file. The Java security restrictions prevent the applet from writing data to anywhere on the local machine.

The text file has the following format:

<type of object ("heading" or "card" or "link")> <keyword> <link to an image> <text associated with the card (used for explanatory information)> <the author of the card> <the date the card was created in the appropriate format> <the priority of the card> <x-coordinate> <y-coordinate> <a web link which the card referrs to and lanches in the actual program>

An example file is as follows:

heading discussion http://sporg.mit.edu/~keel/java/Project/Cards/card01.tif.gif this Arlene 10/22/96 0 100 100 http://web.mit.edu/cdalsass/www
card problem_time http://sporg.mit.edu/~keel/java/Project/Cards/photo.gif this Arlene 10/22/96 0 100 100 http://sporg.mit.edu/card
card paint_picture http://sporg.mit.edu/~keel/java/Project/Cards/paint.gif this Arlene 10/22/96 0 100 100 http://sporg.mit.edu/card

The problems with this are as follows:

1. Text and keywords can only be one word, unless some delimiter is made between the groups of words.
2. The text file must be perfect, or will be likely to cause an error. This includes carriage returns at the end of the file.
3. Url links are long and difficult to write accurately.
4. Information is scattered and not local.
5. There are no sorting or searching capabilities.
6. Adding additional information is a difficult task, requiring modification of the program (e.g. adding a time stamp signifying when it was last modified).
7. The formats for dates must be a certain way and cannot be enforced upon input.
8. There are no methods to enforce the integrity of data.
9. There is no ability to export the information in a different way, for posters or making books of cards.
10. There is no ability to organize the data (into projects for example).

5.3.2 Saving to a Notes Database

Functionality required:
To read all the files from a particular notes view This can be done by requesting a particular view, and then getting all the keys from that view. Each of the keys represent a card or BrownSheet (document) name.

1. To pull the data out of individual forms and be able to strip that data out to be be used for variables within the Java applet.
2. To be able to write to existing forms.
3. To create new forms.

4. To be able to know if forms already exist, if they have a need to be modified.
5. We don't want to submit data for a particular form without it's ever being changed. We would have to look at the time stamp of the data to see if has been modified and look at the time stamp of each of the card.
6. To be able to know if new cards have been created or if there have been changes while editing of the card wall is taking place.
7. To do keyword searches on cards.
8. To organize cards in different ways.

The web address for accessing individual Notes documents which have known entries is as follows:

http://hostname.mit.edu/CardWall.nsf/by+Keyword/the+problem+time

If the entries are unknown, you can look at a view http://hostname.mit.edu/CardWall.nsf/by+Keyword to get the entries.

Use the following syntax to create a new empty document
http://hostname.mit.edu/CardWall.nsf/Card/?CreateDocument

To input data, you will have to use POST and GET statements. This will send the data into the appropriate document. Use the CGI variable which comes within all requests from the web client; Content_Type and Query String.

Content_Type For queries that have attached information, such as HTTP POST and PUT, this is the content type of the data.

Lotus Domino supports ENCTYPE="multipart/form-data" for posting form data. This defines a protocol for how form information is sent from the web client to the server. It differs from the traditional form processing protocol (application/x-www-form-urlencoded). The diferences are that multipart/form-data is better to attach (upload) files that application/x-www-form-urlencoded. It contains a field expressly for that purspose, and may contain binary or text. The latter may add the information from the fields of the form to the url of the next request using the GET method, while the former must create two connections, one to contact the web-server and the other to send the file. multipart/form-data can only be used with the "POST" method of sending information and hence must complete the sumbission with two transmission steps (Musciano; Kennedy, 1996).

It is entirely possible to create Java classes from the cardwall applet which submit this type of encoded form submission input. Using the URLConnection class, it is possible to use HTTP formats to submit data directly. Using the notation of the Domino server allows the applet to attach commands at the end of the URL followed by a question mark. These parameters are passed to the Domino server and run the appropriate scripts. These are called web commands, examples are OpenDatabase, OpenView and ExpandView,

OpenDocument etc.

