THE UNIVERSITY OF CALGARY

Programming Distributed Collaboration Interaction Through the World Wide Web

by

Roberto Augusto Flores-Méndez

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JUNE, 1997

© Roberto Augusto Flores-Méndez 1997

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled “Programming Distributed Collaboration Interaction Through the World Wide Web” submitted by Roberto Augusto Flores-Méndez in partial fulfillment of the requirements for the degree of Master of Science.

_____

Supervisor, Brian Gaines, Department of Computer Science

_____

Ken Loose, Department of Computer Science

_____

Robert Kremer, Department of Computer Science

_____

Steve Norman, Department of Electrical Engineering

_____

Date

ABSTRACT

This Thesis presents the implementation of a client/server concept mapping tool as a support for asynchronous workgroup knowledge sharing on the Internet and the World Wide Web. This tool is composed of a server process, named jKSImapper Server, and two client systems: jKSImapper and jKSImapplet, which allow concept mapping elicitation on the Internet and the World Wide Web, respectively. These systems were developed based on the jCMap class library, which is an object-oriented library implemented using the Java programming language. This class library was designed as a framework for the development of systems supporting abstract visual representations while allowing extensibility to support formal knowledge constraints. This Thesis also includes an overview of the issues required for downloadable code; a description of the characteristics of the Java programming language; and the lessons learned when constructing jCMap based on previously existing C++ code.

ACKNOWLEDGEMENTS

The work presented in this Thesis could not have been possible without the guidance and support of many people. In first place, I would like to thank my supervisor, Dr. Brian R. Gaines, for giving me the opportunity of pursue this work and directing my efforts to complete it. He has been an invaluable source of support and guidance all through my graduate program.

I would also like to thank Rob Kremer for the lengthy discussions we had that helped me to cope with the intricacy of object-oriented abstractions. Likewise, I would like to thank Dr. Mildred Shaw, Dr. Saul Greenberg, Dr. Dickson Lukose, Lee Chen, Mark Roseman, Carl Gutwin, Gladimir Baranoski, Jalal Kawash, and Pim van Leewen for their timely input and advice. A special mention goes to the people at the Computer Science Department (students, faculty and staff) for their unconditional support and friendship.

Many thanks to Dr. Jorge Carpizo and Wilhelm Pérez, who gave me the opportunity to enter graduate school in the first place. Thanks to José Alvarez Jr. for his invaluable job of proof-reading this Thesis.

At last, but not at least, I would like to thank my wife Cecilia and child Ricardo for being a source of motivation and encouragement. Thanks to my parents, Augusto and Ligia, for teaching me that I could accomplish anything I set out to do.

This work was supported by the Consejo Nacional de Ciencia y Tecnología (CONACYT).

To my wife, Cecilia.

To Dr. Herman W. Konrad

In Memoriam

Table Of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

1.1 AIM.

The purpose of this research is to evaluate Java as a suitable programming language for the Internet by using it to implement a concept mapping tool system able to support distributed user environments on the Internet and the World Wide Web.

1.2 MOTIVATION.

All over the globe, individuals face constant and pervasive changes that make it impossible, not just to transcend, but to survive under timely immutable states of knowledge. The challenge is overt: involvement in continuous education is indispensable to overcome such changes.

The learning web, which was first described by Norrie and Gaines (Norrie and Gaines, 1995), is envisioned as part of a learning society where individuals are submerged in a life-long process of acquiring knowledge to cope with evolving environments. Individuals will no longer succeed in isolation or with a fixed amount of knowledge, but rather they will operate as part of collaborative workgroups, where digital computers and networks will be used for communication and collaboration (Shaw and Gaines, 1996). In other words, computer networks will be used to communicate the experiences and knowledge acquired by each of the users on a distributed virtual community to all the members of the community.

Under this scheme, the present thesis will show the implementation of a concept mapping tool as a support for workgroup knowledge sharing on distributed environments. In this

implementation, the World Wide Web (the *Web*) is used as the hypermedia infrastructure and the Java programming language as a solution to provide portable executable code among the diversity of computer operating systems found on the Internet.

1.3 CONCEPT MAPS.

One of the mechanisms used to organize and communicate knowledge among individuals are concept maps (Gaines and Shaw, 1995a). The term concept map, which encompasses a wide variety of diagrammatic knowledge representations, can be defined as diagrams composed of links and nodes of different types. Concept maps can be used to graphically represent and organize arguments and thoughts, providing an alternative to natural languages to communicate knowledge.

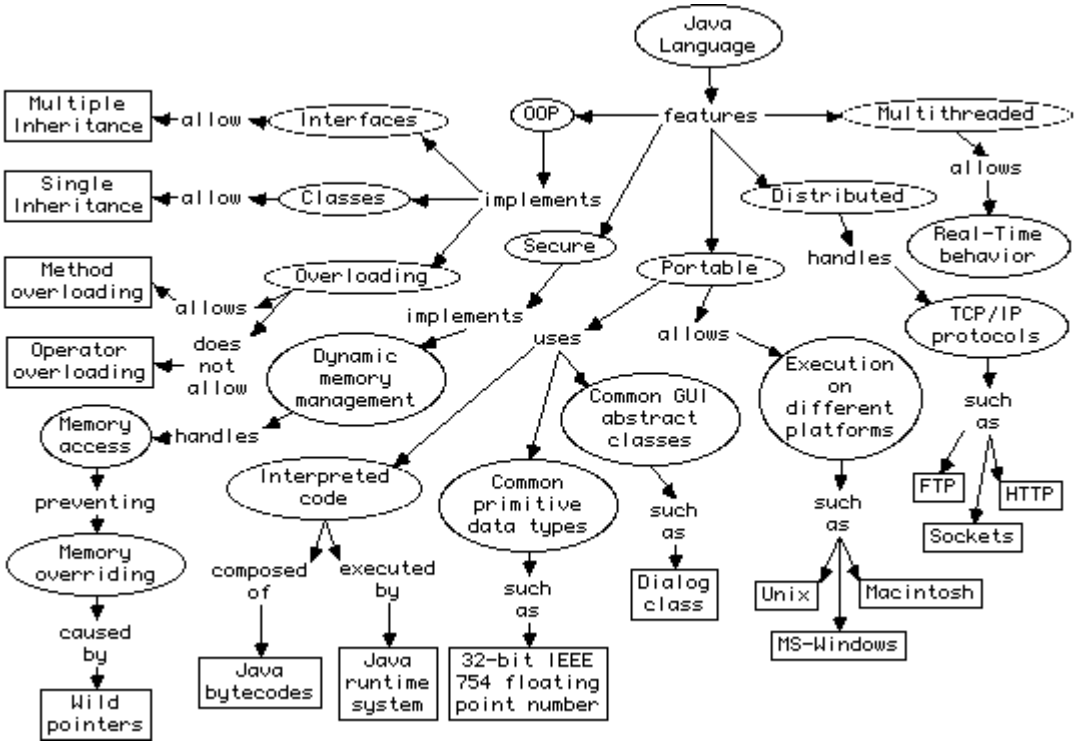


Figure 1. Concept Map on the Java Language.

Figure 1 shows a concept map, which illustrates features found in the Java programming language. In this concept map, circular nodes represent concepts, rectangular nodes represent instances, and labeled arrows represent relationships.

1.3.1 APPLICATIONS.

Concept maps have been applied on areas ranging from education (Novak and Gowin, 1984) and management (Axelrod, 1976) to artificial intelligence (Quillian, 1968) and knowledge acquisition (McNeese, Zaff, Peio, Snyder, Duncan, and McFarren, 1990). Even when some of the applicable disciplines may implement formalisms to interpret and organize relevant information, the test case implementation for this research does not enforce particular constraints. This makes the concept mapping tool applicable as the basis to support diverse formalism domains. In such cases, the concept mapping application can be extended to fulfill specific requirements.

Some scenarios where the test case implementation may be used as a groupware tool are:

- ***Brainstorming*:** where known concepts and relationships are suggested by individuals to conform a meaningful workgroup knowledge structure.
- ***Decision making*:** where current variables are identified on existing knowledge structures leading to logical inferences that, ultimately, will help for decision making.
- ***Information navigation*:** as a hypermedia organizer; where information can be encapsulated on different levels of abstraction for easy comprehension, while allowing expansion to reveal details when required.
- ***Presentation planning*:** as a tool to outline presentations' content in a non-linear manner. This technique allows a quick interpretation of related material and their relationship with the subject of the presentation.

1.4 CLIENT-SERVER COMPUTING.

Since their invention, computer systems have been in constant evolution. During the decade of the 1970's, most computer sites were composed of mainframe systems accessed by dumb terminals. Under this scenario, all data storage and computational power were provided by centralized computers. It was common then to interconnect these mainframes, creating large-scale distributed systems over computer networks.

In the 1980's, personal computers and local area networks attained wide popularity. This circumstance allowed their fast integration to the existing mainframe-based distributed systems. This new arrangement radically modified the centralized infrastructure for computation and data storage, allowing some balance on the execution of code between mainframes and personal computers.

Mainly because of their reliability, mainframes became the providers of services for the increasing number of satellite desktop computers, from where the term client-server is derived. Formally defined, client-server computing is a computer architecture based on client systems (systems requesting services) communicating over networks with server systems (systems providing services). This architecture constitutes the foundation of the client-server systems. Such programs divide their functionality by using code, located on both client and server computers.

1.4.1 THE INTERNET.

The Internet can be defined as a large interconnection of regional networks that communicate with each other using the TCP/IP protocol.

The Internet is essentially a communication infrastructure consisting of 4 elements:

- ***Network Computers*:** forming the physical platform to process, store and transfer data;
- ***Users*:** who are the providers and requesters of information;
- ***Services*:** protocols and standards used to organize, access and transmit information; and

- ***Information*:** which is formatted data with meaningful content.

The Internet started in 1969 as a project developed by the United States Department of Defense. Since then, the Internet has grown from connecting 6 computers on its starting year to more than 6 million computers on 1995. Due to its number of users and the amount of information and services provided, the Internet is considered the most sophisticated client-server computer architecture available. One of the factors that has fostered its wide acceptance has been the implementation of mechanisms to integrate dissimilar operating systems into a common communication environment. Such environment has been found to be suitable for the transmission of knowledge supplied by multiple sources, regarding multiple areas of expertise (Berners-Lee, Cailliau, Frystyk, Secret, 1994).

The Internet incorporates several mechanisms that support distributed and collaborative communities of users and provides access to diverse information resources and services. From the available services, the Web has become the primary standard for information retrieval and exchange, surpassing leading services such as FTP on the volume of information handled (Gaines, Shaw and Chen, 1996).

1.4.2 THE WORLD WIDE WEB.

Since its implementation in 1989, the Web has performed as a stateless client-server retrieval-information service for the Internet. It provides conventions that encompass virtually all Internet services, allowing members to access information on a hypermedia environment using a constant interface.

The Web is composed of several essential concepts, which are described as follows:

- ***Universal Resource Locator (URL)*:** Universal Resource Locators are text strings used to address resources. They are essential for the system to scale and for the information space to be independent of the network and server topology. URL addresses have the following structure (Berners-Lee, Masinter, and McCahill, 1994):

<*scheme*> **:** <*scheme-specific-part*>

where *scheme* represents an Internet protocol (such as FTP, HTTP or GOPHER), and *scheme-specific-part* specifies a string whose interpretation is dependent on the scheme applied. For schemes that use IP-based protocols, the scheme-specific-part will be compliant with the following syntax:

// [<*user*> [**:** <*password*>] **@**] <*host*> [**:** <*port*>] **/** [<*URL-path*>]

where the double forward slash indicates that the string conforms with the Internet syntax; <*user*>:<*password*> indicates optional values representing a user id and password (if required by the service requested); *host* denotes a fully qualified domain name of a network host; *port* indicates a port to request a connection (this argument is optional on services that specify a default port to connect); and *URL-path*, which specifies the details to access a specific resource on the host (e.g., path for locating a file).

One of the advantages of the actual URL syntax is that it allows not just accommodation to existing schemes, but also, to any other that might be developed in the future.

- ***Hypertext Transfer Protocol (HTTP)*:** Rather than just a protocol to transfer hypertext, HTTP is a protocol to transmit any information while providing the efficiency necessary to make hyperlink jumps. Although comprehension of the Hypertext Markup Language is required for Web clients, HTTP is used to retrieve documents in an extensive and unbounded set of formats. To achieve this goal, the client sends a list of data formats and the server replies with data in any of those formats it can produce (Berners-Lee, Cailliau, Frystyk, Secret, 1994).

- ***Hypertext Markup Language (HTML)*:** HTML is a simple markup language used to create hypertext documents that are platform independent (Berners-Lee, Connolly, 1995). HTML is based on a set of instructions called tags, which are used to represent attributes. Single tags represent atomic values (e.g., a line feed), and paired tags are used to limit the attribute's range of influence (e.g., starting italic text formatting, ending

italic text formatting). HTML allows the creation of documents containing formatted text and links to images, sound streams, video sequences, executable code, as well as form fields and tables.

In a common scenario, a client using a Web browser (which performs as the hypermedia interface), communicates with a server through the HTTP protocol to access resources located on that server. These resources are usually HTML documents containing formatted text and hyperlinks to other resources. Such resources may be located on the host site or anywhere over the Internet.

In general, users accessing the Web perform as requesters of resources, which may be any source of data specially formatted to reflect its content. Plain text, images, video, sound and executable code are examples of formatted data. Among the data formats that can be transmitted through the Web, executable code is the most powerful one and provides an array of services to users not provided by other data formats.

1.4.3 Executable Code on the World Wide Web.

As previously mentioned, executable code is the most powerful data format that can be incorporated to the Web. Information handled on the Web can be classified, from the users' perspective, either as dynamic or static, having interaction as the element to differentiate them. Text, images, sound and movies are examples of static content. As it can be derived, the presence of animation is not considered dynamic, since users can not modify the outcome of such predefined information. On the other hand, executable code does provide dynamic content. Executable code allows users to type instructions, select options and make decisions to obtain results according to their needs.

The Web, as a service supported by a client-server environment, allows execution of code both ways, on servers and client sites. The original location of the code and its place of execution are variables that can be used to classify executable code. As shown in Table 1, the variables generate four scenarios:

		Place of Execution	
		Server	*Client*
Location of Code	*Server*	CGI	Downloadable Code
	Client	Mobile Agents	Helpers and Plug-ins

Table 1. Scenarios for Executable Code on the Web.

All the scenarios shown are currently supported by Web servers, with the sole exception of mobile agents. Intelligent mobile agents are a very interesting and broad area of computer science, however, they are out of the scope of the present work. Current research on that domain promises to deliver concepts that may transform the way in which humans interact with computers and society. Further information can be found on (White, 1995) and (Harrison, Chess and Kershenbaum, 1995). Following sections describe the development achieved on the remaining areas.

1.4.3.1 COMMON GATEWAY INTERFACE.

The Common Gateway Interface (CGI) is described as a standard to communicate external applications with information servers (NCSA, 1996). On Web servers, CGI programs (which are usually small applications called “servlets”) are executed based on client request. Such requests are triggered either to obtain information located on the server, or to submit information for storage and/or processing.

Usually, CGI programs are complemented by HTML forms. A form is essentially a distinctive group of *widgets*, or user interface input elements (such as input lines, combo boxes and buttons), included in an HTML document to query information from users. Users interacting with forms can submit information to the server. The server will then act as a gateway to invoke the corresponding CGI program, and to pass the received information as a parameter. The CGI program will process the information and will send back an acknowledgment to the client, usually in the form of another HTML document. This scheme allows a recurrent process of interaction.

An interesting application to exemplify the practical use CGI and HTML forms is found on *WebGrid* (Shaw and Gaines, 1995). WebGrid is a repertory grid system developed in the Knowledge Science Institute at the University of Calgary. In WebGrid, user data represents information relevant to the state of an elicitation of constructs. That information is submitted to the server for processing. Afterwards, the server will reply with an evaluation of such information (e.g., a graphical model from one of the clustering techniques available), or with a new form to continue the elicitation process. WebGrid implements an elegant solution to overcome limitations on the stateless nature of the Web, by storing user information on hidden fields inside HTML documents. This approach has the dual benefit of eliminating the need for the server to store user information (since the client can save HTML documents for later use), and to maintain the state for each elicitation (but at the cost of recalculating all the information for each user request).

Without a doubt, the introduction of CGI and HTML forms represent a big step towards the integration of dynamic content to the Web. Unfortunately, it also introduced some drawbacks, such as:

- ***Increased server workload***: The server is penalized by supporting all the processing needed to execute a CGI application. This problem can be increased by the fact that different instances are started each time a CGI program is requested. This circumstance forces the server to have multiple identical processes carrying on the same task at the same time.

- ***Limited user interaction***: In most cases, users are limited to manipulate HTML forms. This approach might be suitable for simple database applications, but not for systems requiring immediate response over users' events.

These limitations have lead to the search of alternative techniques to improve client interaction while decreasing server workload. As described below, an alternative solution is achieved by sharing some of the processing workload with client computers.

1.4.3.2 HELPERS AND PLUG-INS.

Helper and Plug-in applications are part of an evolution process for handling and embedding proprietary information into Web browsers. They share a basic functionality, since both types represent client-resident programs designed to manipulate information that browsers are not programmed to handle. However, they differ on the degree of integration achieved with Web browsers. While helper applications are executed as browser-external programs, plug-ins can be designed to gain access to the display area inside the browser. Under these circumstances, browsers will act as mere gateways that communicate data to and from server computers. This technique has been successfully used to provide a smooth integration of third-party applications into the Web.

Unfortunately, helpers and plug-ins also have drawbacks. Problems related to such systems include their computer-platform dependency and the need of explicit installation. On typical scenarios, users will need to obtain, install and register specific helper or plug-in applications in accordance to the operating system in use. In practice, this approach will pose an added burden that may discourage the somewhat casual exploration that has given popularity to the Web.

1.4.3.3 DOWNLOADABLE CODE.

Downloadable code is a recent development. The idea behind this technique is to allow client browsers to automatically download and execute small programs (frequently called “applets”) that have been hyperlinked or embedded into HTML documents. This mechanism simplifies the distribution and use of software, by allowing automatic installation and configuration of applications.

In general, any executable code needs to be related to an HTML document for integration to the Web. This integration is accomplished by using any of two methods: hyperlinking or embedding. Hyperlinking consists on creating a hyperlink inside an HTML document pointing to an external file containing executable code. In contrast, embedded code is achieved by using scripting programs, which are commonly written using browser

proprietary scripting languages. These programs are then inserted as part of the content of HTML documents. Examples of embeddable programming languages are JavaScript (Netscape, 1996) and VBScript (Microsoft, 1996a).

However, the possibility of executing programs downloaded from external sources poses security risks. For years, computer hackers have taken advantage of the weaknesses found in programs and operating systems to spread malicious code without users' consent and awareness. If security mechanisms were not implemented, automatic downloadable code would allow even the most naive programmer to produce a highly devastating program. This makes it essential to define parameters and rules to delimit the behavior of downloadable programs without decreasing their ability to produce useful results. In other words, security needs to be enforced to protect the integrity of users' resources without reducing programs' functionality.

Besides security and functionality, a third issue to address is portability, or the ability for programs to be executed unmodified on different operating systems.

As it will be later detailed, portability, security and functionality play a significant role on the reliability of different programming languages and environments that support downloadable code on the Web. One of such languages, which is the subject of study in this research, is the Java programming language. Interpreters for this language have been included in major commercial browsers as a solution to support executable content on the Web.

1.4.4 THE JAVA PROGRAMMING LANGUAGE.

Java was first developed in 1992 at Sun Microsystems as a programming language for consumer electronics software (Naughton, 1996). As time passed, several attempts to commercialize Java, back then called *Oak*, ended unsuccessfully. It was not until 1994, after the Web took by storm the Internet, that it was realized Java has potential to provide embeddable executable content on Web documents. At that point, Sun developed the *HotJava* (Sun, 1996a) browser, which first allowed the execution of Java applets.

However, it was the support given by major commercial companies such Netscape, which implemented Java as an integrated plug-in on its Navigator browser (Netscape, 1996b), that boosted the acceptance of Java among Web users.

Java has characteristics that highly resemble features found in the C (Kernighan and Ritchie, 1978), C++ (Stroustrup, 1991), and ObjectiveC (Cox, 1986) programming languages. Java has been portrayed as an alternative to C++ to create safe, portable, multi-threaded applications for distributed environments.

1.5 RESEARCH OBJECTIVES.

The objectives intended for the present research are:

1. To survey the requirements to develop programs on the Internet and the World Wide Web, as well as the state of the art of programming languages that can be used to implement such programs.
2. To analyze the features of Java as a programming language for the Internet and the World Wide Web.
3. To analyze the implementation requirements for a concept mapping tool to operate on the Internet and the World Wide Web based on previous developments at the Knowledge Science Institute.
4. To design and develop a well-structured implementation of a Java concept mapping tool based on an existing system constructed using the C++ programming language.
5. To compare the Java and C++ programming languages in the light of the implementation experience.
6. To evaluate the Java concept mapping tool in a range of practical applications.
7. To propose further development and research based on the experience and evaluation of the concept mapping tool implemented.

1.6 THESIS OVERVIEW.

This first chapter has been devoted to describe the relevance of the Web as a hypermedia environment and the transcendence of executable content as part of such environment. Chapter 2 will discuss current approaches that integrate executable code to the Web, detailing important issues such as security, portability and functionality. Chapter 3 will show Java in detail: motivations that brought it to life, a summary of its most relevant features, and a description of its system architecture. The differences between Java and C++ at the language level are also explored within this chapter. Chapter 4 introduces the background required for the implementation of the Java concept mapping application. This chapter presents previous work in that area and describes the operations available on the Java concept mapping system when working as a standalone application or as an applet. Such programs are called jKSImapper and jKSImapplet, respectively. Chapter 5 goes deeper in the implementation of these systems and explains the inner workings of both programs, the class hierarchy used for their development, and their runtime system architecture. An interesting addition to this chapter is the description of lessons learned while porting the class hierarchy from C++ to Java. Chapter 6 is devoted to propose further developments and research to improve these concept mapping systems. This closing chapter includes the summary and conclusion for the present thesis.

CHAPTER 2

DOWNLOADABLE CODE ON THE WORLD WIDE WEB

The Internet was designed to integrate any computer platform implementing a set of standard communication protocols, of which TCP/IP is the best known. Without a doubt, the Internet has succeeded in supporting the exchange of data among diverse computer and network architectures. The arrival of the Web marked the beginning of a new era of data access. The Web integrates information into a distributed hypermedia environment as a single consistent interface. In summary, while the Internet established the basis for computer inter-communication, the Web set up the foundations for distributed data integration.

On the Web, browsers are used to access hypermedia information available on the Internet. In the case of downloadable programs, this simple model requires a major feature: platform neutrality.

Platform neutrality addresses the need for relaxed computer constraints to access data. This means that users on a platform neutral environment will be able to access information regardless of the platform used. Additionally, such information will be presented in a consistent format on all available computer platforms. For example, a user on a Macintosh will be able to access information located on a Windows NT server, and this information will be the same as if it was displayed on a PC or a Unix computer.

Platform neutrality encompasses multiple data formats, executable code being one of them. Hence, platform neutrality also leads to the need for portable code. Software distribution over the Web confronts problems different from those faced by traditional distribution models, as explained as follows.

Under a traditional software distribution model, as shown on Figure 2, users freely execute programs bought from software companies, obtained from retailers or downloaded from the Internet. Fortunately, users are not naive about security risks, and most of them exercise (sometimes without consciously noticing it) a selection process for the software they want to use. For example, they may choose software that clearly proclaims their origin, since they will probably trust a piece of software if it comes from a company they are likely to trust; or they may use anti-virus protection programs when dealing with software that may have been tampered with. Unfortunately, this process is not adequate for the Web, since users may not be keen on verifying each program they download for each document that is accessed while navigating the Web.

Proposed models for software distribution on the Web implement a different approach for selectivity and protection. This approach is based on techniques such as digital signatures, code verification and encryption, which are complementary and not mutually exclusive.

Digital signatures are used to identify the author of archives. This identification is used to filter non-trusted sources, and to assign permission privileges according to the level of trustability granted to the provider of the code.

Code verification algorithms can be used to verify that the downloaded code will perform according to the privileges granted by the user.

Additionally, downloadable code can also be encrypted by the author and decrypted by the client, giving the assurance that the code provided has not been modified since its production. Algorithms based on private and public key encryption, such as the “Pretty Good Privacy” (Zimmermann, 1995), can be a viable option in this area. Figure 2 illustrates how these techniques will enhance current models of software distribution, in this case for the Java programming language (Shoffner and Hughes, 1996).