A typical submission from the applet has the following structure:

POST target.mit.edu/Card?CreateDocument HTTP/1.0
Content-type: application/octet-stream
Content-length: 100
key_word=high card
web_link_image=http://target.mit.edu/pictures/card001.gif
web_link=http://web.mit.edu
priority=1
comments=this is a*text with a* few lines*delimited by asterisks
x=100
y=120
links=lowcard,fatcard,skinnycard
author=Charles
text_on_card1
w=130
h=100

It is essential that the design of a module to write HTTP scripts to the web server is modular and support multiple protocols. In this case, we may want to switch protocols and save cards to a different database perhaps. This means that the parts that are relevant to the HTTP protocol are made separate from the parts that have to do with Cards.

Functionality: Class structure:

o keep the one socket and make calls to the server. Do not reconfigure or make new sockets
o keep the knowledge that has to do with cards all in one class.
o have the ability to pass in different urls and command lines.
o keep the HTTP protocol all encapsulated in one class.

results: Although one of the design goals of this system was to make it independant from the database, there were some necessary simplifications which were made. These reduce the generality of this software. The readCards() and saveCards() methods are now embedded in the CardWall class. These are based on HTTP 1.0.

1. Users must always create a view called input with all of the fields to be read in.
2. The first column in this view should be something that will not be used (such as an Icon).
3. New fields may be added to the documents which are read in, however, there is no guarantee that they will map to any variable in the program. The source code will have to be modified for this to be done correctly.

4. There must be a view called "by keyword" which sorts the cards and has keyword as the first entry

Although, this implementation forces the format of the Domino server to be specific (in an HTML table) in terms of how it implements the pages, there is some flexibility left. It is entirely possible to add new fields to the form for the Card. This is useful if someone needs a different type of field for another view of the database, perhaps. If new fields are added to the form, they (of course) will not be added to the cards. This will require the class structure to be changed to compensate for the additional field. In the most general sense, different classes can be dynamically generated for different forms that exist in the database. Then, the applet will be a standard interactive user interface for all Lotus Notes Databases, no matter what forms are used. This would provide additional flexibility in terms of organization of data within forms. Similar to the workspace on the Notes client, a series of icons could represent different databases. When activated, these would open up to a "documents" view that would show the documents contained in the database. Documents cand be organized in different manners by using connections, or by placing the cards in different ways corresponding to any Notes view. Documents could be placed in layers, organized hierarchically without regard to the Notes "view" method of categorization. This provides a highly desirable and easy to use product. It can be used to sort out large web sites, provide instructional material, hold semi-interactive chat sessions, provide an interface to a complicated data system, and can contain other versions of itself.

5.3.3 Additional Notes on Saving Cards

1. When saving cards, we must go through all the last_edited times of each card.
2. If the last edited times are greater than the time of started_session_time, we must save the card into an old document. However, when we create the card, if we just set the last_edited field to the started_session_time value, and be sure to update the value of last_edited for each field during saving, we do not need to get this value from the database. We still must add this to the constructor of the card because it comes from (is stored) within the mainapplet.

3. If the date_created is after the started_session_time, then make the card into a new document using "openform".

4. When saving cards, set the value of last_edited to the value of the card session beginning.

5. All cards must be saved no matter what (because coordinates have changed). It is important to know which fall into this category, as we can just post the x and y values to the server. This means that when a person reloads, they will see the cards appearing in different places.

6. If the user changes the keyword of the card during a session, the new data will not be

able to be entered because the keyword no longer exists within the database. This requires that the original keyword be saved in a field called last_saved_keyword. This field must be updated every time cards are saved to the database. This is not the case if the card is an entirely new one.

5.4 Security model

Users are granted access to the Domino Web Server through basic authentication, the standard for Web security that is based on a basic challenge/response protocol. After sending back a response, users are given access to the databases that contain their names in the ACL. Normally, after attempting to access a database, the HTTP response returns a header with the following line: WWW-Authenticate: basic realm="/". After the password and username are entered correctly, the web browser sends this information back with every request.