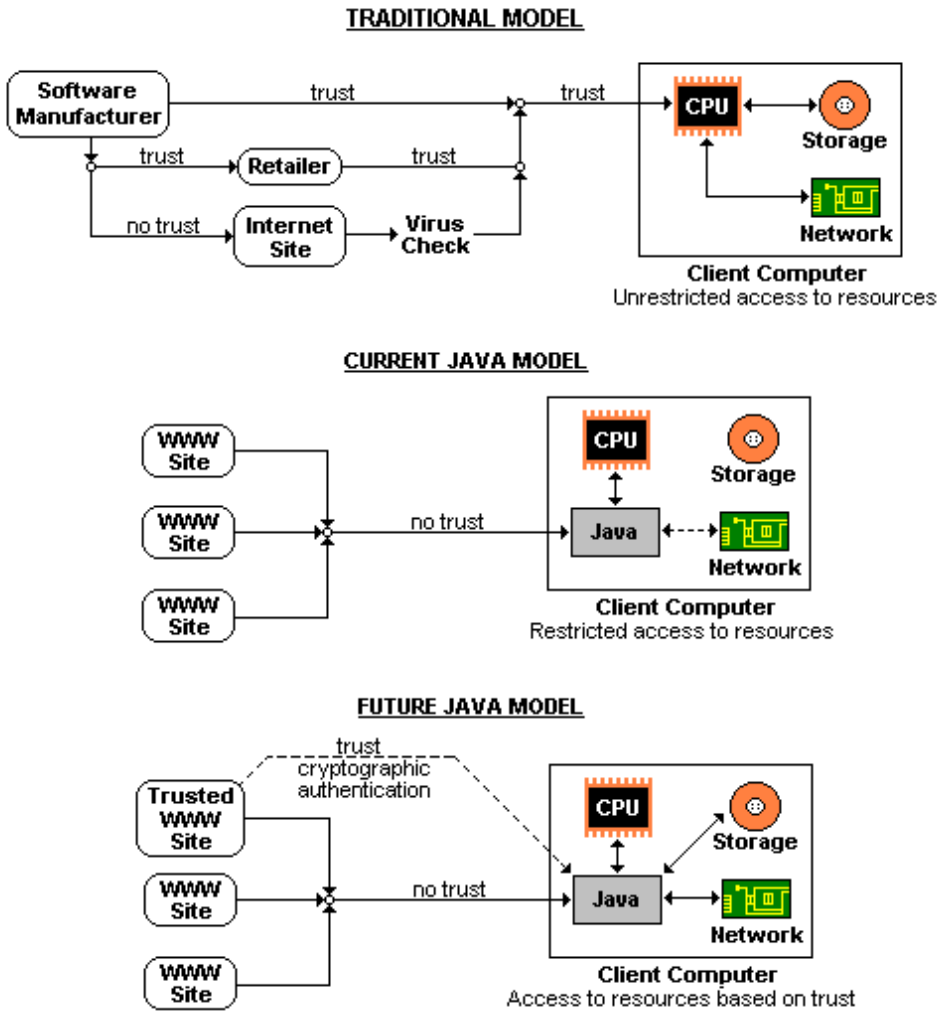


Figure 2. Models for Software distribution.

Even when secure programs can be delivered using these techniques, executable programs still remain tied to the requirements of specific operating system implementations. This limitation has fostered the need to develop approaches to integrate portable and functional downloadable code as part of the World Wide Web.

2.1 Issues for Downloadable Code.

Portability, security and functionality play an important role on distributed systems. These issues are described on detail on the following sections.

2.1.1 PORTABILITY.

Code portability is the characteristic of executing programs regardless of the computer platform on which the code was implemented. Executable code can be produced using diverse programming languages. However, portable executable code is a characteristic that is not widely supported by most languages. According to the level of portability supported, executable code falls into the following classes:

- ***Native Code***: Native code, also known as platform-dependent code, is the result of compiling source code directly to a set of optimized instructions targeted to a specific operating system. This process results on highly efficient code that is virtually non-portable to dissimilar platforms.
- ***Interpreted Code***: Interpreted code comprises options that range from not performing any compilation (using a *scripting language*) to compiling source code to a set of binary instructions (known as *intermediate code*). Because interpreted code is represented in a neutral form, an interpreter will be required at the client site to transform the incoming code to native executable instructions.

Current techniques that support portability range from the ones providing interpreted code that can be run on any platform implementing an interpreter, to the ones having native-compiled versions of a program for each possible client operating system that might request it.

2.1.2 SECURITY.

Topics on this area address the importance of protecting the integrity of user resources and information available to executable code. Security mechanisms and policies will be addressed under the following areas:

- ***Authentication:*** This area covers the need to define the identity of the code and its origin. Mechanisms will be required to identify the supplier of the code and to guarantee that the downloaded code truly comes from the specified supplier. Up to this point,

assumptions can just be made based on the trustworthiness of the originator of the code, but not about the actions that the code might do upon execution.

- ***Authorization:*** During this stage, policies will be required to determine code's privileges to access resources on the client computer. Mechanisms might be used to assert that the code instructions conform to the privileges granted to it. Such mechanisms are relevant to infer code behaviour, but are often inadequate if used on applications derived from programming languages supporting self modification of code at runtime.

- ***Data Integrity:*** Protecting the quality and state of information maintained by the user is one of the most sensitive areas to address. Under this topic, policies and mechanisms are implemented to guarantee that existent information will not be altered by any means or disclosed to unauthorized sources, without user's consent and awareness.

- ***Resource Integrity:*** This area will ensure that client resources will be protected from misuse or abuse in any way that might compromise the integrity of the system. Implemented mechanisms to control executable code will regulate the usage of available resources, including denial of access to non-authorized services. Submission of non-authorized network messages and exhaustion of dynamic memory are some of the hazards that should be prevented.

2.1.3 FUNCTIONALITY.

This issue addresses the capacity of a programming language to produce code that will fulfill user expectations. The potential of a programming language to carry on tasks is reflected through the topics evaluated under the following four major areas:

- ***Data Processing*** comprises characteristics such as object and runtime models, data type implementation, multi-threading support, error recovering mechanisms and memory handling policies.

- ***User Interface*** encompasses the availability of functions to create and manipulate

common graphical user components.

- ***File System*** evaluates the ability to access file structures and their content. Printing capabilities might be an additional topic to include on this area.
- ***Networking*** analyzes the availability of functions to build communicating infrastructures between local and remote computers. Common protocols and facilities to support are sockets, datagrams and HTTP connections.

2.2 INTEGRATION OF DOWNLOADABLE CODE TO THE WORLD WIDE WEB.

Two are the current approaches to integrate downloadable code to the Web: using interpreted programming languages, and using multiple platform-specific programs that can be selectively downloaded according of the operating system of execution.

Microsoft, an industry leader in software for personal computers, is promoting the implementation of the OLE/COM standard, now renamed *ActiveX*, as a standard for components' execution on hypermedia Web documents. However, steering OLE/COM into the Web implies discrimination of computer operating systems, since most of its legacy software is definitively platform dependent. Up to the moment of writing this thesis, Microsoft has clamed that ActiveX is aimed to be an open standard, but very few practical facts have been shown to support such an assertion. While this open standard becomes a reality, Microsoft is promoting a proprietary information file that can be used by browsers to match executable code with the platform of the client computer. This process will result on the selection of the correct code to download and execute (Kirtland, 1996).

In the case of interpreted code, the preferred approach for embedding executable code in Web documents has been by developing browsers with a built-in interpreter to execute downloaded programs. By using this approach, browsers have helped to support the integration of different programming languages into the Web. This is the case of the Python (van Rossum, 1996a), Tcl/Tk (Ousterhout, 1994) and Java programming languages. These languages are supported by various browsers, such as Grail (van

Rossum, 1996b) for Python, SurfIt! (Ball, 1996) in the case of the second, and HotJava, Navigator (Netscape, 1996b) and Internet Explorer (Microsoft, 1996b) for the last one mentioned. All the aforesaid languages have their advantages and disadvantages to implement portable downloadable code in the Web. Due to the aims planned for this research, subsequent analysis on interpreted code will only refer to the Java programming language.

2.2.1 ACTIVEX.

ActiveX can be basically described as Microsoft's OLE/COM technology applied to the Web. There are two required elements for this technology: OLE containers and OLE controls. If applied to the Web, browsers will need to act as containers that embed other OLE controls and containers. In the case of the Internet Explorer, providing this functionality does not represent a problem, since it has been designed as an OLE container from scratch. From the perspective of other browsers, OLE functionality needs to be provided by an external OLE plug-in. In the case of Netscape, this functionality is achieved by using the NCompass (NCompass, 1996) plug-in.

2.2.1.1 PORTABILITY.

Today, ActiveX controls can be produced using any programming language supporting OLE, including C++ and Java –which has been extended by Microsoft to allow the creation of ActiveX components. Unfortunately, these languages require non-portable extensions that restrict their present use to the MS-Windows operating system. To overcome such limitation, efforts have been undertaken to implement OLE/COM on the Macintosh and Unix platforms. If these efforts succeed, they will allow ActiveX components to be available on these platforms as well. However, porting OLE/COM to diverse operating systems does not guarantee automatic portability of ActiveX components, since they will still remain dependent of the platform of development (unless a portable programming language is used to create them).

One solution to support non-portable ActiveX components over multiple platforms is having a different version of the component for each operating system to support. In practice, browsers will need to identify those versions and download the one matching the client platform. This approach has obvious drawbacks, since multiple versions for each control have to be developed and maintained. Nonetheless, this option is the only one available for non-portable ActiveX components.

On a typical scenario, ActiveX controls can be found in three different file formats, prior to their downloading (Microsoft, 1996c):

- ***As a self contained file*:** ActiveX controls can be embedded into portable executable files. Files of this type will carry the extension *.OCX*, *.DLL*, or *.EXE*. This is the simplest way to package an ActiveX control.
- ***As part of a cabinet (.CAB) file*:** Files of this format can hold one or more files needed to execute an ActiveX control. Exactly one archive in the cabinet file is an *.INF* file providing further installation information. This *.INF* file may refer to files in the cabinet as well to files at other URLs.
- ***Referenced by a setup (.INF) file*:** This file type can contain information to specify the location of various files that need to be downloaded to execute an ActiveX control. The syntax of this file allows OLE/COM browsers to identify URLs pointing to ActiveX files to download according to the client operating system (since this file can contain entries for various operating systems). In short, files of this type will act as centralized reference files to let clients know which files to download according to their computer platform.

Apparently, Microsoft is promoting ActiveX as an Internet standard, based more on commercial pressures derived from the legacy software and investments on OLE/COM components, and not because of its portability features. At this point, Microsoft is using Java as a programming language to produce portable controls within ActiveX, and not as a global solution to portability.

2.2.1.2 SECURITY.

Microsoft has based ActiveX security policies on digital signatures and encryption. Since ActiveX controls can be, and most of them are, components made of native code, no mechanisms can be used as code verifiers. In other words, once a component is running, it may access any resource available on the client computer without restrictions.

ActiveX mechanism to inspect and validate signatures is known as the Windows Trust Verification service (Microsoft, 1996d). This service looks at the certificates of approval, within a digital signature, and tells if it comes from a trusted source. Certificates are managed as a hierarchy tree; once a trust point on the tree is found, it is reliable to trust that all the certificates below it are trustworthy. The Windows Trust Verification service is part of the Internet Component Download service, which is the mechanism used to download, certify, and install ActiveX components.

In a typical scenario, ActiveX components are located and downloaded using one of the file formats described on the portability section (portable executable, *.CAB* or *.INF* files). Once the required files have been downloaded, the Windows Trust Verification service will read the signature from a signature block located in these files. Signature blocks contain information about the author of the code, a public key, and an encrypted digest of the file's content. If the certificate provided is marked on the local certificates' database as trustworthy, the trust verification service will decrypt the file's digest and will compare it with a digest created on-the-fly using the same algorithm. This method will determine whether that file's content has been tempered with or not. If these checks are successful, and if the user approves the action, the required files will be installed in the local computer and will request self-registration to a usage module upon execution. This module keeps record of installed controls and their versions, and will be used to determine whether a component has become outdated or not. Further requests for a component will result in the execution of the installed code instead of its repetitive downloading. This behavior will represent an advantage, if it is considered that the size of controls do not tend to be small (between 50 and 200 Kbytes on average) and common

Web clients use slow connections (14.4 Kbps nowadays). These circumstances would make the time required for repetitive downloads very tedious. However, the outdating of an installed component will result on the re-invoking of the Internet Component Download service to download and install the new version.

As seen, the use of digital signatures offers several benefits, but they pose some challenges and drawbacks, as well. The first issue to address is the capability of ActiveX controls to be invoked and initialized by non-trusted sources. Digital signatures provide authentication, which is essential to guarantee that controls were made by honest sources. However, even when a component is trusted as non-intrinsically dangerous, the circumstances on which it is invoked and used may not be as trusting. For example, let us suppose that an ActiveX control is capable of accessing the local file system on the computer of execution. Because the control is provided by a perfectly trustworthy source, the Internet Component Download service will not delay its installation. However, it is possible -and valid- for the same control to be referenced by any other HTML document later accessed by the user, which may initialize and manipulate the control in a way that would risk the integrity of the system (e.g., reading data files, deleting or modifying file structures). One approach to limit components' invocation and initialization is by restricting the data that the control can receive upon initialization; an action that will guarantee the absence of side effects, but also, may restrict the functionality of the control. Another approach is provided by implementing certain API functions that will externalize the different levels of safety scripting allowed by a component. Unfortunately, these approaches are beyond the control of users and rely entirely on developers' willingness to follow them.

A second issue arises due to the need of a central organizer for digital certificates. If digital signatures are to succeed, the first step to accomplish will be the establishment of a centralized organization that will act as a certificate authority for granting certificates to applicants who meet certain registration criteria. This kind of bureaucracy will visibly restrict most of the casual programming (e.g., shareware) found on the Internet. This is

specially true if it is required that programmers from all around the globe submit applications for registration, which presumably will not come without a member's fee.

Digital signatures offer a great service to authenticate the origin of code. However, an endless number of security gaps have to be filled if authentication is intended as a unique mechanism for safe downloading and execution of code. This circumstance may transform ActiveX into a hard-to-manage group of patches and APIs to cover multiple security scenarios.

2.2.1.3 FUNCTIONALITY.

In the case of ActiveX components, functionality is measured according to the programming language used to develop them. In other words, ActiveX components will have as much functionality as their programming language of origin. This circumstance is due to the fact that components will have no constraints once their digital certificates have been checked as trustworthy and the controls have been installed on client computers.

Most ActiveX controls are developed using C++. This language has facilities to allow unrestricted access to internal structures on the operating system of implementation. This circumstance gives developers a good start to implement libraries that support networking device access or multi-threading, which are not implemented as part of the C++ language. Latter versions of the language have implemented an error recovering mechanism based on exceptions.

C++ does not implement any automatic model for memory management. This characteristic provides freedom to manipulate memory allocations and de-allocations according to the best interest of developers. Unfortunately, it also provides a major source of program errors.

Referring to user interfaces, ActiveX controls have been designed to fully exploit the Windows Graphical User Interface. Up to the present time, no data is available to assure that this behavior will be paralleled in OLE/COM implementations other than Windows.

2.2.2 JAVA.

Java is a language designed for programming in distributed environments. If the Internet and the Web are to change the way we think about computers and information, Java will help to the shift by supporting software that can be taken from random sites and be executed on any computer platform. This characteristic helps to support document-centric systems, where people will be able to send information to one another, including spreadsheets, word processors or any other imaginable application that they might need to work with. However, producing secure and portable executable code is one thing, and incorporating it to the Web is another. In the case of Java, Netscape and Microsoft have helped its integration by implementing built-in interpreters inside their widely available browsers.

2.2.2.1 PORTABILITY.

If positioned on the scale between scripted languages (such as Python and Tcl/Tk) and native code, Java stands somewhere in the middle of the scale. In other words, Java source code needs to be compiled, yet the outcome of such compilation is not native code, but intermediate code instead. When Java programs are compiled, each class in the source code is transformed to a class file composed of an intermediate set of instructions called *bytecode* (Sun, 1995a). Bytecode consists of binary operands targeted to a simulated CPU known as the Java Virtual Machine (JVM), which is implemented as part of Java interpreters. The JVM can be seen as an abstract processor that encompasses the operations found on most computer platforms. This characteristic allows a high degree of portability with a relatively low penalty on performance.

2.2.2.2 SECURITY.

Unlike the sole use of digital signatures in ActiveX, where users blindly trust code by their origin, Java bases its security strategy on a radically different premise: trust no one’s code. Java closely checks each instruction prior to execution to guarantee that it will conform with authorization levels specified by the user.

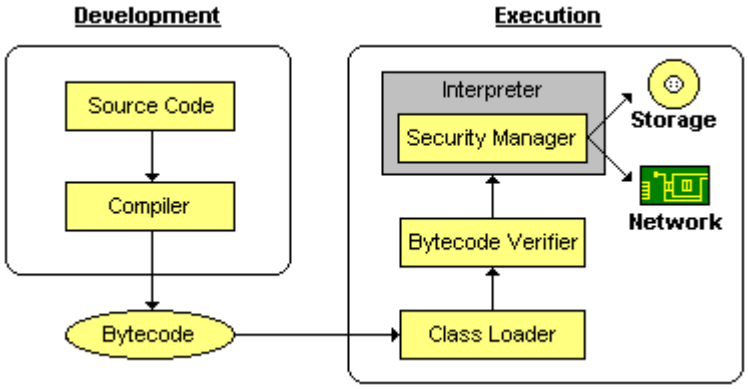


Figure 3. Java Security Model.

As shown on Figure 3, the Java language encompasses four different security stages, starting from the development of applications until their execution. These stages -- or layers as they are also called -- are described in the following sections (Sun, 1995b).

2.2.2.2.1 THE LANGUAGE AND COMPILER.

Under Java, programs are restricted from directly accessing dynamic memory. In practice, all objects used in Java programs are accessed through handles, which are address-independent identifiers used for the manipulation of objects in the heap. Handles help to isolate programmers from specific memory representations (thus enhancing portability), eliminate accidental or malicious accesses to memory locations not part of the application (therefore maintaining the integrity of the system), and helps to the implementation of the garbage collection process (which disposes of any non-referenced object in the heap).

Handles are controlled by an automatic memory manager. The tasks of a memory manager include to keep track of all objects in the heap and to automatically free any memory space occupied by objects no longer referenced in a program during execution (a process that is known as *garbage collection*). This combination of handles and the automatic memory contributes to safe execution of Java code. For a description of the benefits of handles and garbage collection, refer to Chapter 5, Section “5.3.1. Automatic Memory Management”.

Additional restrictions enforced by the compiler are related to casting (to check that all references to methods and variables are used on objects of the appropriate type), conversions (e.g., integers can not be converted to objects, or vice versa) and access to non-initialized variables. These restrictions are found on most typed languages.

2.2.2.2.2 The Class Loader.

The class loader will be responsible for storing incoming code into separate execution environments known as *namespaces*. The class loader ensures that all class files from the same origin reside in a single namespace, thus isolating class archives from different sources. Class files, already stored on the client's computer, will occupy their own namespace and they will have precedence over other class files with the same name, located on others namespaces. This constraint will guarantee that all local classes (e.g., system classes) are protected from being replaced or extended by imported code.

2.2.2.2.3 The Bytecode Verifier.

Security goes further than just forbidding access to dynamic memory. As previously mentioned, Java source code is compiled to class files made of intermediate bytecode. When class files are downloaded, a process is triggered to check that the bytecode has not been forged to deliver malicious behavior. Such process, which is accomplished by the bytecode verifier, will perform several checks on the downloaded class files to guarantee their conformance to the runtime specifications. Verified Java programs will adhere to the following constrains, prior to their execution (Yellin, 1995):

- All register accesses and stores are found to be valid,
- The parameters of all bytecode instructions are correct, and
- There will not be illegal data conversions.

Another important characteristic of the bytecode verification process is that it enhances the performance of the interpreter by doing certain checks that the interpreter will not

need to duplicate, resulting on a slight improvement of code performance during execution.

2.2.2.2.4 SECURITY MANAGER.

The security stages described above will ensure that the downloaded code is well formed, and that will run according to the specifications of the Java Virtual Machine. Up to this point, one can be sure that downloaded bytecode is valid bytecode, but one knows nothing about its intentions. Downloaded programs may be programmed to access files, open network connections, read system properties and execute sub-processes under user privileges, and few users may want downloaded code to have such a freedom. Fortunately, many of the programs' operations can be controlled by parameters put in a properties file located on the client computer. Such parameters will define the behavior enforced by the security manager.

According to each different implementation of the interpreter they may or may not observe security parameters on the parameters file. The behavior followed by the security manager implemented on different Java interpreters is shown in Table 2 (Sun, 1996b). This table indicates circumstances on which a security manager will follow or overlook user parameters.

The principal properties shown in this table are *acl.read* and *acl.write*, which are used to maintain a list of directories and archives that can be accessed by Java programs.

The compliance to these parameters vary depending on the type of Java program implemented, the location of the code and even the interpreter applied for execution. For example, in the fourth operation (to write a file to the "*/tmp*" directory), the parameter *acl.write* enables the "*/tmp/*" directory for reading, thus allowing the completion of the operation. However, Netscape does not allow to write files to the directory specified (thus, ignoring to observe this parameter), while the AppletViewer and command line interpreters do comply with the permission granted.

	Java program type		Applet				Application
	Java interpreters		Netscape		AppletViewer		Command line
	Location of class files		Network	Local	Network	Local	Local
Operations	**Access parameters**						
	acl.read	*acl.write*					
read file in */home/me*	*null*		no ✓	no ✓	no ✓	yes ×	yes ×
read file in */home/me*	*/home/*		no ×	no ×	yes ✓	yes ✓	yes ✓
write file in */tmp*		*null*	no ✓	no ✓	no ✓	yes ×	yes ×
write file in */tmp*		*/tmp/*	no ×	no ×	yes ✓	yes ✓	yes ✓
get file info	*null*	*null*	no ✓	no ✓	no ✓	yes ×	yes ×
get file info	*/home/*	*/tmp/*	no ×	no ×	yes ✓	yes ✓	yes ✓
delete file using *File.delete()*			no	no	no	no	yes
delete file using *exec /usr/bin/rm*			no	no	no	yes	yes
read the *user.name* property			no	yes	no	yes	yes
connect to a networking port on the client computer			no	yes	no	yes	yes
connect to a networking port on a 3rd host			no	yes	no	yes	yes
load a library			no	yes	no	yes	yes
quit the interpreter			no	no	no	yes	yes
display a window without a warning			no	yes	no	yes	yes

✓× compliant/non-compliant with the user-defined parameter.

Table 2. Access parameter compliance of prevalent Java interpreters.

It can also be seen from Table 2, that Netscape takes the more radical approach by restricting all the operations shown. On the other hand, standalone applications executed by the command line interpreter can operate without restrictions, but unfortunately, they overlook user-defined parameters. Amid these interpreters stands the AppletViewer,

Sun's applet interpreter included on the Java Development Kit (Sun, 1996c). This interpreter encompasses a perfect ratio of compliance (as seen by the number of ✓ marks) with the specified user parameters when executing code downloaded from the network.

User-defined parameters provide a satisfactory approach for secure execution of downloaded code. However, the actual implementation of Java (JDK version 1.0.2, at the time of the writing of this thesis) fails to include a cohesive infrastructure for organizing priorities according to levels of trustworthiness (e.g., the origin of the code). Without a doubt, more effort will be needed to develop a comprehensive parameter-based security model in Java that can effectively control applets and applications according to user preferences.

2.2.2.3 FUNCTIONALITY.

Java provides a high degree of functionality without sacrificing portability. Java is considered to be an extended subset of C++. It provides a pure object-oriented environment, with single class inheritance and multiple inheritance on interfaces, arrays, exception handling, automatic memory management, and multithreading. However, it does not implement structures, pointers, unions, enumerated types, bit fields, definition of types, operator overloading or templates, which are found in C++ (for a comprehensive list of differences between Java and C++, please refer to Chapter 3).

From the libraries included with Java, six of them encompass the main attributes of the language:

- **applet:** This small library defines a base class for applets.
- **awt:** This library, which name is the acronym for *Abstract Windowing Toolkit*, encompasses all the base classes for graphical elements.
- **io:** This library includes classes for handling streams, files and I/O in general.
- **lang:** This library contains the pivotal classes of the Java language. This library includes the *Object* class (from where all classes are derived) and the *Throwable* class, which is

the root class for exceptions. Also included in the package are class wrappers for all fundamental data types, classes for string and thread manipulation, and a class for system access.

- **net:** This library comprises all classes necessary for networking access, such as datagrams, sockets and URL connections.
- **util:** This library defines a small number of classes frequently used on programs, such as *Date*, *Vector*, *Hashtable* and *StringTokenizer* classes.

From the above libraries, *awt*, *io* and *net* can be considered the most relevant due to their ability to easily achieve their functionality (graphical user interfacing, file management and networking, respectively) on a portable manner. However, they still need some refinements to become fully functional:

- Networking connections and local file accesses still cannot be used flawlessly due to the absence of a consistent security mechanism for handling access requests to resources.
- Graphical user components have been found difficult to implement and manage, and frequently exhibit distinct behavior when executed on different windowing environments.

In conclusion, these libraries will require more revisions and adjustments before they may be considered of similar quality to those implemented by ActiveX components.

Since Java does not implement templates, it lacks a type-safe generic container library like the Standard Template Library (Stepanov and Lee, 1995). Nonetheless, third-party implementations, such as the Java Generic Library (ObjectSpace, 1996), help to fill the gap. This library provides classes to handle sequences, maps sets, queues and stacks, and algorithms to sort, swap, filter, compare, access and modify objectified elements in containers. Unfortunately, the advantage of strong typing, as implemented in C++ templates, is not present on Java.

2.3 CHAPTER SUMMARY.

This chapter has covered popular approaches to integrate downloadable code to the Web, as well as issues concerning software distribution models and the automatic execution of code.