This functionality is built into most web browsers. When the browser receives the challenge from the server, it runs a script which pops up a dialog box requesting the password and the username. The HTTP Protocol 1.0 [http://www.ics.uci.edu/pub/ietf/http/rfc1945.html#BasicAA] states that the client (browser) must encrypt the username and password into a single string with the format:

"To receive authorization, the client sends the user-ID and password, separated by a single colon (":") character, within a base64 [5] encoded string in the credentials. If the user agent wishes to send the user-ID "Aladdin" and password "open sesame", it would use the following header field:

Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ=="

In addition to the rest of the header information that the Java applet passes to the web server, this header information must be included. In most sites, such as the one used for a class on the "WorkPlace of the Future" (Bill Porter) the applet is actually embedded into another domino/Notes database. It would be nice to pass the user name and password to the applet from the browser's current caching. That is, if the user has already logged on to a site and a username and password are available, use them in the applet as header information.

If an applet is embedded into a protected Domino database and a user enters the site by entering an acceptable uid and password, and then attempts to load cards from a protected domino database with the same ACL, the cards will not be accessable. This is because the username and password are sent (once entered) as header information with every request. The applet attempts to mimic the calls of the browser and unless the username and password information is submitted each time a request is made, there is no way to authenticate users. Once the user enters the UID and password, they are cached somewhere within the browser and sent with every request as long as the current browser is being used. When a new browser is started, the UID and password are forgotten. It is a

challenge to retrieve the UID and password from the browser. It is necessary if the Applet wishes to use previous uids and passwords. This would be needed to authenticate a user in a way that is transparent to him as he moves through the site but between different databases.

In order for an unknown applet to get the strings which represent the UID and password there must be a considerable amount of trust associated with the applet and the browser. It should not be very easy for an applet to get someone's access information to sites, for once that information is obtained, it could simply send the data to another server containing a CGI script where it could be saved. (Flanagan, 1997) indicates that there is a security "hobble" on netscape browsers to access the information stored in the password field of a form. Although this is not the same as the basic authentication scheme done by the browser, it seems very difficult to reach that information.

One solution is to pop up a box at the beginning of the applet session to enter the uid and password information. This allows the cardwall db to have a different ACL than the page that it may reside in. In addition, there is never a guarantee that the applet will be embedded in a basic-authenticated site. The place where the applet resides should be separate from the applet itself and the applet should not have to depend upon where it is loaded from to know if it will be able to load the cards. Although it is an inconvenience on large sites that contain many parts, for the user to reenter Uid's and passwords, it is certainly beneficial in terms of the development of this software. Having the user enter a uid and password anytime they use the CardWall increases the modularity of the program and makes it much easier to export. This feature is not available in the latest version of the CardWall program.

5.5 Storing Links in the Database

Links are stored in the database by using a dynamic keyword lookup against the other cards in the database. This creates a pull-down menu that supports multiple entries for each of the other cards. This allows users to connect card1 with cards 2 and 3 by going to the form for card 1 and pulling down the list, and checking off cards 2 and 3. When the form is put into HTML format through Domino, and used over the internet, the HTML does have the capability to support multi-entry keyword lists. In order to use this feature, you must press control while selecting the cards to link to.

In the input view of the database, there is a field called links that contains the keywords of all the cards that are connected to the current card. A comma separates each key word. When the applet reads in the field, it stores the links as a large string within the card objects as it creates them. When all the cards have been read, it goes through a loop for each of the cards, parsing out the string of comma delimited links and looking them up against the other cards. If it finds the keyword, the link is created and added to both the to and from lists of the receiving and starting cards (this is used to support directional links). If it does not find the keyword that it parsed out of the link string, the exception is caught and the link is ignored.

5.6 Grid Class

The Grid class is designed to be a separate as possible from the rest of the HennApplet. The only interaction occurs when a card or BrownSheet area is moved. At that time it "asks" the grid if it should be realigned to a different place.

1. Constructor takes the size of the grid, size of grid boxes, size of spaces between
2. method get_closest(x,y) looks at the two coords and returns the two coords which are closest to the center of the nearest center of the center of the box.

5.7 Implementing a Remote pool of Objects to Store Data

The need to make the client "thin" has also resulted in the desire to pass the computation over to the server side of the program. The consecutive HTTP POST and GET methods are slow and causing the server and the applet to bog. If some methods can be made remote and placed on the server side, it can lighten the client side.

Changes made to the database are also difficult to modify. If different forms are used, it requires editing of the source code to read in the new fields. For each new field added to the Card, the output string to the POST method must be changed (for saving), the new field must be added to the Card class, and there must be a new variable passed into the constructor of the card from the readCards method.

Finally, it would be nice to have generic objects that hold data that comes from any data source. The data could come from a spreadsheet file or relational database. The DataObject model would be independent upon the source of the data. The applet would simply be receiving objects containing data, regardless of the source.

In a third implementation of the data-storage and retrieval model, a different architecture is set up which allows for the possibility of remote method calls to objects existing back on the host machine where the applet came from. This data may come from any disparate data source.

A remote vector of data objects sitting as a remote server on the host-machine is initialized by reading in data from the server and creating generic objects called DataObjects from the database. The initialization values from the database comes from a daemon running at different scheduled times. This uses the Java RMI (Remote Method Invocation) classes from Sun's jdk version 1.1. When the applet loads to the remote machine, it checks the Registry running on the host machine for any remote objects. It finds the remote object that is a vector containing the DataObjects and requests the DataObjects through a method call which returns the Vector. The applet will make one connection, quickly serializing the objects and passing them to the applet, where they will be constructed into cards. The Card class specifies which of the data inside of the DataObjects it would like to use for its local methods. For example, the card looks within the DataObject to find the w and h parameters that indicate how large the card will be. This

configuration should be the only change that has to be made for any class that uses the DataObject model.

The DataObject should carry a hash table of "key-value" pairs. The key is the name of the variable and the value being its variable value. Both of these should be Strings. It is up to the class that uses the DataObject to convert those strings to the appropriate format. In addition, any conversion of date-times or numbers can be done in the Card. It is better that the logic is encapsulated here than in the CardWallPanel class which should have little knowledge of the internals of the Card.

As the user interacts with the applet, the DataObject should not be modified to reflect these changes. This is because there are variables coming through the inheritance tree like the x,y location of the cards which also are stored in the DataObject. The DataObject should not be modified until is it ready to be written back to the database. Then, any appropriate conversions are made and the data is put back into the DataObject. Next, the DataObject is serialized back to the remote vector of other DataObjects. This is to know which Object replaces the previous one. Finally, any new Objects are created and old objects that have been deleted are removed.

Another way to do this would be to use native C++ calls from the Notes API within the remote methods. These method calls could retrieve information from documents within the Notes database, convert them to Java Objects and pass them back into to the applet through the return variable. This would be an ideal solution, since it would be fast enough for the data to be modified in real time. The remote functions would be called getDocument(String document_id) or getAllDocumentsInView(String view_id). These would return DataObjects or Vectors of DataObjects.

This functionality is necessary for the BrownSheet because many of the spaces will have to be saved repeatedly. In addition, it would be nice to create a card class that reads in data from a completely different form that the one currently used in the Notes database. With little modification, many different classes could read in data from disparate forms and databases and have no trouble with data that does not match up.

5.8 Current Problems with the HennApplet

One of the main problems with the current state of the HennApplet is that there is no ability to uniquely mark documents within the database and hence know if they have been changed. The current way that the HennApplet works is that it reads in the all of the cards at the initialization of the session. This data comes from the input view of the Notes database. Each of these cards has a keyword variable that is stored as a variable within the card. The keyword is editable by the author during the CardWall through the "card properties" dialog box and may be changed during the session. When the user decides to save the cards, the applet goes through the deck of cards, submitting HTTP form data to the web server. The saveCards() method knows which form to save the Card in via the keyword variable. If the user has changed this variable during the session, it does not

cause a problem, because the keyword that was most recently saved is stored in a variable called last_saved_keyword within the Card class. The last_saved_keyword variable is refreshed each time saveCards() is called. But if someone uses the same keyword name, there is a conflict. The saveCards method usually over-writes the first form that was submitted. Another problem occurs if someone removes a card from the database during a CardWall session. There should be a unique ID which never changes as a field from the input file which becomes a unique value to represent the card. Unfortunately, the FormUniversalID variable within Notes databases is not a good choice, because once a card is saved into a form, it cannot be resaved during the same session unless it is re-loaded. Since Notes gives the document this ID, there would have to be an additional transaction to find out this ID once it has been assigned by the Notes server. One option is to create a universal ID composed of a series of random numbers and letters during the CardWall session. This can be saved as an additional field in the input view. Another way is to actually reload the card after it is saved the first time, however since all data for all the cards comes in from a single input file, a new parsing routine would have to be devised to abstract the data from a different format. This would be a bad idea since it is dependent upon the fields within the Card document stored in the database. Finally, after each saving, all cards could be reloaded, which would yield the IDs of the new cards.