Security and portability were described as necessary issues to protect user resources and provide transparent execution of programs among dissimilar operating systems. It was also discussed that the implementation of these techniques has a repercussion on the ability of code to properly achieve users' goals, also known as functionality.

The last part of this chapter concludes describing ActiveX and Java as predominant approaches to provide downloadable code to the Web. A comparison between these approaches was made by comparing techniques included on each implementation that affect the issues of portability, security and functionality. From such comparison it can be inferred that none of these approaches is an absolute winner over the other, since each one has their own characteristics suitable for different circumstances. Developers may prefer Java if they prize portability above all, and they may prefer ActiveX if fully exploitation of the Windows environment is highly valued.

The work on this thesis has been aimed to provide a portable implementation of a concept mapping tool to the Web, selecting Java as the language of implementation. As a result, the next chapter will be devoted to describe in-depth characteristics of the Java programming language and how it differs from C++. Subsequent chapters will detail the implementation and evaluation of the concept mapping tool development mentioned earlier.

CHAPTER 3

THE JAVA PROGRAMMING LANGUAGE

In 1985, researchers at Sun Microsystems were working on an innovative windowing system called NeWS, which stands for Networked Extensible Window System. This system was implemented as a distributed system with client computers running lightweight processes that communicate with server applications using messages (Gosling, Rosenthal and Arden, 1989). Typically, client processes perform as receivers of user events that are translated into commands and transmitted to server processes. In response, server processes return programs for the client to execute. These programs can perform operations on the display and receive events from the keyboard and the mouse, thus, repeating the interaction process. Programs downloaded from servers are created to use the PostScript programming language (Adobe, 1985), which is an interpreted language capable to provide portability of code between different computers and operating systems.

NeWS never caught enough market share to succeed and the project was canceled at the beginning of the 1990's, resulting on the merging of team members into other projects. One of these projects, which had the mandate to develop software for consumer electronics, was the originator of the Java language. When C++ proved not to be suitable for the task assigned to that project, the Java programming language was created. Even if NeWS is not strictly a predecessor of Java, the experience gained from the development of NeWS may have helped to shape the features of the Java programming language.

3.1 OVERVIEW.

A programming language is usually characterized by its main features. Java is depicted as an object-oriented, distributed, secure, multi-threaded and portable programming language. These characteristics are detailed on the following sections.

3.1.1 OBJECT-ORIENTED.

Java is an object-oriented programming language. For programmers, this means that they will need to focus on the application data and on the methods needed to manipulate that data, rather than concentrating on functions and procedures. Java was designed with features found on previously developed object oriented languages. However, it manages to strike a balance between pure object oriented models, such as SmallTalk (Goldberg and Robson, 1989), and non-object oriented models, such as C.

Booch (1994) described several features required for a programming language to be labeled as object oriented:

- ***Abstraction*****:** "*An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus, provide crispy defined conceptual boundaries, relative to the perspective of the viewer*." (Pg. 41)

The parameters relevant to this feature are the availability of instance variables and methods, and class variables and methods. The main difference between these two groups is that instance variables and methods can only be used through an instance object for the class, while class variables and methods can be used directly by specifying the qualifier of the class. Additionally, the values of instance variables are exclusive to each instance, while the value assigned to a class variable is shared by all the instances of that class. All of these features are supported by the Java language.

- ***Encapsulation*****: "***Encapsulation is the process of hiding all the details of an object that do not contribute to its essential characteristics.*" (Pg. 50)

In the case of Java, several levels of hiding can be used for variables and methods. These levels of encapsulation are linked to the following modifiers:

- **public:** no access restrictions.
- **private:** access is granted to invocations from inside the class.
- **protected:** access is allowed to invocations from within the class, from other classes belonging to the same package, and from subclasses of the declaring class.
- **private protected:** access is forbidden to invocations that do not belong to the class of declaration or its subclasses.

If no reserved word is used, then the access is granted to invocations made from within the class of declaration and from classes belonging to the same package.

- ***Modularity*: "***Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.*" (Pg. 57)

Java uses two levels of modularity: by classes (where each class is a container of variables and methods), and by packages (which is the grouping of classes into logical units).

- ***Hierarchy*: "***Hierarchy is a ranking or ordering of abstractions.*" (Pg. 59)

Java's object model is based on a single-inheritance class hierarchy, having the class *Object* as the root class. This means that all classes have just one immediate parent, and that the class *Object* is a super class for all the classes. Even when the class hierarchy is based on single-inheritance, multiple-inheritance is allowed by the use of interfaces. The idea of interfaces is a concept borrowed from the *protocols* found on ObjectiveC. Interfaces are essentially abstract classes that declare, but do not implement, methods, which are to be implemented on inherited classes. Variables declared on interfaces are handled as class variables for all classes using that interface.

3.1.2 DISTRIBUTED.

Java is a distributed programming language. It supports both the TCP/IP (Transmission Control Protocol/Internet Protocol) and the UDP (User Datagram Protocol) families. From these protocols, TCP/IP is used for reliable stream-based communications, and UDP to support fast point-to-point datagram-oriented models. Java networking classes also include classes to handle Internet addresses and to download the contents of resources associated with a URL.

3.1.3 PORTABLE.

Java has the characteristic of being portable, or more accurately said, programs produced on Java can be executed on any computer where a Java Virtual Machine is implemented. This capability of being portable is based on Java's platform neutrality and interpreted nature.

Java's cornerstone to allow portability is based on a proprietary set of intermediate instructions called bytecode, which are used to conform all Java programs. Bytecode are sequences of bytes representing instructions for the Java Virtual Machine, which is a simulated CPU implemented on Java interpreters. In practice, when an interpreter loads a program, each byte is evaluated in software, performing changes on the state of the virtual CPU to reflect the changing state of execution on the program.

Additional characteristics that support portability in Java are the abstraction of primitive data types and graphical user interfaces. In the case of primitive data types, as shown in Table 3, they are specified to be of a fixed size regardless of the operating system of execution.

Type	Contains	Size	Minimum Value	Maximum Value
boolean	*true* or *false*	1 bit	Not Applicable	Not Applicable
char	Unicode character	16 bits	\u0000	\uFFFF
byte	signed integer	8 bits	-128	127
short	signed integer	16 bits	-32768	32767
int	signed integer	32 bits	-2147483648	2147483647
long	signed integer	64 bits	-9223372036854775808	9223372036854775807
float	IEEE 754 floating-point	32 bits	±3.40282347E+38	±1.40239846E-45
double	IEEE 754 floating-point	64 bits	±1.79769313486231570E+308	±4.94065645841246544E-324

Table 3. Java Primitive Data Types.

To handle user interfaces, Java designers developed an abstract windowing library that acts as a wrapper for native widgets found on major graphical environments. This way, the use of components is unified under a single set of classes, which are independent of the platform of execution. Figure 4 shows the class hierarchy for widgets available on the core release of Java. These components represent just a portion of the entire abstract windowing library.

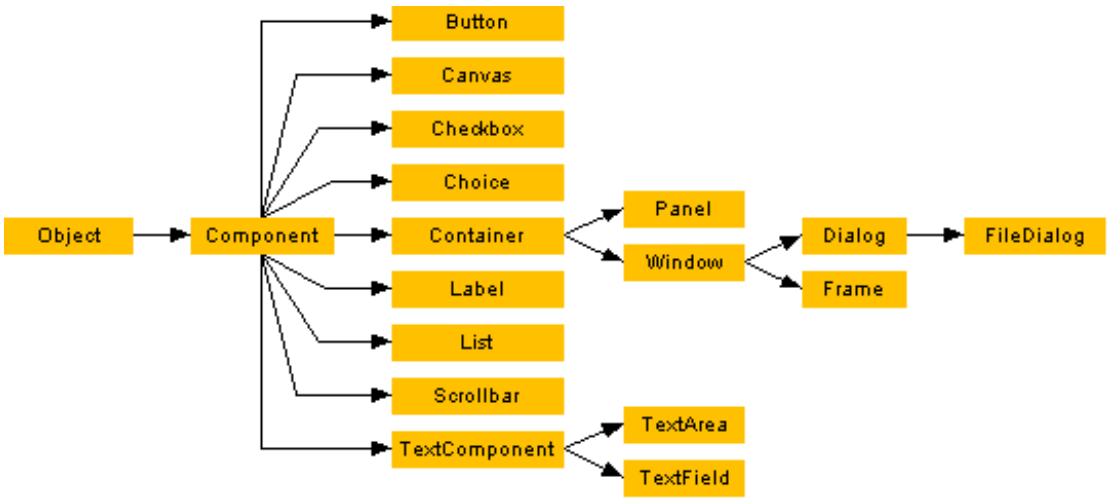


Figure 4. Component classes from the Abstract Window Toolkit library.

Classes found on Java allow the creation of buttons, canvases, checkboxes, radio buttons, labels, list boxes, combo boxes, scroll bars, input lines, input areas, windows, panels, dialogs, windows, and the practical file dialog to select disk files.

3.1.4 SECURE.

Java is intended to be a secure language. Security is an important concern, since Java is targeted to networking environments. Based on the premise that no downloaded program is to be trusted, Java implements several security mechanisms to protect users against malicious code.

When compiled, Java source code is checked for compliance with the memory allocation and reference model. Under this model, declarations for direct access to memory addresses are not allowed. Additionally, memory layout decisions are not made at compilation time. Instead, compilers will generate handles that will be resolved to real memory addresses at runtime, preventing programmers to hack into systems using such addresses.

Even though the use of Java compilers ensures that source code will behave according to safety rules, interpreters do not have the means to check that any downloaded bytecode was produced by a well-behaved compiler. To trust downloaded code, interpreters will subject programs to verification through a series of tests. These tests range from simple verification of the format on instructions to validating the code through a simple theorem verifier. Once the verification process is done, interpreters can proceed to execution knowing that the code will run securely. For detailed information on the verification process please refer to the Java Security section back to Chapter 2.

Unfortunately, the verification process in Java is not as secure as it is claimed, since it fails to have formal semantics and a formal description of the type system. This circumstance makes it impossible to formally prove the correctness of the runtime verifier (Dean, Felten and Wallach, 1996). As a result, the verification process can not be proven correct since its exact behavior for every possible set of bytecode is uncertain.

3.1.5 MULTI-THREADED.

Java is a multi-threaded language. It provides support for multiple lightweight processes within a program. The main problem with writing multi-threaded programs resides on making methods safe to be accessed by multiple concurrent threads. This task usually implies the management of locks to control and synchronize access to resources.

Java supports pre-emptive multi-threading at the language level and through the support of the runtime system and thread objects. Multi-threading is supported at the language level by using locks -- or *monitors* -- for synchronization. Every class and variable has a lock that can be used for this purpose. For example, methods within a class that are declared *synchronized* do not run concurrently. This behavior is automatically enforced by granting the class lock to the first thread entering a synchronized method. The lock will be released by the thread when exiting the method or when put to sleep. Support for threads at the class level is provided by the *Thread* class, which implements methods to start, stop and handle threads, and the *Runnable* interface, which provides the abstraction required for an instances of a class to be treated as a thread.

3.2 PROGRAMMING FOR THE INTERNET AND THE WORLD WIDE WEB.

Rather than creating new HTML extensions, Java made popular the notion of downloadable programs that can run inside Web browsers. The alpha release version of Java, back in 1995, included a Web browser called HotJava. This browser allowed normal Web navigation plus the ability to execute Java applets hyperlinked to HTML documents. Shortly thereafter, Netscape announced its intention to license Java to integrate it to its second version of its market-leading Navigator browser.

HotJava and Netscape Navigator are not the only browsers that support Java applets, but they were first in order of appearance and current market share, respectively. Both these browsers have promoted the use of Java as a programming language for the Web. However, HotJava and Netscape Navigator have followed different patterns of development and, up to the day of writing this thesis, HotJava has only reached beta

release status while Navigator is at the brink of version 4. Due to its wide availability and advanced state of development, Netscape Navigator will be chosen for further studies on the integration of Java to the Web.

In order to evaluate the suitability of Java as a programming language for the Web, two characteristics have to be observed: first, the level of integration with browsers, and second, the availability of tools to perform distributed operations.

3.2.1 INTEGRATION WITH NETSCAPE NAVIGATOR.

Netscape has developed LiveConnect (Netscape, 1996c) to act as an integrator between JavaScript and Java. LiveConnect allows JavaScript code to access variables, methods and classes found on applets embedded on Web documents. Java classes do not need any special settings to be invoked from JavaScript.

```
<SCRIPT>
function handleEvent(id, value1, value2, value3)
{
  document.jKSImapplet.handleJavaScriptEvent(id,
                                             value1,
                                             value2,
                                             value3)
}
</SCRIPT>

<FORM NAME=Options>
  <SELECT NAME=listNEW>
    <OPTION SELECTED> Node - Rectangle
    <OPTION>          Node - Rounded Rectangle
    <OPTION>          Node - Ellipse
    <OPTION>          Line - Binary
    <OPTION>          Line - Trinary
    <OPTION>          Line - Quadrary
    <OPTION>          Context Box
  </SELECT>
  <INPUT TYPE="BUTTON" VALUE="New"
   onClick="handleEvent(0,
                        document.Options.listNEW.selectedIndex,
                        0,
                        null)">
</FORM>
```

Figure 5. Code example for a Java method invocation from JavaScript.

Figure 5 shows a code example where a Java method is invoked from a JavaScript function embedded on an HTML document. In this example, the method

handleJavaScriptEvent, which belongs to the *jKSImapplet* Java class declared on the current document, is invoked after a button (part of an HTML form) is pressed to process choices made on a selection box. As seen from the function *handleEvent*, the invocation of Java methods just requires the definition of a hierarchical path for locating applet's classes and variables.

However, enabling communication in the other direction (from Java to JavaScript) requires a little more effort. In such case, Java applets will need to subclass objects derived from the *JSObject* and *JSException* classes, which are part of the *javascript* package provided by Netscape. From these classes, the *JSObject* class enables Java applets to access JavaScript methods and properties, and *JSException* allows the handling of exceptions thrown by JavaScript code returning an error.

Figure 6 shows a code example taken from the LiveConnect introduction documentation, where the *eval* method from the class *JSObject* is used to send an expression to JavaScript each time the mouse button is released. In this example the expression sent is an invocation to the *alert* JavaScript method.

```
Public void init()
{
  JSObject win = JSObject.getWindow(this);
}

public boolean mouseUp(Event e, int x, int y)
{
  win.eval("alert(\"Hello world!\");");
  return true;
}
```

Figure 6. Code Example for a JavaScript method invocation from Java.

3.2.2 INTERNET NETWORKING.

The *java.net* package included as part of Java provides the infrastructure needed to achieve networking operations. The basic protocols to deal with the Internet are implemented in a few classes that encapsulate their functionality without involving the programmer with low-level networking details. Classes included in this package allow

one to represent Internet addresses, to access resources referenced by URLs, to perform low-level networking using datagrams, and to communicate using stream sockets.

Basic classes required for networking operations are *URL* and *InetAddress*. These classes, and their importance to initialize other classes, are explained as follows:

- **URL:** The *URL* class implements Internet Resource Locators. It provides the most basic interface to perform networking operations, since resources referred by a URL can be downloaded using a single method invocation. *URL* is also used to initialize objects of the *URLConnection* class. This class provides additional methods than those provided by the *URL* class to perform complex manipulation of Internet resources. For example, using *URLConnection* objects it is possible to obtain information about the resource pointed, its content type, length, and date of last modification. Additionally, if the protocol used supports write operations, then methods implemented in this class can allow overwriting the content of a resource pointed to by a URL.

- **InetAddress:** This class supports Internet addresses, and is used when performing networking operations using sockets and datagrams. The *InetAddress* class does not have a public constructor method, but supports static factory methods to create new instances. Such instances can contain the address of the local host or the address of a host specified by name. The *InetAddress* class is used to initialize socket and datagram communications, which are explained as follows:

  - ***Datagrams*:** UDP datagrams are fire-and-forget packets of information that are passed over the network. They provide fast communication. The tradeoff is that they are not guaranteed to reach their destination, and if they do, separate datagrams may not even arrive in the order they were sent. However, when optimal performance is required and the overhead of doing custom verification is justified, datagrams are a valuable mechanism to have available. Classes used for datagram communication are *DatagramPacket* (data container class) and *DatagramSocket* (datagram packet sender and receiver class).

- ***Sockets*:** TCP/IP sockets implement reliable bi-directional point-to-point, stream-based connections between hosts on the Internet. A common model for network communication is to have one or more clients sending requests to a single server program. In such cases the server uses an instance of the *ServerSocket* class to accept connections from clients. When a client reaches the port on which the server is listening, the server allocates a new *Socket* object in a new port for subsequent communication between server and client. After allocating the new connection, the server returns to the listen mode for receiving additional client connections.

3.3 LANGUAGE COMPARISON BETWEEN JAVA AND C++.

Java is a language that borrows much of its terminology and syntax from C++. However, Java is considered a simpler language than C++, since a number of C++ features have been removed from the Java implementation. In certain ways, this reduction allows programmers, familiar with C++, to easily climb the learning curve. Java eliminates some C++ redundancies and non-object-oriented characteristics maintained as legacy from C.

A number of main differences exist between Java and C++. These differences, which range from slight modifications to complete removal of features, are described as follows:

- ***No header files*:** Header files are considered of great benefit for data hiding, since they allow one to declare the prototypes for classes in a readable format while having the actual implementation in a binary file for distribution. On the other hand, the existence of header files creates inconveniences such the maintenance required to keep the consistency between header file declarations and the source file implementation. Java has eliminated header files, and it maintains all the information about a class inside the class implementation.

- ***No preprocessor*:** Java does not include any kind of preprocessor. One of the jobs of a preprocessor is to search for special commands that begin with a hash mark "#". These commands perform conditional compilation and macro replacement. It may seem hard for C++ developers to program without *#define* or *#ifdef*, but Java can make do without

these constructs. In the case of *#define*, Java relies on the *final* keyword to achieve some of its functionality. Additionally, the *import* statement has similar characteristics to the *#include* command, and *#ifdef* commands can be partially simulated by using compilers that optimize blocks of code delimited by boolean expressions that have static values (e.g., *if (false)*)

- ***No global functions or global variables*:** In Java, methods and variables are declared within classes. Likewise, every class is part of a package, resulting on all methods and variables to have fully qualified names. These names are formed using the package name, the class name and the variable or method name. By having static variables and methods it is possible to simulate global functions and variables, but it is not possible to have name conflicts due to the naming convention previously described.

- ***No** goto **statement*:** Java does not implement the *goto* statement and thus, it eliminates the main instrument of the so called “spaghetti code." However, the keywords *break* and *continue* cover some important and legitimate uses of *goto* on looping structures. Furthermore, Java’s well-defined exception handling compensates for the absence of this statement.

- ***No operator overloading*:** Method overloading is a technique that allows the declaration of several methods with the same name but with different list arguments. Operator overloading is a similar technique, but it allows symbols to be declared as methods to perform operations according to the type of the parameters involved. Up to the present version, Java allows method overloading but it does not allow operator overloading.

- ***No structures, unions, typedefs, bitfields, enumerated types or variable-length argument lists*:** Java does not support the *struct* and union types found in C++; however, structures can be simulated using classes without methods. Additionally, Java neither supports *typedefs* (to define new aliases for type names) nor *bitfields* (which can be used to interface hardware devices, for example). Java does not allow one to define methods that take a variable number of arguments. Method overloading and arrays can act as replacements for simple cases of variable-length argument lists. The absence of

enumerated types is a missing feature that may be seen as unusual, since Java has the characteristic of being strongly typed. However, this circumstance may be the result of a design decision to maintain simplicity on the types handled by the language.

- ***No** const **parameter qualifier*:** In C++, when a parameter is specified with the *const* qualifier, the compiler makes sure that the value assigned to that variable will remain unaltered during its scope. As a result, methods receiving a variable passed as a *const* parameter are not allowed to make modifications to its value. In Java, there is no automatic mechanism to perform this operation.

- ***No templates*:** C++ templates are type-parameterized classes or functions. Template based class libraries are not just type safe but also enhance reuse of structures for different type formats. Templates are also of relevance on the context of containers. Since object-based (non-template) containers do not have the mechanisms to enforce certain object type for their elements, there is no way to ensure that a container actually holds objects of the expected type.

- ***Characters are Unicode characters*:** In Java, values of type *char* are not signed. Additionally, characters and strings are composed of 16-bit Unicode characters, allowing easy internationalization of programs that do not use the Latin alphabet. The Unicode character set is composed of more that 34,000 distinct code characters, where the first 256 are ASCII compatible.

- ***Arrays and Strings are objects*:** Arrays and strings behave just as regular objects: they are manipulated by reference, they can be dynamically created with *new*, and they are automatically garbage collected when no longer needed. However, they are special in the sense that they can be manipulated differently than objects. As shown on Figure 7, arrays and strings can be directly initialized by specifying their value. In the case of strings, concatenation can be achieved by placing addition symbols between string variables and constants.

```
String subject   = “John Doe”;
String aliases[] = [“Steven Sagan”, “Vitto Corleone”];
String message   = subject + “ is also known as ” + aliases[1];
```

Figure 7. Code example on initialization of strings and arrays.

- *null* ***is a reserved keyword and*** *boolean* ***is a primitive data type*****:** In Java, *null* is a value that indicates an absence of reference. Unlike C++, where *NULL* is just a constant defined to be 0, *null* in Java is a reserved word that has no value and can not be assigned to primitive data types. On the other hand, *boolean* is defined as a primitive data type that can be assigned a value of *true* or *false*. In contrast to C++, *boolean* values are not integers; they can not be treated as integers, and may never be cast to or from any other type.

- ***Primitive data types are fixed in size and sign, and can not be cast to objects, or viceversa*****:** As previously seen in Table 3 (pg. 37), *boolean*, *char*, *byte*, *short*, *int*, *long*, *float* and *double* are primitive data types available in Java. These variables are always fixed in sign and size, unlike C++ where an integer may be 16, 32 or 64 bits, and characters may be signed or unsigned depending on the operating system of execution. Additionally, Java does not allow conversions between primitive data types and object references, as in C++ (e.g., casting an integer to a pointer).

- ***Parameter-passing is always by value*****:** There are two techniques to pass parameters to C++ functions: *call by value* and *call by reference*. When passing variables to a function using “call by value” a copy of the original data is passed. This circumstance allows modifications on the copy without altering the value of the original variable. On the other hand, when passing a variable using “call by reference” an alias is created for the variable itself. This alias represents the memory address where the variable is located. Under this technique, modifications on the variable passed to the function will result on modifications of the original value as well. In the case of Java, variables are always passed to methods by value. For primitive data types, this assertion means that an independent copy of the original value is passed. In the case of handles to objects,

copies of the *handles* are submitted. This circumstance allows the modification of the object referenced by the original handle, and does not allow the modification of the handle (this behavior is achieved in C++ by using the *const* modifier on pointers and references passed to functions). Java has no mechanisms to modify the original value of arguments from within methods, whether it is a handle or a primitive data type. One way to modify the content of a variable when submitted to a method as an argument is by assigning the return value of the method to that variable upon return.

- ***Threads and synchronization are part of the core language*:** As previously explained on the Multi-threaded section early on this chapter, synchronization in Java is an intrinsic part of the language. Synchronization is achieved by the use and enforcement of locks, which prevent multiple threads from simultaneously accessing critical sections of code. The *Thread* class encapsulates all the information about a single thread of control running on the Java interpreter. This type of support for threads and synchronization of threads is a feature that it is not implemented as part of the C++ language.

- ***Automatic memory management*:** Objects in Java are created on the heap using the *new* keyword. However, there is no *delete* keyword to dispose of them, as in C++. This is because Java implements a memory manager to handle all references to the heap and disposes of objects that are not longer referenced or used in a program. The disposing of objects and the freeing of memory is performed using a process called *garbage collection*. The garbage collector process runs on a low-priority thread whenever the system is idling, or when a request for memory allocation fails to find enough free memory to satisfy such request. The concept of automatic memory management is foreign to C++. In C++, programmers have the responsibility to remember when and where to dispose of allocated objects. It is worth mentioning that garbage collection processes will never be as efficient as explicit, well-written memory allocation and deallocation routines written by programmers. However, it does make programming easier and less prone to errors.

- ***Single inheritance on classes, multiple inheritance with interfaces*:** C++ allows classes to have more than one superclass, using a technique known as multiple inheritance. This technique allows class designers to mix various attributes from different branches of a class hierarchy. Java does not implement multiple class inheritance, but implements multiple interface inheritance. Interfaces are just like classes, but they are not allowed either to declare instance variables, or implement methods.

3.4 CHAPTER SUMMARY.

This chapter covered the fundamental aspects of the Java language, which was described as an object-oriented, distributed, portable, secure and multi-threaded programming language. After discussing these characteristics, Java was scrutinized to find its suitability as a programming language for the Web.

This chapter also showed Netscape's LiveConnect environment as a fundamental part to integrate Java with the Navigator browser. The use of JavaScript and Netscape-tailored Java classes were discussed as required elements to achieve this integration.

Additionally, Java networking classes were explained. Classes that handle Internet addresses and Uniform Resource Locators were introduced as the basis to support datagram and socket communications, and URL connections.

This chapter concluded with a section detailing the main differences between Java and C++. In its role as the most popular programming language, C++ is compared with Java with respect to aspects ranging from the structure of primitive data types to templates and automatic memory management.