The current state of the HennAplet has an abstract method readCards()in the AbstractCanvas class. This is implemented in the BrownSheetCanvas and the CardWallCanvas classes to read in the appropriate data from the Notes database and create cards from them. As the cards are created, a reference to the card is stored in the Vector "deck" (belonging to the AbstractCanvas). This vector is used for rearranging the cards in views and moving the links to the top and bottom. Since the methods required to save and read in cards are relatively short, it is not a major difficulty to use this implementation. In the future, it would be better to have the source of data more independent of the application. This could be accomplished by using DataObjects, which could be a generic container of name-value pairs of Strings. This could also be used to simplify the constructor of the Card class, since a DataObject could be passed into the constructor rather than passing in individual fields. This would force the Card to parse out appropriate data types from the String, rather than having this done in the HennApplet. This would include any conversion of data (e.g. Strings to ints or dates).

The way that the menu works is that it provides calls to the AbstractCanvas. However, some of the calls to the BrownSheet and CardWall need to be sorted out. For example, when a user presses File-SaveCard, it should apply to both the CardWall and the BrownSheet (both AbstractCanvas's). However, when the Environment-Grid menu button is selected this should apply only to the AbstractCanvas which is currently selected. Environment-linking-mode should apply to both. It is important to sort out these events via a separate class so that the logic will not be in the wrong place. The way that this is done now is by over-riding the update() method in the BrownSheetCanvas and the CardWallCanvas and leaving and passing the events to the superclass if necessary.

The variable last_card_clicked within the AbstractCardWall is static. Each instance of AbstractCanvas should have a fresh copy of this variable. This includes the BrownSheet, CardWall and any others. This should be changed in the future. It is necessary to use Observables to make the variable last_card_clicked private.
It is not as easy as changing the status of the variable from static to private however, since it was made static to be accessible from other classes within the cardwall package.

One of the most urgent changes that must be made to the HennApplet is that the Object-Model-View must be implemented correctly. Although the controllers such as the grid dialog and card properties menus do a sufficient job of emulating this paradigm, the CardWall and BrownSheet are not effectively implemented. The interface must be totally separate from the data contained within the cards. This means that although there will still be Cards, the Cards should not contain the data. The deck in the AbstractCanvas should hold this information. The interface should simply notify the deck when changes have been made to the cards that represent the actual data in the model. This is two-way information, changes to the deck will have to be reflected back to the user interface and vice-versa. The Links should also be separate from the interface. It may be best to hold the Links within the objects within the deck.

Finally, the user interface must be worked out better. There are clearly too many (3) menus and control panels suspended outside of the browser. It is probably best to remove the buttons and use only the menu, which replicates the buttons anyway, and provides some text to explain the function of the button. A suggestion to simplify the user interface by removing the tool buttons and replace these functions which pull-down menus. The card property and the BrownSheet area property dialogs should pop up when the user selects them from the menu.

Finally, make the arrangements extremely modular. Create an interface for the views which allow the option of saving sets of x,y coordinates. This is not a typical arrangement that uses logic and links to reorganize cards, but is much simpler. This will be necessary as an option, however.

Conclusion

This paper carefully documents the process of Henn User Programming. Although retrieving data from disparate sources has not been easy, the documentation is useful because it clarifies the complexity and variation of the process. This process is not documented in any other materials I have found. Future developers of this project should find this paper a valuable reference.

This thesis also lists the types of improvements that the Henn programmers would like to see and provides viable design concepts to realize those improvements.

The analysis of the programming process has led to the conclusion that there are significant technical and practical problems to overcome. The main problem is that programming workshops are hindered by the time consuming task of scanning in cards. This does not seem to be a difficulty that will be overcome in the near future. In order for this software prototype to actually be used, its value must outweigh the inconveniences of scanning in cards. This indicates that the software must be considerably developed and should provide great value for the programmers before it is actually used.