Chapter 4 is an overview of the Java concept mapping tool implemented as a test case for this research. This chapter will show previous concept mapping tool developments, as well as the motivation underlying such developments. The system architecture for the test case, which has been named jKSImapper, is further discussed in Chapter 5. That chapter will also discuss lessons learned when porting previously developed C++ classes to Java.

CHAPTER 4

IMPLEMENTING A JAVA CONCEPT MAPPING TOOL

The purpose of the systems implemented as test cases for this research is to provide support for distributed interaction of concept maps on the Internet and the Web. To achieve this goal, two primary steps were required: first, to develop a class structure to support the manipulation of concept maps; and second, to extend such a system to allow multi-user elicitation of concept maps on the World Wide Web.

The work required to accomplish these steps was not done under isolated circumstances, but as part of a previously established effort to promote the use of concept maps in several areas of expertise. In particular, the test case systems implemented were highly influenced by previous developments at the Knowledge Science Institute of the University of Calgary. These developments, and their path of influence, are briefly described on the following section.

4.1 PREVIOUS WORK.

The Knowledge Science Institute has a comprehensive history of concept mapping development. Initial implementations have encouraged the development of subsequent systems, as shown on Figure 8. In this figure, links are used to construct a development hierarchy where nodes depict diverse systems and class libraries. Nodes are differentiated by their shape and fill color. Ellipse nodes represent class libraries (in this case, CMap and jCMap); rectangular nodes represent standalone applications; and shaded rectangular nodes represent Web-embeddable applets and plug-ins.

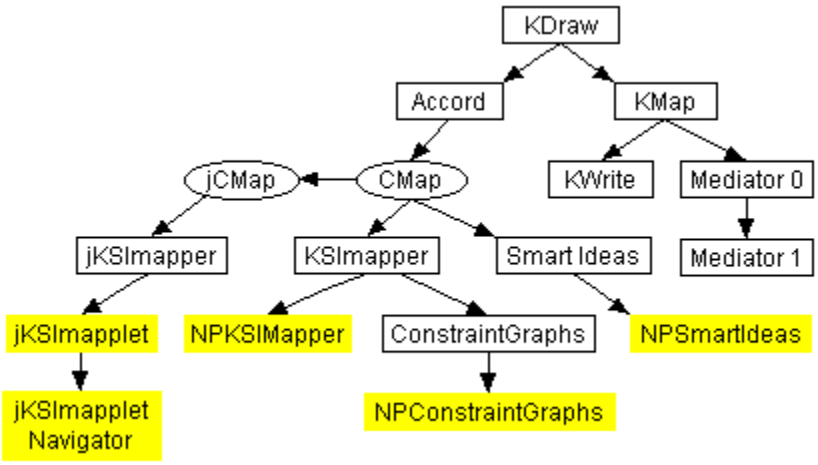


Figure 8. Concept Mapping Development at the Knowledge Science Institute.

Each of these concept mapping developments is briefly explained as follows (Gaines, Kremer and Flores-Méndez, 1996):

- **KDraw** (Gaines, 1993) is a visual language version of the CLASSIC (Borgida, Brachman, McGuiness and Resnick, 1989) knowledge representation language. Developed for the Apple Macintosh, this system implements formal concept maps for single user environments. This system allows undo and redo.

  Characteristics:

  * Formal
  * Executable
  * Undo/redo

  Platforms:

  * Macintosh

- A direct descendant of KDraw, **KMap** (Gaines and Shaw, 1995b) is a generic visual language tool that can emulate specific languages, including KDraw. All operations on the concept maps are sent as events to AppleScript (Goodman, 1993), enabling KMap to be programmed for different applications.

Characteristics:	*Platforms:*
∗Formal and informal (language definable)	∗Macintosh
∗Executable (in KDraw emulation)	
∗Undo/redo	
∗Hypermedia	
∗Scriptable (AppleScript)	

- **Mediator** is a generic knowledge management system. **Mediator 0** is a system derived from KMap, which has been extended to implement scripts for enabling two users to access concept maps in a synchronous, "What-You-See-Is-What-I-See" (WYSIWIS) manner.

Characteristics:	*Platforms:*
∗Multi-user interface (weak WYSIWIS)	∗Macintosh
∗Undo/redo	
∗Hypermedia	
∗Scriptable (AppleScript)	

- **Mediator 1** is implemented using a concept mapping tool known as XConMap (Lapsley, 1995). XConMap is essentially a re-implementation of KMap under the Unix XWindows operating system. It was designed to provide an Internet implementation of Mediator by working in conjunction with Netscape Navigator.

Characteristics:	*Platforms:*
∗Hypermedia	∗XWindows

- **KWrite** is a fully capable word processor implementation. This system allows to embed multimedia information and concept maps produced by KMap and KDraw.

Characteristics:

- Formal
- Executable
- Undo/redo
- Hypermedia
- Scriptable (AppleScript)

Platforms:

- Macintosh

- **Accord** (Kremer, 1993) is a concept mapping tool based on KMap. This system is implemented on the Microsoft Windows operating system. In addition to the features found on KMap, Accord also implements the notion of persistent hyperbases.

Characteristics:

- Hypermedia
- Persistent hyperbase

Platforms:

- Windows-16

- **Smart Ideas** (Smart Ideas, 1996) is a concept mapping tool that was developed based on the Accord implementation. This system is available as a commercial product.

Characteristics:

- Commercial system (as opposed to research)
- Hypermedia
- Persistent hyperbase

Platforms:

- Windows-32

- **NPSmart Ideas** is the Netscape plug-in version of Smart Ideas. It allows browsers to access concept maps stored as read-only resources on the Internet.

Characteristics:

- Commercial system (as opposed to research)
- Web browser-embeddable
- Hypermedia
- Persistent hyperbase

Platforms:

- Windows-32

- **KSIMapper** is the demonstration program for the CMap class library, which was developed using the C++ programming language. This application allows manipulation of informal concept maps on single-user environments.

  Characteristics:

  ∗Undo/redo

  Platforms:

  ∗Windows-32

- **NPKSIMapper** is the Netscape plug-in version of KSIMapper. It implements functions to allow interaction with Netscape Navigator embeddable widgets through JavaScript.

  Characteristics:

  ∗Web browser-embeddable

  ∗Multi-user interface (weak WYSIWIS)

  ∗Undo/redo

  ∗Hypermedia

  ∗Scriptable (JavaScript)

  Platforms:

  ∗Windows-32

- **Constraint Graphs** (Kremer, 1996) is a system based on the KSIMapper implementation. This system has been extended to emulate formal visual languages.

  Characteristics:

  ∗Formal

  ∗Can be constrained to multiple formal visual languages

  ∗Undo/redo

  Platforms:

  ∗Windows-32

- **NPConstraint Graphs** is the Netscape Navigator plug-in version of Constraint Graphs.

Characteristics:

- Web browser-embeddable
- Formal
- Can be constrained to multiple formal visual languages
- Multi-user interface (weak WYSIWIS)
- Undo/redo
- Hypermedia
- Scriptable (JavaScript)

Platforms:

- Windows-32

- **jKSImapper** is the Java implementation of KSIMapper. This system is based on the Java jCMap class library, which was derived from the C++ CMap class library. jKSImapper allows manipulation of concept maps on single and multi-user environments.

Characteristics:

- Multi-user interface (weak WYSIWIS)
- Undo/redo

Platforms:

All platforms where a Java interpreter is implemented.

- **jKSImapplet** is the Web-browser-embeddable and downloadable version of jKSImapper.

Characteristics:

- Web browser-embeddable
- Multi-user interface (weak WYSIWIS)
- Undo/redo
- Hypermedia
- Scriptable (JavaScript)

Platforms:

All platforms where a Java interpreter is implemented.

- **jKSImapplet Navigator** is a Java applet based on jKSImapplet that implements an interactive interface for World Wide Web navigation. This browser-embeddable applet

allows URL addresses to be associated with nodes for accessing hyperlinked HTML documents and concept maps. This system also supports multi-user environments.

Characteristics:	*Platforms:*
∗Web browser-embeddable ∗Multi-user interface (weak WYSIWIS) ∗Undo/redo ∗Hypermedia ∗Scriptable (JavaScript)	All platforms where a Java interpreter is implemented.

- **CMap** is a C++ class library on which the KSIMapper and Constraint Graphs applications are based.
- **jCMap** is a Java class library on which the jKSImapper and jKSImapplet applications are based.

Of the programs and classes described above, four were specifically implemented for this research. These are the jCMap class library, the jKSImapper standalone application, the jKSImapplet Java applet, and jKSImapplet Navigator. jKSImapplet Navigator has been included as an example of the application of jKSImapplet as a Web navigational tool.

4.2 SYSTEM REQUIREMENTS.

The purpose of this section is to describe the requirements for the test case systems. As mentioned at the beginning of this chapter, there are two general requirements for the test case system: it should be able (a) to handle concept maps, and (b) to allow multi-user manipulation of those concept maps on the Web. These issues are addressed on the following sections.

4.2.1 DEVELOPMENT FRAMEWORK.

The creation of a development framework requires the definition and understanding of the components of the system to construct. In this case, the definition and understanding

of the entity *concept map* will help to elicit the requirements for a concept mapping system.

Unfortunately, the term “concept map” eludes a concise description and definition. In Chapter 1, concept maps were defined as diagrams composed of links and nodes of different types. However, this description is too broad and ambiguous to be useful.

Instead, a definition for concept maps can be found by examining their use under the light of three levels of analysis (Gaines and Shaw, 1995b):

- ***The abstract perspective*:** From an abstract perspective, the basic concept map data structure consists of typed nodes, some of which are linked. Each node has a type, a unique identifier and a content (which may itself be structured, for example, as label plus other data). A node may enclose other nodes, allowing the construction of hypergraph structures in which a single link may connect sets of nodes. Links are visually represented by lines between nodes. These lines may or may not contain arrow heads to represent logical flow.

- ***The visualization perspective*:** From a visualization perspective, concept maps may be seen as diagrams, using the term to mean a drawing with reasonably well understood meaning in certain domains. To provide a consistent relation between the visual features and their internal infrastructure, the visual attributes of nodes and links need to be in one-to-one unique correspondence with their types. The node type may determine, and itself be determined by, the node shape, frame color, fill color, whether the type name is displayed and, if so, in what type face, style, size and color, and whether part of the content is displayed and, if so, in what type face, style, size and color. If links are typed, their types may determine, and be determined by, labeling, line thickness, color, cross-hatching, or other forms of decoration.

- ***The discourse perspective*:** From a discourse perspective, concept maps may be seen as an instrument to represent and communicate knowledge through visual languages. In this case, an abstract structure represented as a diagram in visual terms can be used as a knowledge representation and communication instrument when interpreted by some

community. This circumstance creates an exact parallel between natural languages and visual languages, where the abstract grammatical structures and their expressions in a medium take on meaning only through the practices of a community of discourse.

Each of the perspectives mentioned above represents one of three layers that can be supported by concept mapping tools. In the case of the test case application presented, the class library used to implement it supports each of these perspectives to certain extend, as explained as follows:

- The ***abstract perspective*** is supported by defining node and link classes, and by defining classes that allow their manipulation inside logical data structures. A context box class, which is a variation of the node class, allows the arrangement of nodes inside its boundaries for grouping purposes. The node, context box and link classes implement content in the form of text labels. The link class has attributes to store the number of lines segments composing each link instance, as well as information on the attachment, if any, maintained by each one of those line segments. Hypermedia behavior is supported by using networking classes that allow interaction with remote processes and access to Internet resources.

- The ***visualization perspective*** is supported by classes that display visual attributes related to the underlying classes from the abstract layer. Visualization classes allow the displaying of shape and text attributes for nodes; and line segments, arrow heads and a text label for links. These attributes can be customized using a color palette and, in the case of text labels, different font types, styles and sizes.

- The ***discourse perspective*** is supported on its most basic form, meaning that knowledge structures can be freely constructed, but without any automated process enforcing specific formalisms. This characteristic can represent an advantage for authors willing to extend the system to support their formal representation of choice. As it will be mentioned further on Chapter 6, implementation of formalisms is part of future work proposed for this system.

4.2.2 SUPPORTING MULTI-USER ENVIRONMENTS.

In recent years, computers have evolved from being purely a computational machine to a symbolic manipulator (word processors, graphics applications, databases) and, more recently, to a vehicle for human communication (e-mail) and knowledge repository (artificial intelligence, expert systems) (Gaines, 1991). Eventually, computers have matured to interact with the surrounding human environment. The first computers viewed data as the center of computation, with almost no human interaction as part their execution process. As time passed, this scenario evolved, and computers started to support interaction not just with single users but with entire communities of users working together.

Currently, most software systems are designed to support interaction between user and computer. However, market exigencies urge for software that also supports interaction between users, since much of an individual's work is within the context of a group. Organizations are not made up of people working individually at their desks, but rather people cooperating and collaborating as groups or teams. In this context, Computer Supported Collaborative Work (CSCW) is an area that studies the use of computers to enhance and extend people's natural abilities to collaborate for achieving their goals (Kremer, 1993). As shown in Table 4 (Ellis, Gibbs and Reln, 1991), CSCW systems must provide support for individuals working alone or in sub-groups, and at the same or at different times.

	Same Place	***Different Place***
Same Time	face-to-face meetings	synchronous remote interaction
Different Time	asynchronous interaction	asynchronous remote interaction

Table 4. Interaction modes of CSCW systems.

The interaction modes shown in this table are explained as follows:

- ***Face-to-face meetings*:** Face-to-face meetings are the same as traditional meetings involving whiteboards. Their difference, however, resides in the application of computational tools to achieve goals. In a common scenario, meeting members use computer terminals to manipulate shared information, while facilitators can use large electronic boards to guide and synchronize members' efforts.

- ***Synchronous remote interaction*:** Synchronous remote interaction is similar to face-to-face meetings in the sense that members work at the same time, as they do in meetings, but using terminals from different locations. This interaction mode has the disadvantage of restricting the use of gestures and basic human interaction, which are common components of traditional meetings.

- ***Asynchronous interaction*** and ***asynchronous remote interaction*:** Under these interaction modes, systems are used by one member at a time (single user environment), regardless of whether the system is executed locally (asynchronous interaction) or from a remote computer (asynchronous remote interaction).

The four interaction models mentioned above are covered by the present test case implementations. Remote interaction is supported by a server process that acts as a command broadcaster and central repository of information. Further information regarding the server implementation is detailed on Chapter 5, under the Section "5.2. Runtime System Architecture."

4.3 JKSIMAPPER.

This section describes the features implemented by the standalone Java concept mapping system, which is named jKSImapper. This system is based on the Java jCMap class library, which descends from a previous developed C++ class library known as the CMap class library.

jKSImapper is a standalone Java application that allows the generation and manipulation of concept maps. This system uses a graphical user interface based on windows, dialogs

and menus to handle user requests. A pointing device is also required to modify specific attributes of objects, such as position, size and link attachment.

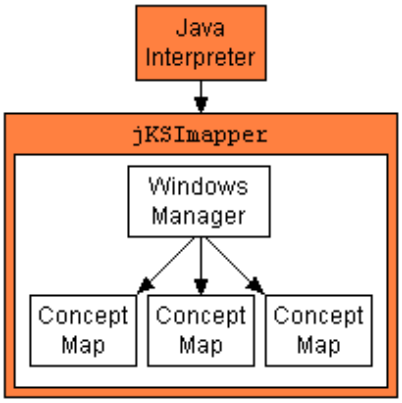


Figure 9. jKSImapper components.

As shown on Figure 9, jKSImapper is composed of two general component types: a Windows Manager and a group of zero or more Concept Map Windows.

4.3.1 The Windows Manager.

As its name suggests, the Windows Manager is designed to control and to organize several windows, which in this case will contain one concept map per instance. The Windows Manager can be seen more as an abstract tracker of displayed windows, than a part of the concept mapping process.

One advantage of using this approach is that one single Java interpreter supports all the concept maps that a user might want to utilize at a given time. If each concept map window has been implemented to use its own instance of the interpreter, it would have resulted on more resources being consumed to maintain each one of those independent instances.

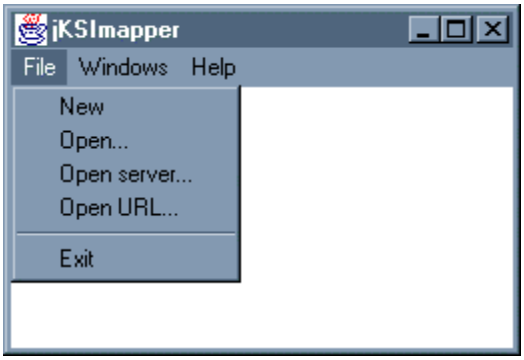


Figure 10. The Windows Manager.

The Windows Manager implements procedures that allow users to display and access concept mapping elicitation windows. As shown on Figure 10, there are three options on the menu bar of a Windows Manager. These options are:

- ***The "File" menu option*:** This pull-down menu contains most of the options of the Windows Manager. These options allow users to initiate concept maps from scratch, to load concept mapping files stored on local or remote computers, and to exit the application. Remote files can be downloaded from a jKSImapper Server, or from the Web using URLs.

As will be explained further in Chapter 5, the opening of a local file indicates that events generated by the user will be handled locally, without the mediation of any other (locally or remote) process. The same policy applies to files downloaded using URLs. On the other hand, the opening of a server file implies that the user is joining or starting a remote session on the server. Each one of these sessions support one independent concept mapping elicitation.

- ***The "Windows" menu option*:** This menu option contains one single sub-option named "List...", which displays a list containing all active Concept Map Windows. By selecting items from this list, users are able to bring to top those existing windows. This option is of great value for locating specific concept map windows when the screen is cluttered with multiple windows from this or other applications.

- ***The "Help" menu option*:** Up to the present implementation, just an "About" option is supported by this menu. Selecting this option will display an information dialog containing the name of the application, its place and year of development, and a copyright notice.

4.3.2 The Concept Map Window.

Concept Map Windows provides the core functionality for concept mapping elicitation. As shown in Figure 11, Concept Map Windows can manipulate nodes, links, and context boxes, to construct complex structures.

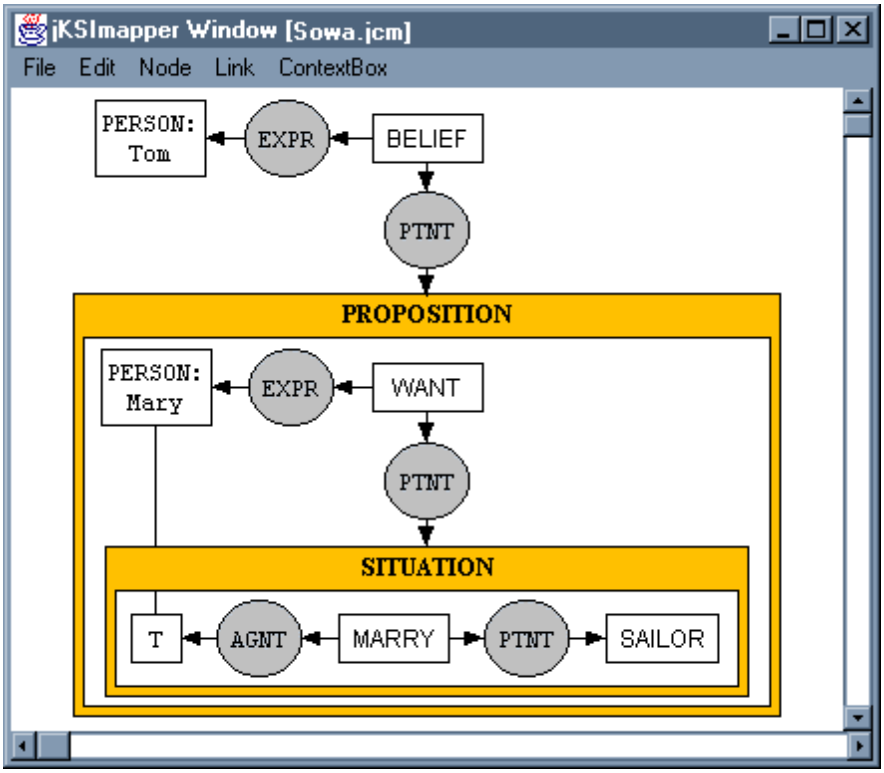
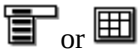
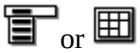
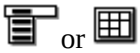
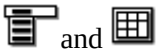
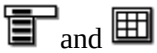
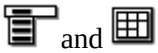
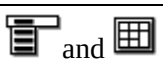


Figure 11. A formal concept map describing the sentence "Tom believes that Mary wants to marry a sailor."

In this figure, a graph created using the formal representation language known as Conceptual Graphs (Sowa, 1984) is used as an example. This graph represents the

sentence “Tom believes that Mary wants to marry a sailor.” The structure shown on the concept map can be read as “There exists a *belief* whose experiencer is *Tom* and whose patient is the proposition that there exists a *want* whose experiencer is *Mary* and whose patient is the situation that there exists a *marriage* whose agent is something (Mary) and whose patient is some *sailor*.”

Concept Map Windows rely on a set of predefined operations to modify the composition of concept maps. As shown in Table 5, these operations are applied to one or several target elements and can be activated by using specific user interface components.

	Target Element			
Operation	Node	Link	Context Box	Concept Map
Creation				
Deletion				
Selection				
Move				
Resize				
Attach/Detach				
Text Label				
Font				
Color				
Shape				
Line Segments				
Arrow Heads				
Undo/Redo				
Save				

User Interface Mediums:  Menu  Mouse  Keyboard

Table 5. Operations performed in Concept Map Windows.

The operations shown in this table are explained as follows:

- ***Creation of elements***: This operation provides the initialization of three different concept mapping objects, which are described below:

  - **Node:** As shown on Figure 12, nodes can be instantiated as rectangle, ellipse, or rounded rectangle shapes. The default attributes for a node are: empty text label, white fill color, black color for border and text, and plain style 12-point Courier font. Text labels are displayed at the center of the node. New nodes can be instantiated by using the "Node | New | <*shape*>" menu option (where <*shape*> represents one of the shapes available).

Figure 12. Available Shapes for Nodes.

  - **Link:** As shown on Figure 13, links can be created as a two (binary), three (trinary), or four (quadrary) line-segmented arcs. The default attributes for a link are: empty text label, no arrow heads, black color for lines, arrow heads and text, and plain style 12-point Courier font. Text labels on links are positioned at the center of the joining point of the line segments. New links can be instantiated by using the "Link | New | <*line-segments*>" menu option (where <*line-segments*> represents one of the available link configurations).

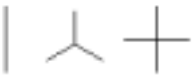


Figure 13. Available Links.

  - **Context Box:** Context boxes can be considered a specialized type of node. Context boxes share all the attributes existent for nodes except for the shape and the position of the text label. As shown on Figure 14, context boxes are just delivered in a single shape: as a rectangular frame containing an empty rectangular space. Context boxes

have the peculiarity that the inside rectangle has a small offset downwards in comparison with the geometrical center of the external rectangle. This offset creates a header at the upper edge, which is used for displaying a text label. Context boxes are instantiated by using the "ContextBox | New" menu option. The default attributes for a Context box are: empty text label, white fill color, black color for border and text, and plain style 12-point Courier font.

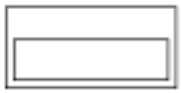


Figure 14. Example of a Context Box.

In general, newly created objects are displayed on the upper left corner of the painting area.

- ***Deletion of elements:*** This operation is applied to selected objects. Once this pre-condition is fulfilled, selected elements are removed by choosing the "Edit | Delete" option from the menu, or by pressing the <*delete*> key from the keyboard.

- ***Selection of elements:*** Selection of elements is achieved using any of two methods: (a) by mouse-clicking on the element (the body in nodes, lines in links, and in the frame of context boxes); or (b) by enclosing the targeted element(s) with the selection rubber band.

Figure 15 shows three selected elements: the rectangular node "BELIEF," the circular node "PTNT," and the link joining them. As it is depicted, the visual representation of a selection state is marked by small solid squares at the edges of the rectangular area occupied by the element, in the case of a node or context box, or by small squares located at the extremities of line segments, in the case of a link. Figure 15 also shows a dotted rectangle surrounding the node labeled "PERSON: Tom." This dotted rectangle is called the selection rubber band, and it is used to select one or more elements of a concept map. The selection rubber band is invoked by clicking on an empty point of the drawing area, and dragging the mouse to a position that will enclose the element(s) to

select. To add elements to an existing selection, users may click on new elements while holding the <*control*> key. This technique is also used to deselect one of the elements of a group of selected objects.

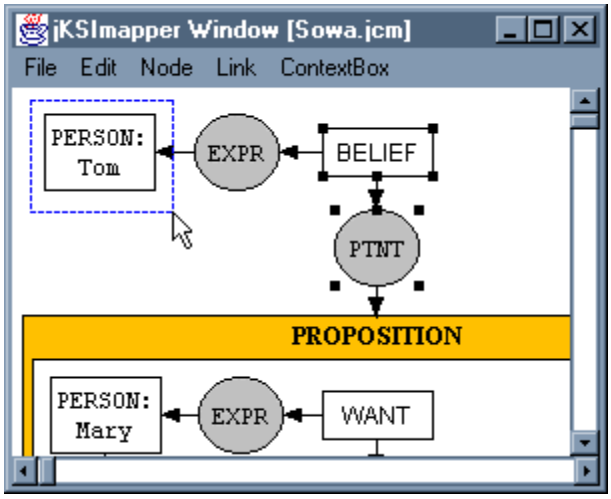


Figure 15. jKSImapper selection example.

- ***Movement of elements:*** Movement is accomplished by mouse-clicking inside the boundaries of an element and then dragging the mouse without releasing the button.

Links are moved by clicking on one of their line segments. The movement of links with line segments attached to nodes, as well as the movement of nodes with line segments attached to them, will result on the resizing of those line segments. Additionally, the dimensions of a line segment will be determined by the joining point of a link and the closest point of attachment to a node. In the case of context boxes, their movement will result on the movement of all the elements enclosed by their frame.

It is worth mentioning that the movement of a selected object will result in the movement of all selected objects by the same offset.

- ***Resizing of elements:*** Nodes and context boxes can be resized by clicking and dragging one of their selection squares (thus, selection mode is a pre-requisite). Links cannot be resized, but (as previously explained under “Movement of elements”) their line

segments can be resized if the joining point of a link is moved while the line segments are attached to a node or context box.

- ***Attachment/detachment of line segments:*** Selection squares on the edges of link line segments on a link can be clicked and dragged over the body of a node (or over the frame of a context box) for attachment. Attached line segments can be detached by dragging the line segment's selection square to an empty point on the drawing area. In case of detachment, line segments will return to a default stationary position.

- ***Editing of text labels:*** Nodes, context boxes and links support text labels. A text label is assigned to selected objects by using the "Edit | Set Label | Edit..." menu option. This option displays a dialog box requesting a string of characters to be used as a label. If just one object is selected when invoking this option, the object's current label is displayed for edition. No text will be displayed if multiple objects are found to be selected.

- ***Selecting font properties for text labels:*** Text labels can support three different font attributes, which are described as follows:

  ∗ **Type:** Text labels can use *Courier*, *Helvetica*, and *TimesRoman* font types.

  ∗ **Style:** Text labels can use *plain*, *italic*, *bold* and *bold-italic* font styles.

  ∗ **Size:** A set of predefined sizes has been chosen for the present implementation. These sizes range from 8 to 40 points.

Font attributes can be modified by using different selections available under the "Edit | Set Label | Font" menu option.

- ***Selecting objects' colors:*** jKSImapper allows to color different visual attributes on selected objects. Colors supported are, in alphabetical order, black, blue, cyan, dark gray, gray, green, light gray, magenta, orange, pink, red, white and yellow. These colors can be applied to the following attributes on objects:

  ∗ **Nodes and Context boxes:** Color is supported on the body, border and text label of nodes and context boxes. Color is modified by using the sub-menus available on the

"Node | Set Color" and "ContextBox | Set Color" menu options for nodes and context boxes, respectively.

- **Links:** Colors can be applied to line segments, arrow heads, and text labels. Changes are done by using options available on the "Link | Set Color" sub-menu. Colors and font attributes are modified by accessing nested menu options.

- ***Modification Nodes' shape:*** Nodes can be instantiated and modified using any of three geometrical shapes: rectangle, ellipse, and rounded rectangle. Options to modify the shape of selected nodes can be found under the "Node | Set Shape" menu option.

- ***Modifying the number of Line Segments on Links*:** jKSImapper allows the creation of three link types, which are differentiated by their number of line segments. To modify the number of line segments of a link, users need to access options found under the "Link | Set Arity" menu option.

- ***Arrow Head settings on Links*:** jKSImapper provides different arrow head arrangements. As shown on Figure 16, arrow heads implemented can be found in the any of the following configurations:

  (a) Undirected: No arrow heads are shown. This is the default configuration when a link is created.

  (b) Directed: One line segment will contain an arrow directed towards its edge, while the remaining line segments will stay arrow-less.

  (c) Double directed: All line segments, except one, will show an arrow directed toward their edge. The remaining line segment will have an arrow head directed toward the joining point of the link.

  (d) Bi-directed: All line segments will have an arrow head pointing toward their edge.

  (e) Double bi-directed: Line segments will have two arrow heads each, one pointing to the joining point of the link, and the second pointing towards the edge of the line segment.

These configurations can be found under the “Link | Set Arrows” menu option.

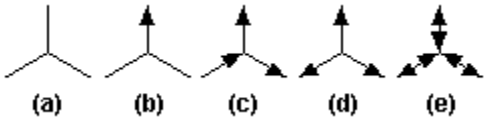


Figure 16. Arrow head configurations exemplified on trinary links.

- ***Undo/redo operations:*** All events generated by users are translated into commands that are stored after being executed by the system. Every operation performed by users results in the creation of a command that will be executed and stored on a history list (a description of the command and history list implementation is found on Chapter 5, under the Section “5.1.4. Command Handling Classes”). Commands maintained in the history list can be replayed backward (undoing previous operations) or forward (to recreate previously undone commands). When the user is participating on a multi-user session, messages requesting undo or redo operations are sent to the server and then broadcast to the clients. Under this scheme, identical history lists are maintained on the server and on each of the client members participating on each session. History lists maintained by server and client processes are emptied when the file in the session is saved.

The undo and redo options can be found under the “Edit” option on the menu bar.

- ***Saving a concept map***: Concept maps can be saved locally or remotely. To save a file, Concept Map Window instances provide three options under its “File” menu bar option:
  - **Save:** This option saves the concept map on a previously defined file. jKSImapper will retain information of the place where a concept map was previously saved, whether locally or remotely. In the case that a concept map has not been saved before, a dialog asking for a new (local) file name will be displayed.
  - **Save locally as:** This option will present a file dialog requesting a file name. This file name will be used to store the concept map on the local computer.

- **Save remotely as:** This option will display a dialog inquiring for the location of a networked computer running an instance of the jKSImapper Server process. Once a communication channel to the server has been opened, a dialog box requesting a file name will be displayed to the user. The file name provided is used for storing the concept map on the selected remote computer.

As explained in previous sections, the origin of a loaded file has repercussions on the mechanism used to handle operations generated by the user. The same circumstance is applicable when saving a file: if the file is saved locally, new commands will be handled locally; if the file is saved remotely, a public session will be created and subsequent commands will be submitted to the server process.

4.4 JKSIMAPPLET.

As described in previous chapters, Java applets running inside Web browsers confront more restrictions than Java applications do when it comes to accessing resources on client computers. For example, it is possible for an application to access the client's file system, or to open simultaneous communication channels to different servers on the Internet. However, such competence is not granted to applets, which are limited by security mechanisms enforced by Web browser implementations. These security mechanisms prevent applets (a) to access local storage resources, and (b) to open network connections to computers other than the server from where the applet was downloaded.

Limiting the functionality of programs is one method to provide a secure environment for the execution of downloadable code in the Web. Another method might be to support well-defined security mechanisms to restrict the access to computer resources according to user-defined access policies. This method has the advantage of just restricting the functionality of those programs that do not conform to the level of trustworthiness granted by the user. Unfortunately, the absence of this (or a similar) security method has lead to the implementation of more restrictive (but simpler) security mechanisms.

Under this scenario, implementing jKSImapplet required the modification of some functions available on jKSImapper. Prominent adjustments were performed on two specific areas:

- ***User interface*:** Java does not provide mechanisms to implement menus in applets. A viable solution to interact with the user is achieved by embedding widgets in the HTML document containing the applet. These widgets were programmed to trigger events that would result on the calling of Java methods inside jKSImapplet. Linking widgets' events with jKSImapplet methods is achieved using Netscape's JavaScript language facilities. A more elegant (and complex) solution is discussed on Chapter 6, under "User Interface Improvements."

- ***File storage*:** Web browsers totally restrict applets from accessing local disk storage. This circumstance makes it impossible to save concept maps on the client computer, and no alternative solution is provided in the present implementation. As a result, jKSImapplet just supports manipulation of concept maps maintained by sessions held on server processes; specifically, by a server process running on the server from where the applet was downloaded.

Figure 17 shows an instance of jKSImapplet embedded on a Web document, along with the widgets provided to interface with the user. Most operations available as menu options on jKSImapper were implemented in jKSImapplet by using buttons and combo boxes. A detailed description of the implementation of these widgets is found in Chapter 5 under "Runtime System Architecture."

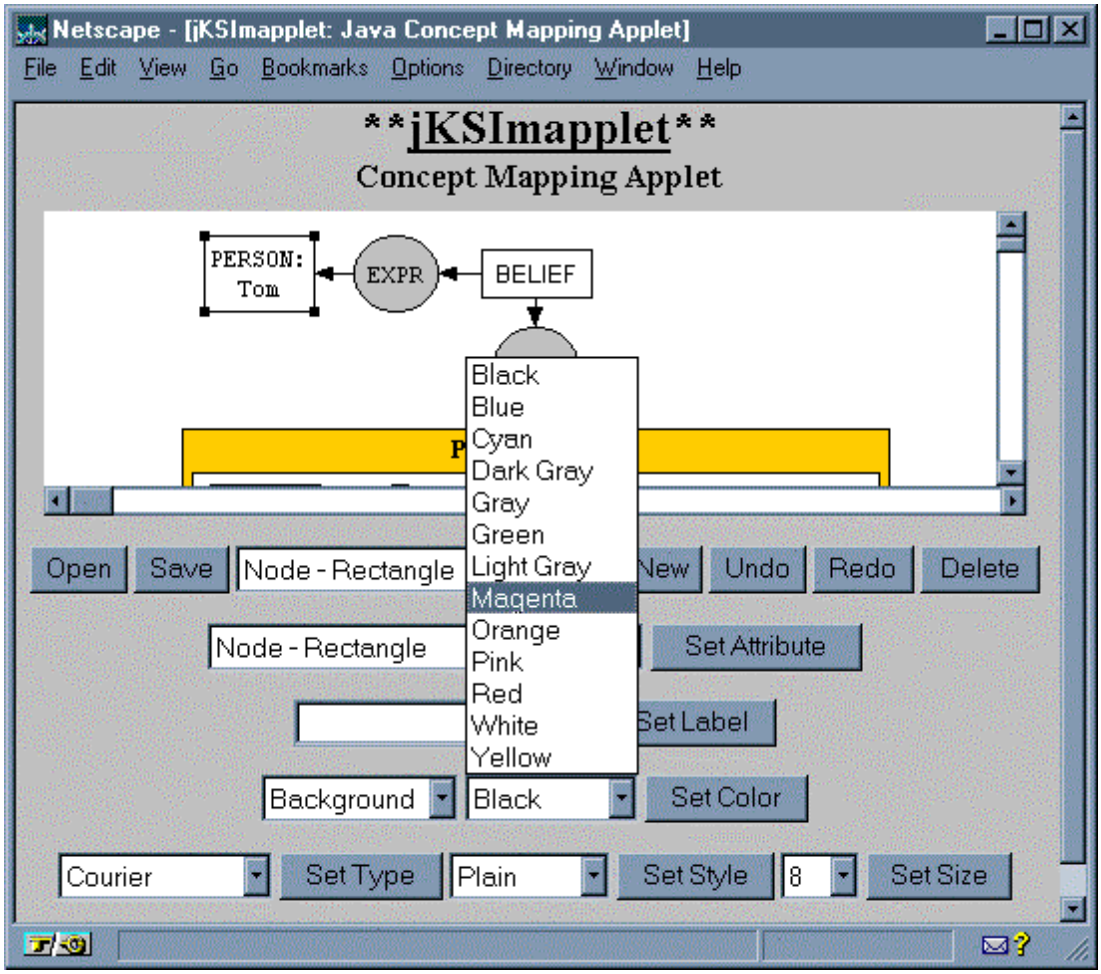


Figure 17. HTML widgets interacting with jKSImapplet through JavaScript.

4.4.1 jKSIMAPPLET NAVIGATOR.

Systems that allow multi-user manipulation of concept maps across the Internet are valuable assets for sharing ideas and information. Users might find it useful to have tools supporting their work without requiring their presence in a meeting room to achieve their goals. In this case, groups can benefit from using jKSImapper and jKSImapplet as tools for eliciting concept maps from members spread across different locations.

However, concept maps are not just aimed to communicate and gather ideas; they can also be used to organize information in different levels of abstraction. Concept maps can provide an active hypermedia interface by supporting hyperlinks to access resources. In

the case of the Web, concept mapping tools can be used as hypermedia components to access Internet resources. To this end, jKSImapplet Navigator was developed as an example of the potential of concept maps to act as hypermedia navigator tools on the Web.

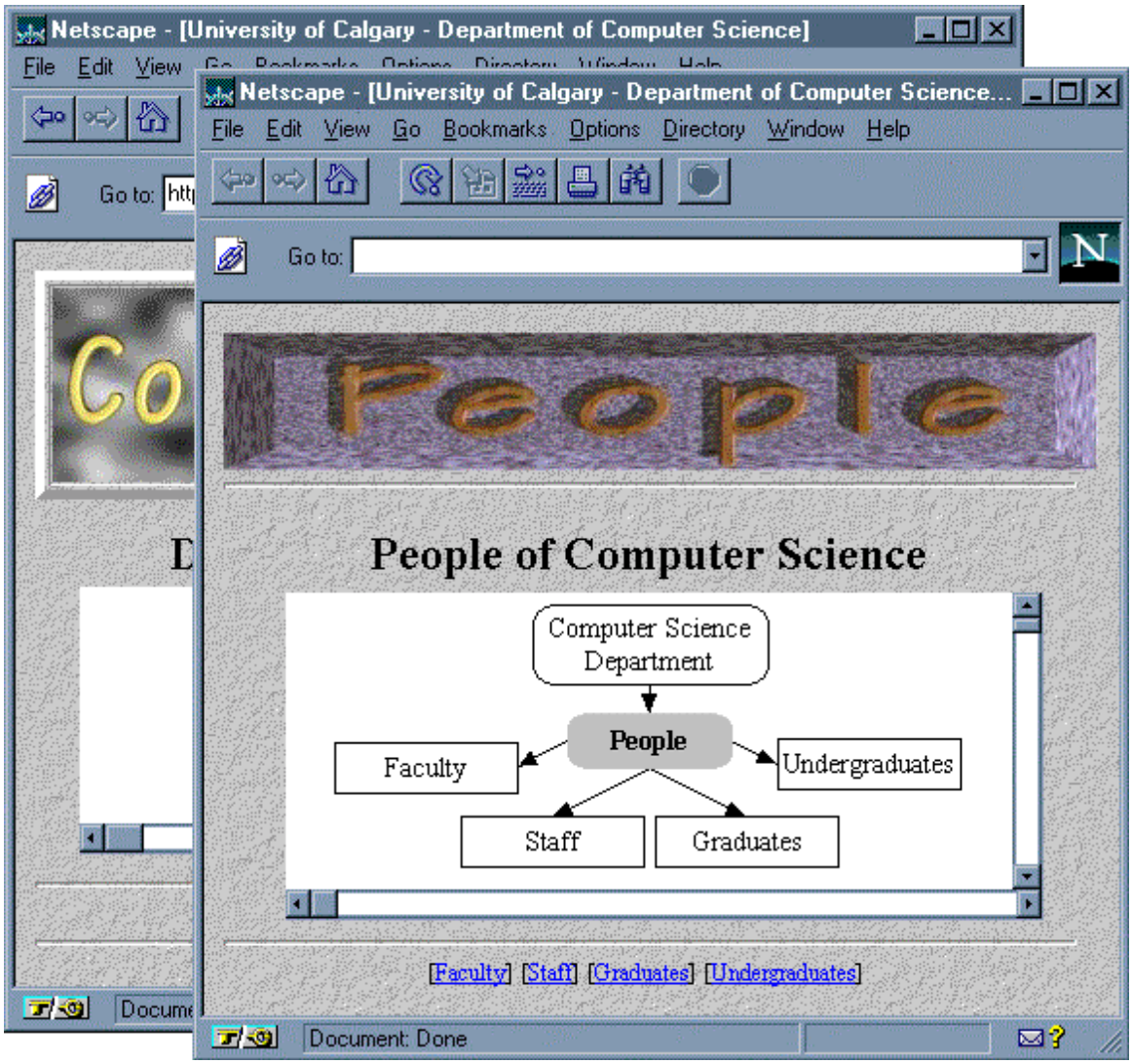


Figure 18. jKSImapplet Navigator.

jKSImapplet Navigator adds URL access capabilities to jKSImapplet. This modification permits read-only access to concept maps using URLs. It is also possible for the system to

open a multi-user concept mapping session if the downloaded concept map comes from a server where a jKSImapper server process is under execution.

Figure 18 shows a Netscape Navigator window being displayed as a response to a user selection on a concept map displayed on a separate window.

4.5 CHAPTER SUMMARY.

This chapter has described an informal concept mapping tool application. This application, known as jKSImapper, was explicitly developed as the test case for this research. It was implemented based on the jCMap class library, which was developed using the Java programming language. jKSImapper has been extended to perform as a Java applet inside Web browsers. This applet, called jKSImapplet, was further extended to perform as a navigator tool for the Web, resulting on a new applet named jKSImapplet Navigator. jKSImapplet Navigator allows access to HTML documents and concept maps addressed by URLs.

jKSImapper is based on work previously developed at the Knowledge Science Institute, as it was described on the first section of this chapter. Such systems were invaluable as a guide for the implementation of the test case systems presented.

The description of previous developments was followed by a discussion of the system requirements. A general development framework was determined by addressing the abstract, visualization and disclosure perspectives. Multi-user support was devised by analyzing the quadrants required for CSCW systems. These requirements were implemented in jKSImapper.

The last part of this chapter was devoted to introducing the operations performed by jKSImapper, jKSImapplet, and jKSImapplet Navigator. jKSImapper was described as composed of two elements: a Windows Manager and Concept Map Windows. The operations performed by jKSImapper were described in detail on Sections 4.3.1 and 4.3.2. jKSImapplet was introduced as a generic concept mapping applet derived from jKSImapper. One of several possible applications for jKSImapplet was depicted by

jKSImapplet Navigator, which is an applet capable of accessing Web resources and hypermedia concept maps inside Netscape Navigator.

The next chapter (Chapter 5) will discuss the implementation of the applications described in this chapter. This chapter will start with a description of the jCMap Java class library, which will be followed by a discussion of the implementation of the runtime system architecture for the systems presented. Chapter 5 will end with an account of the lessons learned while porting C++ code to Java.

CHAPTER 5

SYSTEM ARCHITECTURE

The previous chapter introduced the jKSImapper and jKSImapplet concept mapping systems created as part of the present work. These developments were described before by the functionality provided at the user interface level. The present chapter will provide a different perspective of these systems, detailing the underlying implementation structures.

It has been mentioned earlier that the concept mapping systems implemented were based on the jCMap object-oriented class library. However, little has been said about the composition of the library. The first part of this chapter will be devoted to describe the jCMap class library and its most relevant classes. The second part will depict the use of the class library for supporting concept mapping client applications, client applets, and server processes; as well as the relationships existing among objects of the class library that support concept mapping systems at runtime. The final section of this chapter will be dedicated to present topics that were addressed when porting the classes from C++ to Java.

5.1 THE JCMAP CLASS LIBRARY.

jCMap is a class library formed by more than 60 Java object-oriented classes. Figure 19 depicts a concept map with the most important classes from this library.

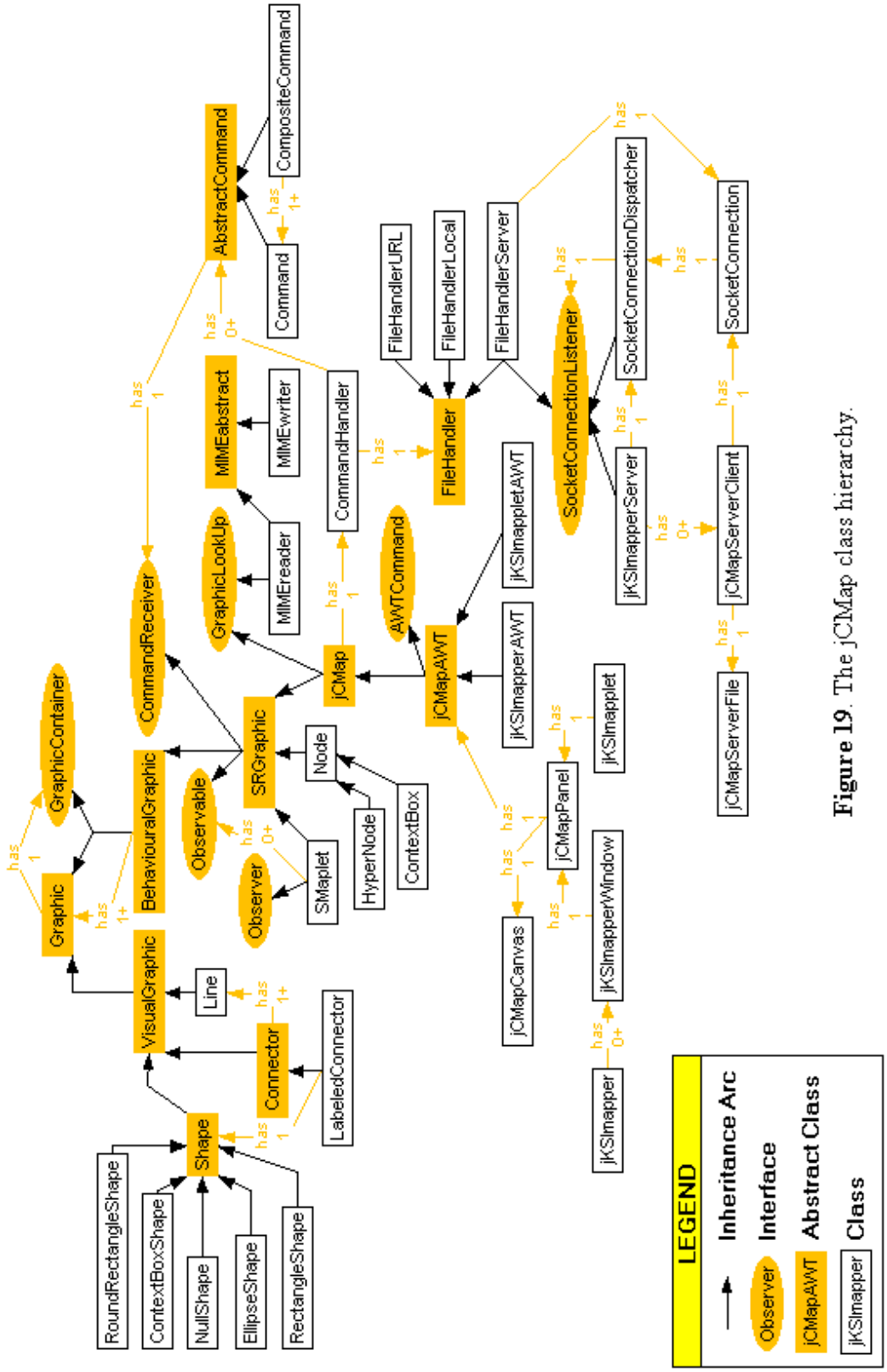


Figure 19. The jCMap class hierarchy.

This figure contains different visual elements that represent classes, interfaces and relationships. Rectangular nodes on the figure were used for representing classes; ellipse nodes, to depict interfaces; shaded nodes, for showing abstraction on classes and interfaces (which are abstract by default); and non-shaded nodes, to represent instanceable classes. Links also carry different meanings. Non-labeled links are used to picture inheritance, and labeled links are used to show association between objects at runtime.

Classes in the jCMap Class Library are organized under different areas of application according to the tasks they were designed to perform. These areas are: the Behavioural Graphic, Visual Graphic, User Interface, Command Handling, Networking and Server, and File Storage. These areas of application, as well as their classes, are explained on the following sections.

5.1.1 THE BEHAVIOURAL GRAPHIC CLASSES.

These classes provide the backbone infrastructure to perform concept mapping elicitation. They implement the behavior needed to support the manipulation of graphics under a cohesive and consistent environment. The classes composing the Behavioural Graphic class hierarchy, which are depicted on Figure 20, are described as follows:

- ***Graphic:*** This is the base class for all graphics handled on a concept map, whether they are designed as abstract or visual elements. This class provides methods and attributes to establish the runtime identity and ownership for each graphic handled during elicitation. Instances of this class will be owned, at runtime, by objects inheriting the *GraphicContainer* interface type.
- ***GraphicContainer:*** This interface declares methods that manipulate a group of graphics, and allow such graphics to query their parents about the concept mapping elicitation on which they participate.

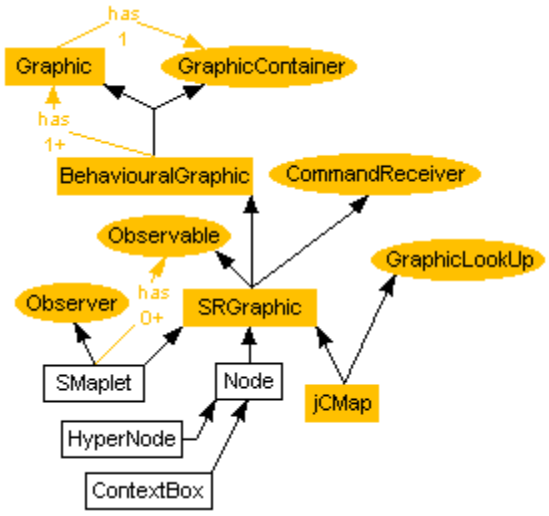


Figure 20. The Behavioural Graphic class hierarchy.