The HennApplet prototype is still not in a useable state. Although the CardWall module has many of the capabilities that it needs, it is unwieldy and difficult to use. Most of the problems are the little things, like the amount of time it takes to download the applet, the fact that the applet reloads every time the web browser is resized, and the fact that the interface needs to be better integrated into the applet. It is also difficult to upload the images into the Images database once they are scanned in. One recommendation is that the CardWall and BrownSheet modules be incorporated into an application that is stand-alone (local) until the applet technology improves. The code could be converted back into an applet version in the future. A problem with the stand-alone application, however, is that the image and cards database is no longer available through the web medium from a stand-alone application. Another problem is that the software will no longer be accessible from anywhere – a loss of a significant benefit of the architecture of this application.

Another major problem with the prototype is that it is difficult to develop effective user interfaces with Java and the web browser. Future developments of the applet may yield better-designed user interfaces.

It is foreseeable that the BrownSheet module will be easier to assimilate into actual use than will be the CardWall. The BrownSheet applet presents rapid versatility without any of the digital content management difficulties present with the CardWall. Since the BrownSheet is not dependent upon the images used in the CardWall, it is much easier to develop a successful system that may be integrated into actual use. The advantage of not having to enter the data into a spreadsheet and having the BrownSheet available with different views is significant.

The development of the HennApplet reflects a basis for further development, as it provides basic classes and functionality needed to make the application more functional. The requirements for animation are also available with the latest (Beta 1.1) version of the applet.

Integration of the database has been reasonably successful, although the card document and views need some graphic design work. It is somewhat difficult to edit cards from the web interface. One very useful aspect of the database is that it provides a way to create documentation automatically via scripts.

There are some design problems associated with the present classes. The controller-object-view model must be realized before the code becomes too complicated. Also saving and reading in cards must be further refined, particularly in the areas of security restrictions, robustness and speed.

Reference List

1. Allen, Thomas J. *Managing New Product Development: Organizational and Architectural Issues and Solutions.* New York: Oxford University Press, (in press).
2. Flanagan, David. *Java in a Nutshell, A Quick Desktop Reference for Java Programmers.* O'Reilly and Associates, Inc. 1996.
3. Flanagan, David. *JavaScript, The Definitive Guide.* O'Reilly and Associates, Inc. 1997.
4. Hall, Marty. *ImageLabel.java, Icon.java.* Java classes. 1996.
5. Henn Architekten Ingeniere. *Programming.* Promotional Materials. 1992.
6. Henn Architekten Ingeniere. *Technische Universitat Munchen, BMW AG Facultat fur Maschinenwesen Garching Programming.* Programming Brochure. June/July 1991
7. Keel, Paul. *Process and Relational Analysis, Capturing Architectural Thought.* SMArch. Thesis. MIT School of Architecture and Urban Planning, Feb 1997.
8. Keel, Paul. *Waylines - An Architectural Method for Wayplaning.* SPORG Paper. 1995.
9. Kisiel, Arlene. *Model for a Teaching Practice, Development of a Prototype through Design Inquiry.* SMArch. Thesis. MIT School of Architecture and Urban Planning, Feb 1997. Appendix 3.
10. Kohlert, Christine; Gerhard, Ziriakus. *Boston MA. Interviews, Tuesday, April 15–Friday April 18, 1997.*
11. Laffra, Chris. *Advanced Java, Idioms, Pitfalls, Styles, and Programming Tips.* Prentice Hall PTR. 1997.
12. Malone, Thomas W. *Tools for Inventing Organizations, Towards a Handbook for Organizational Processes.* Working Paper. 1992 MIT Center for Coordination Science.
13. Malone, Thomas W. *Inventing the Organizations of the 21st Century, Control, Empowerment and Information Technology.* Working Paper 1995, MIT Sloan.
14. Musciano, Chuck; Kennedy, Bill. *HTML, The Definitive Guide.* O'Reilly and Associates, Inc. 1996.
15. Shiffman, Hank. *Boosting Java Performance: Native Code and JIT Compiler.* Silicon Graphics reference paper. 1997.
16. Shlaer, Sally; Mellor, Stephen J. *Object-Oriented Systems Analysis, Modeling the World in Data.* Prentice Hall Corp. 1988.
17. Wutka, Mark. *Hacking Java, The Java Professional's Resource Kit.* Que Books. 1997.
18. William Pena; Caudill, William; Focke, John. *Problem Seeking: An Architectural Programming Primer.* Washington AIA Press. 1977.