- ***BehaviouralGraphic:*** This class is designed to reflect the behavior of an individual graphical object inside a community of graphics. It also provides the foundations to support visual representation by maintaining references to visual graphic classes. *BehaviouralGraphic* implements most of the abstract methods defined on the *Graphic* abstract class and the *GraphicContainer* interface.

- ***CommandReceiver:*** This interface declares methods necessary for the processing of commands. Subclasses of this interface will provide methods for requesting, constructing, executing and undoing commands. Such methods will be implemented on the abstract class *SRGraphic* and its descendants.

- ***Observable*** *and* ***Observer:*** These two interfaces declare methods to create a one-to-many dependency between graphics. This kind of interaction is also known as "observer" (Gamma, Helm, Johnson and Vlissides, 1995). Under this scheme, an observable (*Subject*) object is the issuer of notifications. It sends out these notifications to its observers without having to know who these observers are. Any number of subscribers, which are instances of the *Observer* class, can be assigned to receive notifications.

- ***SRGraphic:*** This class implements the basic functionality to process commands and mouse-driven user requests. It also provides the grounds to convert graphics data to a streamable representation used for storage. Also implemented in this class are methods declared on the *Observable* and *CommandReceiver* interfaces.

- ***SMaplet:*** This class provides methods and attributes to manipulate links. Such methods will define algorithms to process link-specific commands and user requests. Instances of this class are designed to handle visual graphic instances of subclasses derived from the *Connector* class. Additionally, this class implements methods defined on the *Observer* interface, allowing observation of changes to *SRGraphic* objects.

- ***Node:*** This class extends methods implemented on its parent *SRGraphic* class, allowing node manipulations. This class was specifically designed to handle visual graphic representations derived from the *Shape* Visual Graphic class.

- ***ContextBox:*** This specialized type of node implements methods for handling context box shapes. Specialized operations, implemented in this class, allow graphics enclosed in the shape of the context box to move along when the location of the context box is modified.

- ***HyperNode:*** This class extends the behavior of the *Node* class, allowing the manipulation of Internet resources. Remote resources can be addressed using URLs or filenames. In the later case, filenames are used to address concept mapping archives located on a jKSImapper Server process. Instances of this class are used by jKSImapplet Navigator systems as active elements for Web navigation. Access to targeted resources is achieved by single-clicking on *HyperNode* objects displayed on a concept map.

- ***jCMap:*** This class is designed to support the manipulation of graphics as a cohesive group. Each *jCMap* object maintains a *CommandHandler* instance to process commands, either targeted to itself, or to one of the abstract graphics maintained during an elicitation. This class can be considered the pivotal component of a concept mapping system.

5.1.2 The Visual Graphic Classes.

These classes are used to graphically represent information maintained by Behavioural Graphic objects. As a result, Visual Graphic objects will just exist in a concept map as long as they are related to an Behavioural Graphic object. Classes from this area are designed to maintain information related to the drawing of the graphic of choice. Figure 21 depicts the classes in the Visual Graphic class hierarchy. These classes are briefly described as follows:

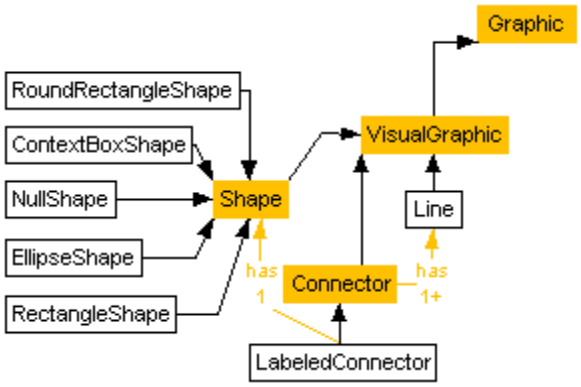


Figure 21. The Visual Graphic class hierarchy.

- ***Graphic:*** This class is the same class described on the behavioural graphic hierarchy. As previously stated, this class provides the basic characteristics for visual and abstract graphic classes.

- ***VisualGraphic:*** This abstract class define some variables and methods that will be common to displayable graphics. Because of the limited information declared on this class, it can be considered more a referencing class than a common point of functionality for visual classes.

- ***Line:*** This class encompasses enough functionality to display a single line segment belonging to a *Connector* object, which is used for grouping line segments conforming a link. *Line* objects are aware of their participation as line segments on a *Connector* (this relation is not shown in the class hierarchy). This awareness gives them the assurance that one of their edges will be positioned on the joining point shared by the lines

belonging to the *Connector*. Instances of the *Line* class have a limited scope: they do not make any assumptions about other graphics they may point, and hold just enough information for drawing arrow heads on their edges.

- ***Connector:*** This abstract class defines the mechanisms to manage a group of line segments. Instances of this class provide the foundations to visually represent *SMaplet* objects. Objects of this class maintain information to color line segments and draw arrow heads (arrow head configurations were described on Chapter 4, under “Arrow Head Settings for Links”). Operations on this class allow one to find out if an arbitrary point on the painting area is occupied by any line segment composing a *Connector* instance. Such functions allow *SMaplet* objects to query their visual representation to determine whether a mouse event has been target to them or to some other displayed graphic.
- ***LabeledConnector:*** This class extends operations found on the *Connector* class to include a *Shape* visual object as part of the connector.
- ***Shape:*** This class implements functions that are common to visual objects displaying a shape. Attributes found on this class store label, boundary and color values. The text label is drawn having as its center the geometrical center of the shape. Additional methods on this class indicate whether or not the graphic encloses an arbitrary point in the painting area. Such operations allow one to determine if *Shape* instances are targeted by mouse.
- ***RectangleShape:*** This class implements methods to display a rectangular graphic.
- ***RoundRectangleShape:*** This class implements methods to display a rounded rectangle graphic.
- ***EllipseShape:*** This class implements methods to display an ellipse graphic.
- ***NullShape:*** This class implements methods for handling a graphic that does not draw a shape. Objects from this class are particularly useful for handling text labels on *LabeledConnector* instances.

- ***ContextBoxShape:*** This class implements the behavior required for displaying context box shapes. Different from other shape representations, *ContextBoxShape* instances display a text label inside their upper frame, and not at the center of the geometrical shape.

5.1.3 The User Interface Classes.

This hierarchy of classes provides the functionality needed to integrate concept mapping classes to windowing environments. These classes, which are shown on Figure 22, allow concept mapping systems to be implemented as standalone applications, or as applets inside hypermedia Web browsers. These classes are described as follows:

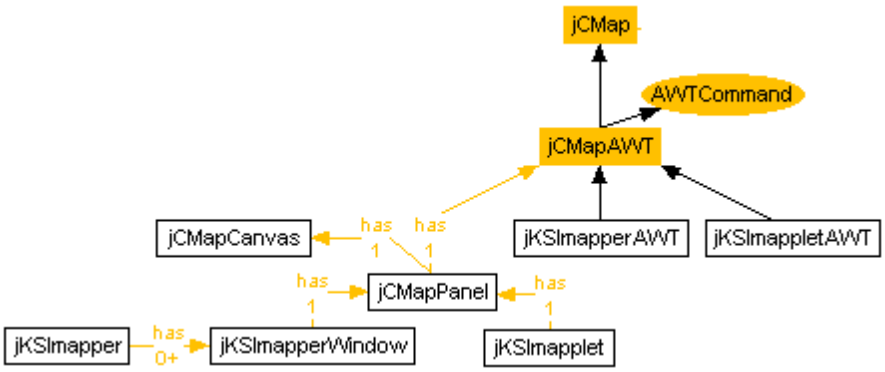


Figure 22. The User Interface class hierarchy.

- ***jCMap:*** This class is not part of the User Interface class hierarchy. Nevertheless, it is shown on this figure due to its parenthood over the *jCMapAWT* class. For detailed information concerning the *jCMap* class, please refer to the Behavioural Graphic class hierarchy.

- ***AWTCommand:*** This interface provides a set of constant values, which are used by applets and applications to modify a concept map. These values can be linked to menu options or to Web embeddable widgets. The *AWTCommand* interface can be considered just as a repository of constant values since it does not declare any new methods.

- ***jCMapAWT:*** This class implements the behavior required to receive and direct user requests to the appropriate objects for their handling. Methods on this class are designed to receive user requests as defined on the *AWTCommand* interface. Such requests will be translated into operations to create, delete, and modify concept mapping elements.

- ***jKSImapperAWT:*** This class is designed to support behavior-specific responses for concept mapping systems implemented as standalone applications. Instances of this class maintain a reference to the *jKSImapperWindow* object on which the concept map is displayed (this reference is not shown in the class hierarchy). This reference will be used for enabling and disabling the "Save", "Undo" and "Redo" menu options, according to the history list maintained by this instance.

- ***jKSImappletAWT:*** This class implements methods to make available Java applet's operations to concept map graphics. Applet objects define methods to interact with the Web. Such operations are used by *HyperNode* instances to invoke Web resources when using jKSImapplet Navigator applications.

- ***jCMapCanvas:*** This class provides a painting area to draw graphics composing a concept map.

- ***jCMapPanel:*** If one class is to be considered the unifying element between the graphical user interface and the concept mapping classes, it should be this class. *jCMapPanel* implements methods to join and synchronize user interface elements with concept mapping objects. This class is designed to support vertical and horizontal scroll bars, a drawing canvas, and an instance of a class derived from *jCMapAWT*. This instance can be either an instance of the *jKSImapperAWT* class (for standalone applications) or an instance of the *jKSImappletAWT* class (in the case of applets).

- ***jKSImapperWindow:*** This class allows the integration of *jCMapPanel* instances as part of a graphical windowing system. Figure 23 depicts the integration of components over a *jKSImapperWindow* instance. Methods implemented on this class support a menu hierarchy that will direct user options to the concept mapping structures maintained by a

jCMapPanel instance. The functionality provided by this class was described in Chapter 4, under the section labeled “Concept Map Window".

- ***jKSImapper:*** This class implements operations that support the execution of *jKSImapperWindow* instances. The functionality supplied by this class is described in Chapter 4, under the “Windows Manager” section.

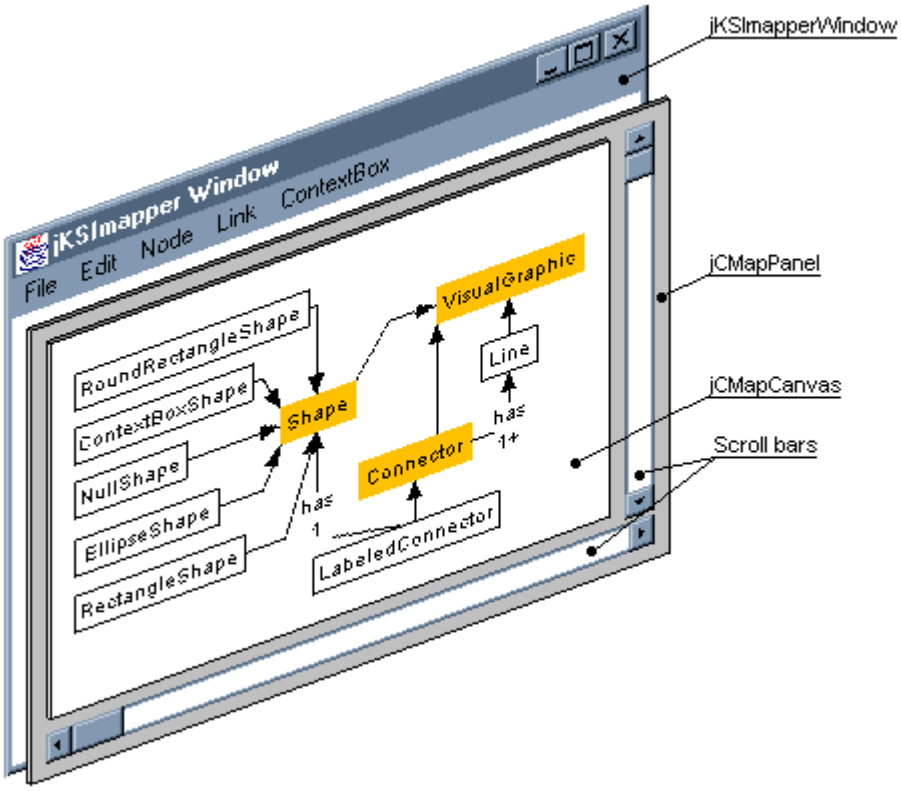


Figure 23. Runtime Composition of a jKSImapperWindow object.

- ***jKSImapplet:*** This class provides methods that support a concept mapping applet. Operations implemented on this class are used for receiving *AWTCommand* instructions generated by JavaScript widgets embedded on Web documents. Figure 5, which was introduced in Chapter 3 (pg. 40), depicts an example of a Java method invocation from a JavaScript function.

5.1.4 The Command Handling Classes.

These classes allow the definition and manipulation of commands. Commands are requests encapsulated as objects, and are issued by concept mapping graphics as the result of user interaction. Advantages of using a Command class are that requests can be queued in a history list (to support undo operations, for example) and can be used for broadcasting. Figure 24 shows the Command class hierarchy, which is composed of the following classes:

- ***AbstractCommand:*** This abstract class declares an interface to execute and undo operations. One of the attributes implemented in this class stores a reference to an object of the *CommandReceiver* type (which was described on the "Behavioural Graphic classes" section). This reference points to the object that will carry out operations in response to a specific command request.

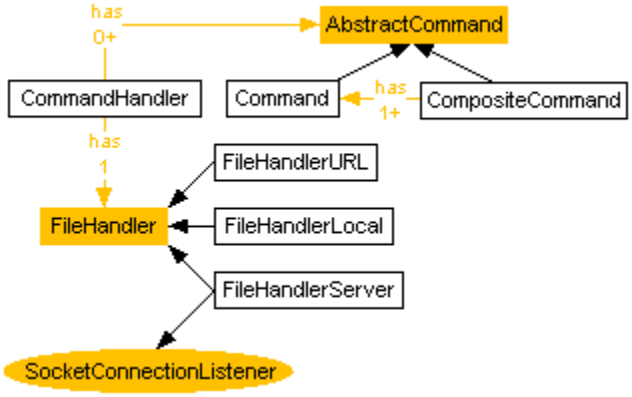


Figure 24. The Command Handling class hierarchy.

- ***Command:*** This class implements operations to execute and undo a command. Defined on this class are thirteen constants used for representing command operations. These commands store information to carry out a request and to undo the effects of a request once it has been executed. Commands defined in this class, along with their required state information, are described in Table 6 below these lines.

Command	Description
cmNEW	Command to create a new graphic. • Stores the names of the abstract and visual classes to create. The number of line segments composing a link is also stored when creating an abstract class of type *SMaplet*.
cmDELETE	Command to erase graphics. • Stores a *CompositeCommand* object containing the graphics deleted.
cmMOVETO	Command to change the location of a graphic. • Stores the graphic's previous and current positions.
cmONTOP	Command to place a graphic at the top of the Z-order. • Stores the current Z-order position of the graphic.
cmSELECT	Command to modify the selection state of a graphic. • None.
cmATTACHARC	Command to attach or detach a line segment to a graphic.. • Stores the *Observable* object to which the line segment is attached (if any); the *Observable* object to which the line segment will be attached (if any), and the identification number for the line segment itself.
cmRESIZENODE	Command to modify the dimensions of a node. • Stores the graphic's original and current dimensions.
cmSETCOLOR	Command to change the color of visual attributes on a graphic. • Store the values of the original and current color, as well as the attribute that will be affected by the change.
cmSETLABEL	Command to modify the label on a graphic. • Store the original and current text label, and the previous position of the graphic. In the case of nodes and context boxes, their dimensions are also stored.
cmSETARROWS	Command to assign an arrow head configuration to a link. • Stores the original and current arrow head configuration.
cmSETARITY	Command to modify the number of line segments on a link. • Stores the new line segment number for the link, the current and previous *Connector* instances (which visually represent the link), and the current and previous tables of *Observable* objects (used to specify the objects attached to the line segments).
cmSETSHAPE	Command to modify the shape of a node. • Store the class name for the new shape, and the previous and current *Shape* objects.
cmSETFONT	Command to modify the font attributes of a text label on a graphic. • Stores a value representing the font attribute to modify (either type, style or size), the value to be assigned, and the previous value for the modified attribute. In the case of nodes and context boxes, the previous position and dimension will be also stored.

Table 6. Command values defined on the Command class.

- ***CompositeCommand:*** This class allow the grouping of an open-ended number of commands. It provides methods to execute a sequence of commands as a single entity.
- ***CommandHandler:*** This class provides the methods required to process and store commands. Features implemented include a multi-leveled undo and redo history list for commands. This history list will store the commands executed by the concept mapping system. Additional operations allow traversing backward through the history list (for undoing commands), as well as traversing forward (for re-executing previously undone commands).
- ***FileHandler:*** This abstract class defines the basic behavior to route commands generated by users, and implements a framework to manipulate concept mapping data using a specialized MIME format specification (refer to Section "5.2.3. MIME File Format" for a detailed description of the format used). This abstract class can be extended to handle data files located in local and remote computers. Additionally, it declares methods that can be extended to permit local processing of commands (enabling single-user sessions), or to route commands to remote servers (allowing multi-user sessions).
- ***FileHandlerLocal:*** This class extends the mechanisms implemented in the *FileHandler* class to allow manipulation of concept mapping data files in client computers. This class supports single-user concept mapping elicitation.
- ***FileHandlerURL:*** This *FileHandler* sub-class implements methods that permit to read concept mapping data files located on the Web. Files are located using URLs and are downloaded as read-only resources. This class supports single-user concept mapping elicitation.
- ***FileHandlerServer:*** This class extends the file manipulation process by implementing a communication channel that transmits and receives commands from a jKSImapper

Server process. This class allows clients to access and remotely store multi-user elicitation sessions maintained by a server.

- ***SocketConnectionListener*:** This interface, which is described in the following section, declares methods used by classes receiving messages from *SocketConnection* objects.

5.1.5 The Networking and Server Classes.

These classes implement the mechanisms necessary for allowing communication from client systems to server processes. Additional classes included on this hierarchy will provide the functionality required for implementing server processes capable of supporting multiple independent elicitation sessions. Classes encompassed on this hierarchy, which is shown on Figure 25, are described as follows:

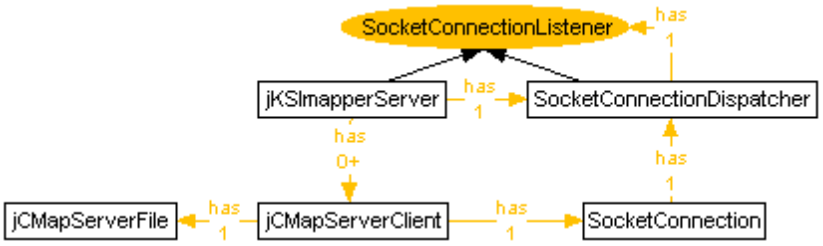


Figure 25. The Networking and Server class hierarchy.

- ***SocketConnection:*** This class implements methods for supporting a networking communication channel using TCP/IP sockets. Messages received from remote sources are submitted to objects implementing the methods specified by the *SocketConnectionListener* interface.

- ***SocketConnectionListener:*** This interface declares methods that will be implemented by classes designed to receive messages from *SocketConnection* objects. Methods declared on this interface will be used to notify the reception of a message and to inform the listener that the communication channel has been closed.

- ***SocketConnectionDispatcher:*** This class implements methods declared on the *SocketConnectionListener* interface. The main purpose of this class is to cache messages

received from a socket, and to process those messages on an orderly manner using an independent thread of execution. Messages received are placed at the end of a message queue. This queue is used for prioritizing messages according to their arrival time. At runtime, the thread of execution will redirect queued messages to an object of the *SocketConnectionListener* type for their execution.

- ***jKSImapperServer:*** This class is used for implementing server processes. It contains methods to accept and maintain client connections. Once a client is accepted, a communication channel is created to receive commands from the client. Clients can request joining an existing elicitation session, or they may request the creation of a new elicitation session. Server processes will act as broadcasters of client commands and as data repositories for concept mapping data. For more information on the runtime behavior of server processes, please refer to the section named “Server System” later in this chapter.

- ***jCMapServerClient:*** This class implements the operations necessary to maintain a communication channel between the server process and one of its connected clients. *jCMapServerClient* objects will be executed in coordination with server processes.

- ***jCMapServerFile:*** Instances of this class are used by the server process to maintain the current state of a concept mapping elicitation. This class implements methods to preserve a history list of commands generated by clients participating on a session, and methods for saving concept maps on the server computer.

5.1.6 The File Storage Classes.

Classes implemented on this area are used to read and write concept map data to secondary storage. The format in which this information is stored, is based on the specifications described on the Multipurpose Internet Mail Extension format (Borenstein and Freed, 1993). A detailed description of the format implemented can be found in the section “File Storage” further in this chapter. Classes implemented on this hierarchy, depicted on Figure 26, are described as follows:

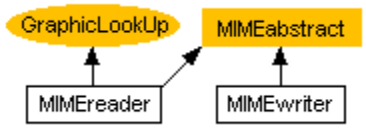


Figure 26. The File Storage class hierarchy.

- ***MIMEabstract:*** This abstract class declares data used to transform concept mapping data into MIME compliant text files.
- ***MIMEreader:*** This class implements the operations needed to reconstruct concept mapping structures from MIME formatted text lines.
- ***MIMEwriter:*** This class provides methods to construct a MIME compliant stream of data containing concept mapping structures.

5.2 RUNTIME SYSTEM ARCHITECTURE.

In the context of the work developed for this thesis, applets and applications perform as client systems to interact with server processes. While client systems are designed to perform as elicitation agents to construct concept maps, server processes are designed to be central coordinators of elicitation sessions and as data repositories. The following sections will be devoted to describe the Runtime System Architecture for these components. For the clients, the jKSImapplet and jKSImapper systems will be depicted; and for the server, the jKSImapper Server process. Additionally, the File Format used to store information will be described.

5.2.1 CLIENT SYSTEMS.

Concept mapping client systems developed from the jCMap class library can be executed as Java applets or as Java standalone applications. As explained below, the differences between applets and applications are evident on the context of their use:

- Java applets are programs downloaded from remote servers and executed locally by Web browsers. This process is automatically performed each time an applet is

encountered during Web navigation. The execution of Java applets requires browsers with a built-in Java Runtime Interpreter.

- Java applications are also programs, but they do not require the presence of a Web browser for downloading or execution. In that sense, Java applications are like any other regular standalone program that users may choose to install on their computer. Java applications require a Java Runtime Interpreter installed on the client computer.

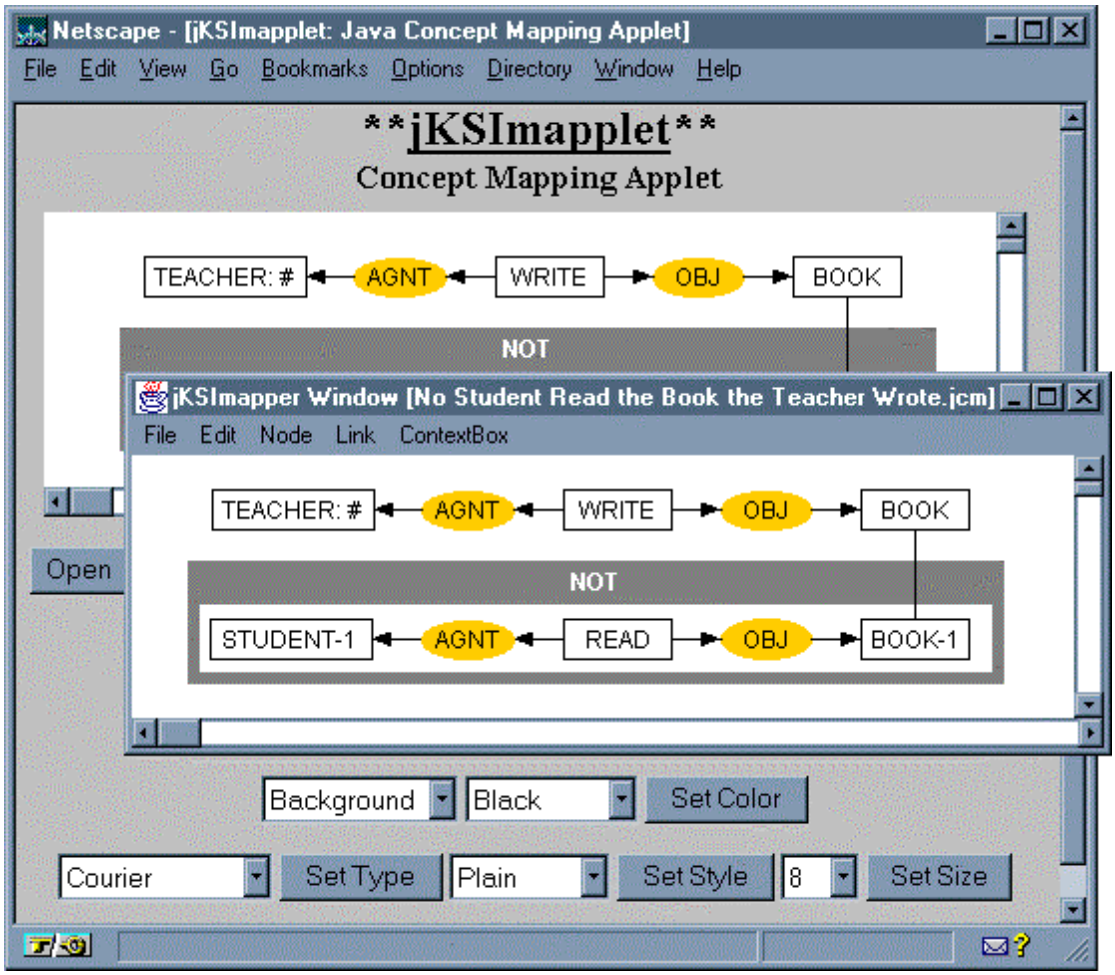


Figure 27. Groupware Concept Mapping Collaboration using jKSImapper and jKSImapplet.

Client systems implemented as part of this thesis are the jKSImapper application, and the jKSImapplet applet. Figure 27 shows instances of jKSImapper and jKSImapplet while

collaborating on the construction of a conceptual graph describing the sentence “No student read the book the teacher wrote”. Such construction is maintained in a remote session supported by a server process.

5.2.1.1 jKSImapper.

jKSImapper is the client concept mapping standalone program implemented using the jCMap class hierarchy. Instances of this program require two structures to achieve their design goals: one to support the program’s execution, and another one to support the concept mapping manipulation itself.

5.2.1.1.1 Execution Structure.

Execution is initiated by an explicit user invocation of the concept mapping program. Normally, instances of the application will be invoked by typing an instruction on the command line, as depicted on Figure 28. In this figure, the program is invoked by typing the name of the Java runtime interpreter followed by the name of the jKSImapper startup class, which is *jKSImapper*.

```
prompt> java jKSImapper
```

Figure 28. Sample invocation of jKSImapper on a Command Line.

This command will trigger the execution of the Java interpreter which will automatically load the classes required for the execution of jKSImapper. Java classes must be stored on the local computer, and they will be loaded by the interpreter as required by the application’s execution pace. This means that not all the classes in the application are loaded at once, but just the classes required to start the application are loaded at this point. Additional classes, such as the ones involving visual representation of graphics, will be loaded when needed. The execution structure for jKSImapper instances is illustrated on Figure 29.

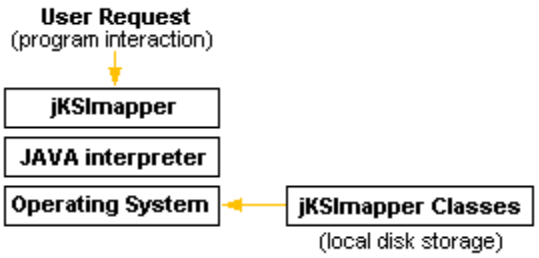


Figure 29. jKSImapper Execution Structure diagram.

5.2.1.1.2 CONCEPT MAPPING STRUCTURE.

This structure deals with the relationships existing among classes at runtime. Figure 30 shows class association upon execution. Nodes on this figure represent instances of classes, which are named after their text label. Shaded nodes are used to indicate abstract classes. When an abstract class is shown it is implied that instances from non-abstract subclasses will be used. Lines are used to represent relationships between instances. Relationships have special line terminators to indicate common multiplicity values. A solid ball represents “many,” meaning zero or more. A hollow ball indicates “optional,” meaning zero or one. A line without terminator symbols indicates one-to-one. Additional links are used to show connection with resources external to the runtime structure.

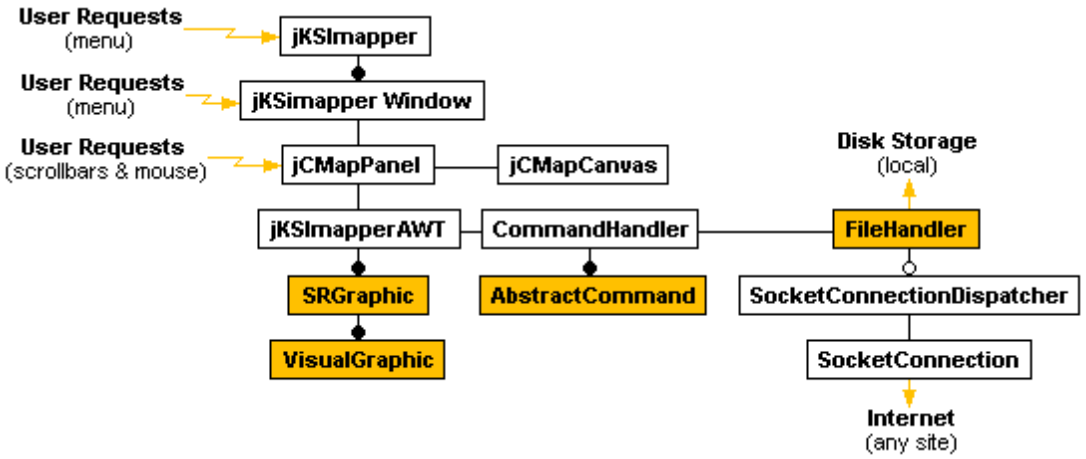


Figure 30. jKSImapper Concept Mapping Structure diagram.

In the case of Figure 30, an instance of *jKSImapper* is shown to receive events from the user via menu options. Such instance will also maintain zero or more *jKSImapperWindow* objects. Each of these objects will receive menu events from the user, and will hold exactly one instance of *jCMapPanel*, which will receive events generated by the user as a result of scrollbar and mouse manipulation. This instance is shown to be associated with one instance of *jCMapCanvas* (which provides the concept mapping painting area), and one instance of *jKSImapperAWT* (used for maintaining the elicited concept mapping data). The *jKSImapperAWT* object, just mentioned above, will hold zero or more objects derived from the *SRGraphic* abstract class (such objects can be nodes, links or context boxes); and a *CommandHandler* object for processing and storing instances from subclasses of the *AbstractCommand* class (either *Command* or *CompositeCommand* class instances). *SRGraphic* instances are associated with one or more visual instances of subclasses derived from the *VisualGraphic* class. The *CommandHandler* object depicted holds one instance of a class derived from *FileHandler*, which can be *FileHandlerLocal*, *FileHandlerURL* or *FileHandlerServer*. These classes are designed to load and store concept mapping data on local secondary storage; to read remote data files addressed by URLs; and to maintain a remote connection with a server process, respectively. Remote connections are implemented by using instances of the *SocketConnection* and *SocketConnectionDispatcher* classes. Such objects will send and receive commands when participating on a multi-user concept mapping elicitation session.

5.2.1.2 jKSIMAPPLET.

jKSImapplet is the client concept mapping applet implemented using the jCMap class hierarchy. As well as jKSImapper instances, this program will require two runtime structures, one for supporting the applet's execution, and another one for supporting the concept mapping manipulation data.

5.2.1.2.1 EXECUTION STRUCTURE.

In the case of jKSImapplet instances, execution is initiated automatically as a result of Web navigation. The execution of the applet is started after accessing a Web document containing a reference to the concept mapping applet. Figure 31 shows an example of a reference to the applet embedded on an HTML document. In this figure, the *code* parameter on the *<applet>* tag is used to invoke the *jKSImapplet.class*, which is the jKSImapplet start up class. From the remaining parameters, *codebase* is used to identify the URL base location for the class, and *width* and *height* to represent the display area designated to the applet when executed on a browser.

```
<applet code    = jKSImapplet.class
        codebase= http://www.cpsc.ucalgary.ca/~robertof/jKSImapper/
        width   = 500
        height  = 300>
</applet>
```

Figure 31. Sample invocation of jKSImapplet declared inside an HTML document.

If a reference to jKSImapplet is met while navigating on the Web, the client's browser will react by invoking an instance of the Java interpreter to download and execute the applet's classes. Java-enabled browsers are deployed with a Java interpreter integrated as part of the browser.

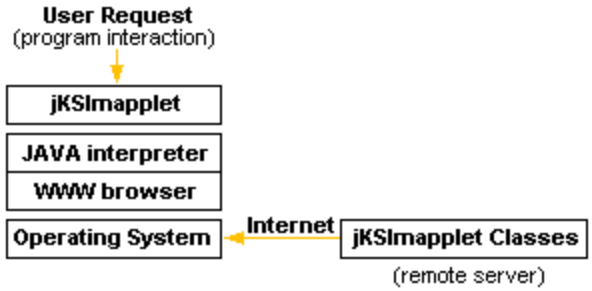


Figure 32. jKSImapplet Execution Structure diagram.

As in the case of jKSImapper, jKSImapplet classes are requested by the interpreter when they are needed for execution. In the case of jKSImapplet, this circumstance is more evident; the required classes will stop the applet's execution until they have been fetched from the remote server where the classes reside. The execution structure for jKSImapplet instances is illustrated on Figure 32.

5.2.1.2.2 CONCEPT MAPPING STRUCTURE.

This structure deals with the relationships existent among the loaded classes conforming the applet at runtime. Figure 33 shows jKSImapplet object associations upon execution.

In this case, an instance of *jKSImapplet* is shown as the startup object. This instance is designed to receive JavaScript events generated by widgets embedded in the HTML document. *jKSImapplet* objects are associated with one instance of the *jCMapPanel* class. This instance will receive events generated by the user as a result of scrollbar and mouse manipulation. This object will be associated with one instance of *jCMapCanvas* (which provides the concept mapping painting area), and one instance of *jKSImappletAWT* (which maintains the elicited concept mapping data). *jKSImappletAWT* is the applet counterpart of the *jKSImapperAWT* class found on jKSImapper. As before, the instance of *jKSImappletAWT* will hold zero or more objects derived from the *SRGraphic* abstract class (which can be nodes, links or context boxes); and a *CommandHandler* object to process instances from subclasses of the *AbstractCommand* class. As in jKSImapper, *SRGraphic* instances on jKSImapplet can be associated with one or more visual instances of subclasses derived from the *VisualGraphic* class.

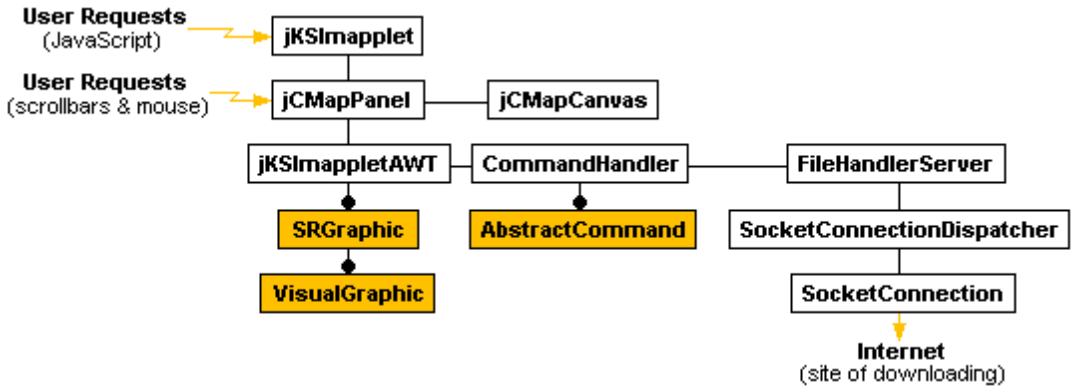


Figure 33. jKSImapplet Concept Mapping Structure diagram.

Differently from jKSImapper, jKSImapplet instances are not allowed to save information on the local computer, due to security constraints implemented by Java interpreters. This circumstance requires a remote connection with a jKSImapper Server process, which can

act as a data repository. Without such a connection, it would seem unreasonable to start a concept mapping elicitation if the data could not be stored or shared with other participants in a multi-user session (unless the concept map created is used for one-time construction purposes, such as the source to create static images). As a result, the *CommandHandler* instance will maintain one instance of the *FileHandlerServer* class, which will hold objects derived from the *SocketConnection* and *SocketConnectionDispatcher* classes. Such objects will be used to send and receive commands when interacting as part of a multi-user concept mapping elicitation session, and as a communication media to request storage of concept mapping data.

Additional security constraints restrict the behavior of such remote communications as well. In the case of the jKSImapper program, instances are allowed to establish communication with server processes running anywhere on the Internet. On the other hand, jKSImapplet applets are just allowed to communicate with the Web server from where the applet's classes were downloaded. This circumstance necessitates a remote server process on the same server where the classes are published, if concept mapping applets are to be of any use.

5.2.2 SERVER PROCESS.

The jKSImapper Server is a process executed by server computers. This program has been designed to receive and maintain network connections with client concept mapping programs to support multiple multi-user concept mapping elicitation sessions. Server processes are able to store concept mapping data on secondary storage. These tasks are achieved by using a collection of classes from the jCMap class hierarchy. These classes, and their relationship at runtime, are shown on Figure 34.

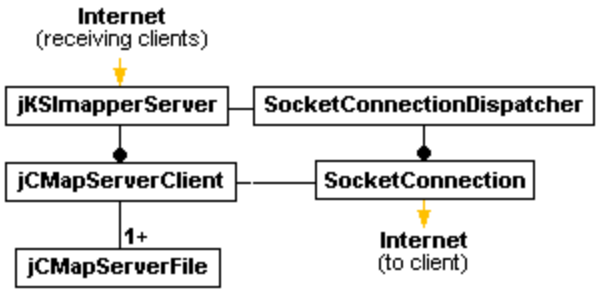


Figure 34. jKSImapper Server Structure diagram.

After starting execution, server processes are composed of instances of the *jKSImapperServer* and *SocketConnectionDispatcher* classes. *jKSImapperServer* objects implement a thread of execution that constantly listens over a specific socket port. This socket port is used by clients to request access to a concept mapping elicitation session. Once a client has requested access, the *jKSImapperServer* object will create a new *jCMapServerClient* instance, and will return to the listen mode for more incoming client connections. The task of this newly created object is to create and maintain a socket connection (using a *SocketConnection* instance) to be used for further communication with the client program. Individual socket connections will be created for each client requesting access to the server process. After requesting a connection with the server process, clients are the sole participants of a new elicitation session. Such session will be maintained by a new *jCMapServerFile* instance. After such session is initialized, the client has the choice to remain on this session (for starting the construction of a concept map), or to access a previously started elicitation session. In the case of requesting access to an existing session, a handle to the *jCMapServerFile* assigned to the requested session will replace the first *jCMapServerFile* instance assigned to the client. *jCMapServerFile* objects are designed to support individual concept mapping elicitation sessions. In order to do so, instances of this class will maintain a history list with all the commands generated by participants of the session. Additionally, they will support data structures to store the elicited concept map on a server file. Sessions on a server process can be identified either by the filename assigned for storage or, in the absence of a filename, by

using the name of the client that first created them. It is worth mentioning that duplicates might exist on the later case.

As depicted on the diagram shown on Figure 34, each one of the connections with clients will direct their requests to a single *SocketConnectionDispatcher* object. This object, which runs on an independent thread of execution, will be responsible for caching and processing incoming requests on an orderly manner. Methods implemented on this class will receive client requests and will process them using methods implemented on the *jKSImapperServer* class.

Clients connected to a server process have the ability to send and receive commands generated as part of a concept mapping session. Such commands are stored by the server process on the session's history list where the client is currently active. Commands are then broadcast to all of the participants in the session (including the sender) for execution. Currently implemented session-oriented commands were listed in Table 6, early on this Chapter.

In addition to session-oriented commands, clients can submit commands to request services provided by the server itself. Up to the current implementation, there are four operations performed by servers. Server-oriented tasks and commands are described as follows:

- ***Client requests to join a session***: This operation is issued by a client to request access either to a new, or to an existent concept mapping elicitation session on the server process. As mentioned on previous paragraphs, clients are automatically attached to a new session when they are first connected to the server. This session can be used for starting the elicitation of a new concept map from scratch. However, users might also prefer to work on a previously created elicitation session, or may want to participate in a session found on the server. To make such request, a "`GET`" command is implemented. This command executes one of five operations according to the parameter following the command name. These different variations of the "`GET`" command are issued in a

specific order to carry out a task. The sequence in which this request has to be handled is described as follows:

- "**GET *[*filename*]**": This text command is issued by a client requesting a change of session. The optional "*filename*" parameter indicates the name of the requested session. If the parameter is omitted, a new unnamed session is created. When submitting a filename as a parameter, one of the following two conditions can occur:
  - *The filename exists on the server*: First, the server will check if currently active sessions handle the requested filename. If such a session exists, the client's current session is replaced by the other existing one. On the other hand, if a session does not exist for the requested filename, a new session will be created.
  - *The filename does not exist on the server*: In this case, a new session is created and it is labeled with the submitted filename. This filename will be used to save the concept map, when requested.
- "**GET *START**": This command is sent to the client after the reception of a request for a change of session. Following this command will be the information stored on the server file requested, if any. This concept mapping information, which is expected to be MIME formatted, will be used by clients to reconstruct previously stored concept maps.
- "**GET *HISTORY**": This command marks the end of the information provided by the "GET *START" command, and the beginning of a set of commands stored on the history list for the requested session. It is possible that the history list in a session may not contain any commands if, for example, the session was recently created and no commands have been issued by participant elicitators. If commands do exist on the history list, they will be send to the client system for execution. These commands will complete the concept map constructed with the concept mapping data first submitted after the "GET *START" command was sent.

- "**GET *END**": This text command marks the end of the list of commands submitted by the "GET *HISTORY" command. After receiving this command, the client systems will have all the information required for constructing a consistent version of the concept map maintained by the server.

- ***Client requests to save a concept map***: Clients can use this operation for saving concept mapping information on a server file. This information can be saved under any filename, and not just to the currently active session. Save operations are invoked using the text command "**SAVE*[*filename*]**", where "*filename*" is an optional parameter that evokes the name of the file where the concept mapping information will be saved. If this parameter is not provided, the concept map will be saved using the filename assigned to the session. If no filename has been assigned to the session yet (e.g., the session was recently created) then the request will be overlooked. Following the initial command, the server will expect a stream of data containing the concept mapping data, which is supposed to be MIME formatted. This information will be followed by the closing command "**SAVE*END**", which will mark the end of the data entry, and the saving request. Additionally, if a session is open for the specified filename, the server will clean up any commands that might exist on the history list in order to be consistent with the state of the elicitation.

- ***Client informs that it is closing its communication with the server***: Clients can send the text command "**BYE ***" to the server process to inform that the currently active networking connection is no longer needed and it will be closed. Upon reception of this message, servers will clean up any objects that might be in use by the retreating client. However, sessions remain active if other clients are connected to it or if the history list contains unsaved commands.

- ***Client requests existing concept mapping files and active sessions***: Clients can inquire about files and sessions currently maintained on the server. Sending an "**ALL ***" text command to the server process, will result on the previously mentioned information to be transmitted back to the client. The information will contain the filenames assigned to

sessions or the name of the client that created a session, if a filename has not been given to it. Clients will use this information to display a selection list, from which the user might choose one of its elements as a target for an operation to be executed (e.g., joining a new session).

5.2.3 MIME File Format.

As a normal feature found on a variety of applications, users of jKSImapper and jKSImapplet are able to save their work on secondary storage for later use. As mentioned on previous sections, jKSImapper applications can store concept mapping data, either locally or remotely, while jKSImapplet applets are just allowed to store information on server computers. In the case of jKSImapper and jKSImapplet a new format was implemented (Kremer and Flores-Méndez, 1996). This format was designed to be compatible with the format published on the “Multipurpose Internet Mail Extension” document (Borenstein and Freed, 1993).

Figure 35 and Figure 36 show an example of a concept map and its resulting information when stored to a file.

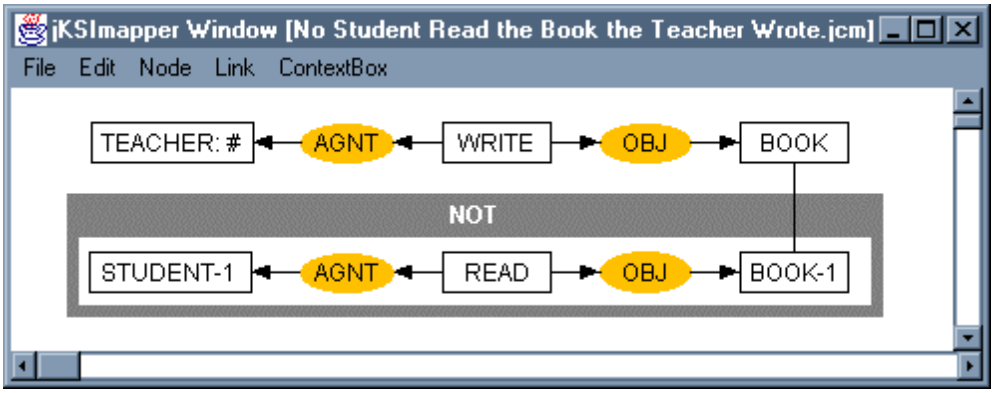


Figure 35. Conceptual Graph describing the sentence "No student read the book the teacher wrote".

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=****KSIDeLiMiTER****

--****KSIDeLiMiTER****
Content-Type: application/x-CMap; version=0.1

22
;
"ContextBox","ContextBoxShape",11,"NOT",30 51 408 61,-7f7f80,-7f7f80,-1,1,1,12
"SMaplet","LabeledConnector",16,"",394 44,2,0,4,1,1,0,-1000000,-1000000,-1000000,1,0,12
"SMaplet","LabeledConnector",12,"",347 25,2,0,4,1,8,1,-1000000,-1000000,-1000000,1,0,12
"SMaplet","LabeledConnector",13,"",277 25,2,0,8,1,5,1,-1000000,-1000000,-1000000,1,0,12
"SMaplet","LabeledConnector",14,"",212 25,2,0,9,1,5,1,-1000000,-1000000,-1000000,1,0,12
"SMaplet","LabeledConnector",15,"",140 25,2,0,6,1,9,1,-1000000,-1000000,-1000000,1,0,12
"SMaplet","LabeledConnector",17,"",279 90,2,0,7,1,2,1,-1000000,-1000000,-1000000,1,0,12
"SMaplet","LabeledConnector",18,"",348 90,2,0,1,1,7,1,-1000000,-1000000,-1000000,1,0,12
"Node","RectangleShape",6,"TEACHER: #",42 15 81 20,-1,-1000000,-1000000,1,0,12
"Node","RectangleShape",5,"WRITE",218 15 54 20,-1,-1000000,-1000000,1,0,12
"Node","RectangleShape",4,"BOOK",367 15 54 20,-1,-1000000,-1000000,1,0,12
"Node","RectangleShape",3,"STUDENT-1",41 80 81 20,-1,-1000000,-1000000,1,0,12
"Node","RectangleShape",2,"READ",218 80 54 20,-1,-1000000,-1000000,1,0,12
"Node","RectangleShape",1,"BOOK-1",367 80 54 20,-1,-1000000,-1000000,1,0,12
"Node","EllipseShape",10,"AGNT",147 81 46 19,-3800,-3800,-1000000,1,0,12
"Node","EllipseShape",8,"OBJ",297 16 45 19,-3800,-3800,-1000000,1,0,12
"Node","EllipseShape",7,"OBJ",297 80 45 20,-3800,-3800,-1000000,1,0,12
"SMaplet","LabeledConnector",20,"",139 90,2,0,3,1,10,1,-1000000,-1000000,-1000000,1,0,12
"SMaplet","LabeledConnector",19,"",210 90,2,0,10,1,2,1,-1000000,-1000000,-1000000,1,0,12
"Node","EllipseShape",9,"AGNT",147 16 46 19,-3800,-3800,-1000000,1,0,12
;

--****KSIDeLiMiTER****--
```

Figure 36. Data representing the concept map displayed on Figure 35

An interesting detail about this example is that the graphics displayed are sorted by their visual Z-order. This preserves the graphical precedence of overlapping graphics. This is exemplified on the link bridging from the nodes “BOOK” and “BOOK-1," which requires a dominant Z-order position over the “NOT” context box, in order to be shown as a continuos line.

Figure 37 shows the rules applied by jKSImapper and jKSImapplet when storing and reading concept mapping data. These rules, which were used on the previous example, are defined using an augmented Backus-Naur Form (BNF) notation (Crocker, 1982).

MIMEfile	=	MIMEheader *discardLine MIMEmultipartbody *discardLine
MIMEheader	=	MIMEversion *discardLine MIMEcontenttype
MIMEversion	=	“**MIME Version**” “**:**” Version
MIMEcontenttype	=	“**Content-Type**” “**:**” “**multipart/mixed**” “**;**” “**boundary**” “**=**” delimiter
MIMEmultipartbody	=	[delimiterInitial MIMEbodypart] delimiterFinal
MIMEbodypart	=	“**Content-Type**” “**:**” “**application/x-CMap**” “**;**” “**version**” “**=**” Version *discardLine CMap
CMap	=	CMapHeader CMapBody
CMapHeader	=	NextID CRLF “**;**” CRLF
CMapBody	=	*(Link / ContextBox / Node / HyperNode) “**;**” CRLF
Link	=	LinkAbstract “**,**” LinkVisual “**,**” GraphicID “**,**” LinkParameters CRLF
ContextBox	=	ContextBoxAbstract “**,**” ContextBoxVisual “**,**” GraphicID “**,**” NodeParameters CRLF
Node	=	NodeAbstract “**,**” NodeVisual “**,**” GraphicID “**,**” NodeParameters CRLF
HyperNode	=	HyperNodeAbstract “**,**” NodeVisual “**,**” GraphicID “**,**” NodeParameters “**,**” quotedString CRLF
LinkParameters	=	Label “**,**” Position “**,**” Arity “**,**” <Arity>(ArcNumber “**,**” GraphicID) “**,**” ArrowsFlag “**,**” LabelColor “**,**” LineColor “**,**” ArrowColor “**,**” FontType “**,**” FontStyle “**,**” FontSize ; *<Arity> is a reference to the “Arity” value previously read.*
NodeParameters	=	Label “**,**” Position “**,**” Dimension “**,**” FillColor “**,**” BorderColor “**,**” LabelColor “**,**” FontType “**,**” FontStyle “**,**” FontSize
NextID	=	integer
LinkAbstract	=	“**SMaplet**”

LinkVisual = "**LabeledConnector**"
ContextBoxAbstract = "**ContextBox**"
ContextBoxVisual = "**ContextBoxShape**"
NodeAbstract = "**Node**"
NodeVisual = "**RectangleShape**" / "**RoundRectangleShape**" / "**EllipseShape**"
HyperNodeAbstract = "**HyperNode**"
GraphicID = integer
Label = quotedString
Position = integer "**,**" integer
Dimension = integer "**,**" integer
Arity = integer
ArcNumber = integer
ArrowsFlag = integer
LabelColor = RGB
LineColor = RGB
ArrowColor = RGB
FillColor = RGB
BorderColor = RGB
FontType = integer
FontStyle = integer
FontSize = integer
Version = versionMajor "**.**" VersionMinor ["**.**" VersionRelease [[SPACE] "**(**" versionPlatform "**)**"]]
versionMajor = integer
versionMinor = integer
versionRelease = integer
versionPlatform = quotedString
delimiter = "******KSIDeLiMiTER******"
delimiterInitial = "**--**" delimiter CRLF
delimiterFinal = "**--**" delimiter "**--**" CRLF
discardLine = *CHAR CRLF
quotedString = "" *(ALPHA) ""
RGB = 6(DIGIT)

Figure 37. Backus-Naur Form notation for the jKSImapper MIME file format.

5.3 PORTING C++ CODE TO JAVA: LESSONS LEARNED

The work presented on this thesis was developed using a Java class library modeled on existing C++ code. Even when these programming languages share many characteristics, it is common to find features, implemented in Java that are not natural to C++, and viceversa.

As a result, porting the existing C++ class library to Java required facing issues derived from differences inherent to these languages. These issues will certainly be confronted by other porting projects as well. The issues addressed while implementing jKSImapper are documented in the following sections.

5.3.1 AUTOMATIC MEMORY MANAGEMENT.

C++ identifies objects by using pointers to their memory address. Pointers help to maintain high computational efficiency, since very efficient code can be produced by using them, but it is also true that most of the bugs injected into programs are because of their actual use (MacGuire, 1993). It is possible for C++ programmers to access objects that have not been initialized yet, or even worst, to access objects for which their memory space have been released and may already be in use by other objects. This latter type of error can go unnoticed during program execution and surface in later stages, making the source of the error very hard to detect and correct.

Java has eliminated this potential source of problems. Instead of using pointers, Java implements handles, or indirect references to objects located in the heap. Handles are controlled during program execution by an automatic memory manager, which is implemented as part of Java interpreters. One of the main tasks of the memory manager is to remove from memory any object that is no longer referenced by a variable in the program under execution. This process, which is known as garbage collection, is executed automatically by the memory manager when the interpreter is idle or when a request to allocate memory in a highly fragmented heap is not satisfied.

The implementation of a memory manager in Java helps to ameliorate, or avoid altogether, some problems that may arise on long-running C++ applications, such as (Cox, 1986):

- ***Memory fragmentation*:** It is possible for a program to run out of memory, even though there is plenty free space, because the available memory is heavily fragmented and there is not enough contiguous memory space for holding new objects requested.
- ***Memory leaks*:** It is possible to run out of memory because objects that are not needed were not properly released (or not released at all), resulting on memory space that will remain wasted.

Additionally, the combination of handles and a memory manager helps Java interpreters to detect attempts to access non-initialized objects and avoids the possibility of accessing an already released object, since objects in memory are only destroyed when no references to them are found in the program.

To avoid accessing non-initialized objects, Java enforces three states for object variables. These states are: non-initialized, initialized to `null`, and initialized to an object handle. These states are checked for validity at compile and execution time. Figure 38 shows code examples with variables on each different state, where (a) represents an attempt for invoking a method of a non-initialized variable (this effort will not compile); (b) illustrates a method call using a variable that has been initialized to *null* (this code will compile, but will throw a *NullPointerException* when invoking the object's method); and (c) shows a method invocation for a previously initialized object (which is the correct procedure to follow).

```
a) Object obj;
   obj.toString();

b) Object obj=null;
   obj.toString();

c) Object obj=new Object();
   obj.toString();
```

Figure 38. Java variable states.

In the case of the Java class library ported from C++ as part of the work developed for this thesis, having an automatic memory manager greatly simplified the efforts required to develop and test the concept mapping system implemented. This circumstance was enhanced by the existence of a well-defined behavior to detect variable access errors, which, in the same circumstances, may crash the computer or (worse) go unnoticed under C++ programs.

5.3.2 Portability and Graphical User Interfaces.

Two characteristics that are found on C++ are that it can handle computer internals and does not define mechanisms to manipulate user interfaces. In latter case, C++ frees developers to create libraries to handle the user interface of choice, resulting in operating system (and even vendor) specific libraries for manipulating user interfaces (e.g., in the case of MS-Windows, Borland has developed the Object Windows Library while Microsoft has promoted the Microsoft Foundation Classes). Together, these C++ characteristics (access to computer internals and absence of policies to unify graphic user interfaces) make it difficult to design programs that are portable among computer platforms.

In contrast to C++, Java implements a set of classes to support graphical user elements common to diverse operating systems, and restricts programmers from accessing computer internals. These features, added to the fact that Java explicitly defines the structure of primitive data types (see Section “3.1.3. Portability”), contribute to the

production of portable programs. Unfortunately, the mechanisms implemented on Java to manipulate graphic user interfaces are restrictive and immature if compared with current C++ implementations. In the case of version 1.0.2 of the Java Development Kit, Java does not implement elements that are common to graphical environments, such as pop-up menus and modal dialog boxes, for example.

jKSImapper would benefit from the use of pop-up menus to support the modification of graphical attributes on objects from a concept map; this could be achieved without requiring the selection of an object and then selecting options from the main menu, as it is done under the current version. Additional benefits that may result from the use of pop-up menus include displaying just the operations applicable to the selected object, thus providing a visual identification of the actions supported by each particular object. The original C++ library from which jKSImapper was modeled does implement pop-up menus; however, this facility had to be removed due to the absence of support by the Java language.

Java does not properly support modal dialogs. Under normal circumstances, the execution of a modal dialog will result in a dialog being displayed, while blocking user input to other windows until it is dismissed. After the dialog's dismissal, the code will continue its normal thread of execution with the next operation following the instruction that requested the modal dialog. This circumstance allows one to check information provided on the dialog and to react to those inputs just after the dialog's dismissal. Unfortunately, such behavior is not properly followed in Java. When a modal dialog is invoked, input interaction is indeed restricted to the dialog, but the thread of execution does not wait for its dismissal to continue running the program. This circumstance eliminates the possibility of including code for analyzing the input generated by the user immediately after closing the dialog. Such a scheme forces the development of modal dialogs having some "knowledge" of the task to perform after receiving user's input.

Several techniques can be implemented for handling such input, each with different levels of complexity. The most common, and easy to implement, technique requires dialogs to

maintain a reference to an object responsible to perform the analysis of the input. In other words, the dialog will know about one or more objects (and one or several methods part of such objects), which will be responsible for processing input resulting from user interaction (note that the dialog itself can be the object responsible for handling the input; however, dialogs that perform tasks in isolation are not common, and usually do not perform meaningful operations). This approach has the disadvantage of increasing the complexity of code and increasing the dialogs' level of coupling with other classes, hurting reusability.

It would be interesting to observe one exception to the rule for the behavior of modal dialogs on Java: Instances of the class *FileDialog* (used for selecting disk files), does act according to the behavior expected for a modal dialog (this is, stopping code execution on the instruction invoking the dialog, and resuming execution just after such instruction, when the dialog is closed). The reason behind this different behavior may be that file dialogs are directly provided and supported by native windowing environments, and not by the language itself.

5.3.3 CLASS INHERITANCE AND INTERFACES.

Another relevant difference between C++ and Java is the absence of multiple inheritance for classes on the latter language. Instead, Java relies on a limited form of multiple inheritance that is achieved using by interfaces. An interface is an abstract class that just declares methods and does not implement any code at all. Classes are allowed to be inherited from zero or more interfaces, but from exactly one class at a time (all Java classes have the *Object* class as a superclass).

In the case of the C++ class library ported, the absence of multiple inheritance on Java represented a problem, since several C++ classes were implemented using multiple parent classes. Fortunately, multiple inheritance can be simulated by making one class an attribute or associate of another class. This way, one object can invoke the desired functions of another class, using delegation rather than inheritance. Delegation consists of

catching an operation on one object and sending it to another object that is part of, or related to, the first object (Rumbaugh, Blaha, Premerlani, Eddy and Lorensen, 1991). However, implementing delegation increases the amount of code (and possible errors) on the class that encompasses other classes, since methods have to be implemented to redirect invocations to the appropriate methods on the associated objects.

5.3.4 GENERIC PROGRAMMING.

Templates are used in C++ to construct structures that can be reused employing different data types, such as container classes. Templates are particularly helpful to enforce type correctness at compile time. However, Java does not provide such functionality. Instead, it relies on object-based containers, which do not have the mechanisms to enforce specific types on the elements they store, a concept known as type safety. Since containers implemented in Java manipulate objects of the *Object* type (all objects have this class as a supertype), it is possible to insert into containers elements that are not of the type expected by the application. The compiler and interpreter are not capable of catching such errors. Under this circumstance, problems are likely to arise when retrieving elements erroneously inserted, since these objects will not behave as expected. The only error that might be triggered is if a fetched object is cast to an incompatible type. This action will generate a *ClassCastException* exception. Unfortunately, knowing of an error when retrieving and using a contained element does not help to identify the circumstances on which the incompatible object was inserted. This scenario is similar to trying to uncover errors caused by memory access errors using C++ pointers.

Additional disadvantages of using Java containers are (Myers, Bank and Liskov, 1997):

- When new elements are added to a Java container, if they are of a primitive type such as *int*, then it is necessary to objectify them. For example, *int* variables must be explicitly wrapped in an *Integer* objects to make them usable as *Object* instances. This wrapping step is awkward for the programmer and has runtime overhead.

- Whenever an element is fetched from a container, it must be explicitly cast from *Object* to the expected type. If the element has a primitive type, it must be also unwrapped after the cast, adding even more cost and coding complexity. However, casting is not a substitute

The problems described above are ameliorated in C++ by using templates. When elements are added to a container, there is no need to objectify them, and when they are retrieved, there is no need for the expensive runtime cast or unwrapping, and it will always be the assurance that the elements handled are of the correct type.

5.3.5 PARAMETER-PASSING.

C++ supports two ways of passing parameters to methods and functions: by reference, and by value. When passing a parameter by value, a copy of the original data is submitted to the function. This option allows modifications on the copy without altering the value of the original variable. On the other hand, passing a parameter by reference implies the use of an alias, which is a reference to the memory address where the original variable is located. Using this technique, modifications on the variable passed to the function, will result on modifications of the original value as well.

In the case of Java, parameter-passing is only performed by value. Such implementation represents a limitation, since there is no straightforward mechanism to return new objects within a method besides the return value of the method itself. Alternative approaches to overcome this limitation are, to create wrapper objects for returning new instances, or using public variables as temporary receptacles for newly created instances.

5.3.6 MULTI-THREADING.

Many environments have what it is called multi-tasking in the operating system, which is different from multi-threading. Under multi-tasking environments, tasks are known as heavy-weight processes; under multi-threading environments, tasks are known as light-weight processes. The difference is that heavy-weight processes are contained in

separated address spaces, and should be considered as different programs running concurrently under the control of the operating system. On the other hand, light-weight processes, or threads, share the same address space and cooperatively share the same heavy-weight process.

Programs from different languages can be executed as heavy-weight processes. However, not all languages (including C++) introduce mechanisms to describe a clean way to deal with light-weight processes. In contrast, Java was designed as a multi-threaded language from scratch. It provides mechanisms for controlling the execution of threads and their synchronization.

In the case of the jCMap class library, threads were implemented to improve the performance of network socket communications, and to coordinate the execution and broadcasting of commands in the server process.

5.3.7 WORLD WIDE WEB INTEGRATION.

Java and C++ present different approaches to integrate programs to the Web. In the case of C++, programs are required to implement functions that communicate and interact with browsers. Programs implementing such interface are called plug-ins. The primary goal of the plug-in interface is to allow an existing platform-dependent program to seamlessly integrate with browsers to take advantage of the networking and hypermedia capabilities provided by the Web. An example of a plug-in interface is found on the Netscape's plug-in Application Program Interface (Netscape, 1996d.) This API declares functions that C and C++ programs are required to implement and that Netscape will access at runtime. Reciprocally, Netscape offers functions for the plug-ins to invoke during execution, thus, allowing an interaction process between the programs and the browser.

Previous efforts at the Knowledge Science Institute have lead to the implementation of the NPKSIMapper plug-in, which is the Netscape plug-in version of the C++ KSIMapper concept mapping tool (Kremer, 1996).

Due to their inherent Web nature, Java applets do not follow the rules applied to plug-ins. Instead, applets can be freely downloaded and executed in any Java-aware client browser, but security considerations dictate that most user resources should not be available to the imported applet. Limitations imposed on applets by Web browsers are, for example, to invalidate any efforts to open network connections to servers other that the host where the applet was downloaded, and to restrict access to client storage devices.

In contrast to Java applets, plug-ins can provide higher efficiency of code and seamlessly access to local and remote resources, but they are restricted to a specific operating system and by the proprietary characteristics of the plug-in API. These circumstances make plug-in applications virtually non-portable among browsers and operating systems.

5.4 CHAPTER SUMMARY.

The material presented on this chapter provides valuable information to understand the composition of the class library implemented as part of the present thesis. Additionally, this chapter presents the runtime structures conformed by classes from the library, and an analysis on the main topics addressed and learned from translating C++ code to Java.

The jCMap is a class library composed of more than 60 Java object-oriented classes. This class library was used to develop the jKSImapper standalone application, the jKSImapplet and jKSImapplet Navigator applets, and the jKSImapper Server process.

These applications allow multi-user concept mapping elicitation on distributed and multimedia environments supported by the Internet and the Web. In this section, classes in jCMap were described as components organized under six task-specific groups: Behavioural Graphic, Visual Graphic, User Interface, Command Handling, Networking and Server, and File Storage. These groups, as well as their member classes, were covered on detail during the first part of this chapter.

The principal goal for developing the classes discussed on this chapter is to construct programs that can satisfy concept mapping needs. To this end, client systems were complemented with the development of a server process that supports multi-user

elicitation. The second part of this chapter was devoted to describing such runtime structures, which are composed of jKSImapper and jKSImapplet as clients, and the jKSImapper Server as the centralized server process. Additionally, this section was used to explain the structure of the file format followed by client systems to transform and store concept mapping information.

The third part of the present chapter discussed the issues confronted as part of the effort to port the existing C++ class library to Java. Main differences between these languages include the presence of an automatic memory manager, support for light-weight threads, and graphical user interfaces as integral part of the Java language. Additional distinctions include issues such as parameter-passing to methods and the absence of multiple inheritance and templates; all of them are supported by C++, but not by Java. Another relevant difference is found in the mechanisms used by Java and C++ programs to integrate to Web browsers, one in the form of applets and the other as plug-ins, respectively.

The following and last chapter of this thesis will discuss issues concerning future research and improvements for the concept mapping systems implemented as part of this work. This chapter will also evaluate the objectives for the present research and will draw the conclusions of this thesis.

CHAPTER 6

EVALUATION AND FUTURE DEVELOPMENT

In summarizing the work set forth in this thesis, it is useful to analyze the original stated aim for the present research. As presented at the beginning of Chapter 1, the goal of this research was to evaluate Java as a suitable programming language for the Internet by using it to implement a concept mapping system able to support distributed user environments on the Internet and the World Wide Web.

To achieve such goal, the Java programming language was thoroughly analyzed and used to develop a concept mapping system that can work as a standalone application or as an applet inside Web browsers. These programs, which are named jKSImapper and jKSImapplet respectively, are designed to perform as elicitation agents that interact with server processes. These server processes will perform as centralized coordinators and data repositories. In this context, the Web acts as a hypermedia integrator, and the Internet as the transport medium.

6.1 EVALUATION.

Seven objectives were defined for this research. These objectives, which were listed in Section 1.5 in Chapter 1, will be revisited and analyzed in the following sections as a guide to evaluate the work presented on this thesis.

6.1.1 REQUIREMENTS.

The first objective was:

To survey the requirements for developing programs on the Internet and the World Wide Web, as well as the state of the art of programming languages that can be used to implement such programs.

This objective was met in Chapter 1 and Chapter 2. Chapter 1 addressed general concepts related to client/server systems, the Internet and the World Wide Web. This chapter also introduced the notion of executable code on the Web, and identified portability, security and functionality as requisites to well-behaved Web programs. These issues (portability, security and functionality) were addressed in detail on Chapter 2, where they were used to analyze Java and Microsoft's ActiveX, which are the foremost techniques to integrate downloadable code to the Web.

6.1.2 BACKGROUND.

The second objective was:

To analyze the features of Java as a programming language for the Internet and the World Wide Web.

This objective was answered on Chapter 3, where Java was defined by describing the main attributes of the language. Such attributes depicted Java as an object-oriented, distributed, portable, secure and multi-threaded programming language. Additional topics covered on this chapter were the facilities provided by the language to effortlessly support Internet communication, and the integration of applets to Java-aware Web browsers, such as Netscape's Navigator.

6.1.3 CURRENT WORK IN THE FIELD.

The third objective was:

To analyze the implementation requirements for a concept mapping tool to operate on the Internet and the World Wide Web based on previous developments at the Knowledge Science Institute.

Chapter 4 addressed this objective by describing the implementation requirements essential to the development of a concept mapping tool aimed at a multi-user environment. The first part of this chapter briefly introduced concept mapping tools previously developed at the Knowledge Science Institute. The experience gained after developing such tools greatly contributed to the definition of a set of requirements on which to base the implementation of the present concept mapping programs.

On the other hand, requirements for a multi-user system were defined according to the Computer Supported Collaborative Work interaction modes for a community of users working on a common project. Such requirements represented guidelines for the development of multi-user systems on distributed environments, such as the Internet and the Web.

6.1.4 IMPLEMENTATION.

The fourth objective was:

To design and develop a well-structured implementation of a Java concept mapping tool based on an existing system constructed using the C++ programming language.

This objective was met in Chapter 4 and Chapter 5. The former chapter detailed graphical elements and commands available to the user interacting with jKSImapper and jKSImapplet. Chapter 5 went deeper on the subject and analyzed each of the classes constituting the Java jCMap class library. This library, which is a pivotal part of this research, was derived from a previously developed C++ library, called CMap. The design goal for jCMap is to support the operation of jKSImapper, jKSImapplet, and the jKSImapper server process.

6.1.5 DEMONSTRATION.

The fifth and sixth objectives were:

To compare the Java and C++ programming languages at the light of the implementation experience, and

To evaluate the Java concept mapping tool in a range of practical applications.

The fifth objective was covered in Chapter 5, Section 5.3. This chapter addressed specific issues that were confronted, and will certainly be confronted by other projects, when porting C++ code to Java. Such issues, which reflect the differences existing between these programming languages, include topics ranging from multi-threading and parameterized types, to dynamic memory access models and program integration with the World-Wide Web.

The sixth objective, to evaluate jKSImapper and jKSImapplet in a range of practical applications, was met through the many examples displayed on this research, which are mentioned as follows:

- Figure 8 (pg. 50), from Chapter 4, where jKSImapper was used as a general purpose concept mapping tool to represent a hierarchical structure of systems.
- Figure 11 (pg. 62) and Figure 17 (pg. 72), from Chapter 4, Figure 35 (pg. 104), from Chapter 5, and Figure 39 (pg. 122), from the current chapter, where jKSImapper and jKSImapplet were used to represent Sowa's conceptual graphs for the sentences "*Tom believes that Marry wants to marry a sailor*," "*No Student read the Book the Teacher Wrote*," and "*Dickson went to Calgary by plane*," respectively.
- Figure 18 (pg. 73), from Chapter 4, where an instance of jKSImapplet Navigator is used as a Web navigational tool. Concept maps handled by this applet were used as graphical indexes to access Internet resources (in this case, an HTML document containing another concept map).
- Figure 27 (pg. 92), from Chapter 5, where jKSImapplet and jKSImapper are shown collaborating on a multi-user setting to elicit a concept map.

6.1.6 FUTURE WORK.

The seventh objective was:

To propose further development and research based on the experience and evaluation of the concept mapping tool implemented.

jKSImapper cannot be considered a final or complete product. Instead, it can be labeled as the first comprehensive layer of functionality for the development of domain-specific concept mapping systems. As explained on Section 6.2 below, the functionality provided by jKSImapper can be enhanced in two non-exclusive ways: by extension, or by improvement.

6.2 AREAS OF FUTURE DEVELOPMENT.

This section is an account of the future development issues for jKSImapper. As explained through this work, this implementation provides a well-defined functionality to handle concept maps. Consequently, future developments are aimed to enhance such functionality, either by improving the techniques applied to their inner mechanisms, or by extending them with new domain-specific functionality.

6.2.1 EXTENDING FUNCTIONALITY.

The extension of functionality implies the specialization of existing functionality to perform under domain-specific constraints. This specialization is achieved by adding classes and methods to support operations specific to a community of users.

This section will propose viable extensions to the Java systems presented on this thesis. It is important to bear in mind that, since these systems provide a generic solution for concept mapping elicitation on the Internet and the Web, the suggestions made on following sections are exclusive to the perspective of the author, and they do not represent limitations on the application of the systems to other domains.

As explained below, two different extensions are proposed: one, to support graphical elicitation of conceptual graphs; and, second, to integrate jKSImapper as a component for the Habanero Environment, which is a Java groupware system developed by the National Center of Supercomputing Applications (NCSA).

6.2.1.1 CONCEPTUAL GRAPHS.

Conceptual graphs are defined in the literature as finite, connected, bipartite graphs (Sowa, 1984). They are described as finite, since users' memory is bound to retain just a finite number of concepts and conceptual relations. They are depicted as connected, because two parts that are not connected would represent two independent graphs. Finally, they are defined as bipartite, since there are two distinctive types of nodes, concepts and conceptual relations, where every arc links a node of one kind to a node of the other kind.

Conceptual graphs can be denoted using three different representation types. Such representations will be described and illustrated using the sentence "*Dickson went to Calgary by plane*" (Lukose, 1996):

- *Diagrammatic form*: In diagrams, concepts are drawn as boxes, conceptual relations as circles, and arcs as arrowed links connecting boxes and circles. This representation is illustrated in Figure 39.

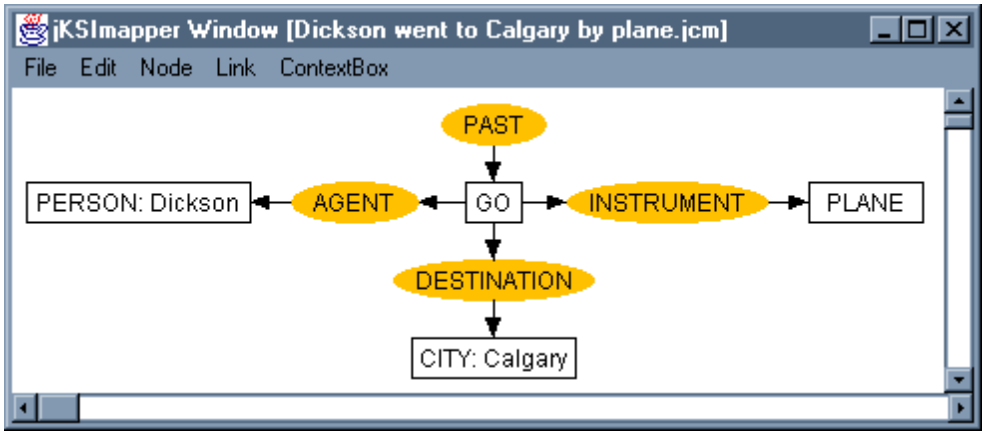


Figure 39. Example of a Conceptual Graph as a Diagram.

- *Linear form*: Linear form is a text-based representation for expressing conceptual graphs. Boxes drawn for concepts (under the diagrammatic representation) are abbreviated to square brackets, and circles are represented as rounded parenthesis. As a

result, the same conceptual graph depicted in Figure 39 can be depicted as shown on Figure 40.

```
[GO] -  -> (AGENT) -> [PERSON: Dickson]
        -> (INSTRUMENT) -> [PLANE]
        -> (DESTINATION) -> [CITY: Calgary]
        <- (PAST)
```

Figure 40. Example of a Conceptual Graph in Linear Form.

- *First-order logic*: Conceptual graphs can also be represented using first-order logic. This representation is exemplified in Figure 41, where the sentence "*Dickson went to Calgary by plane*" is used, as in the previous examples.

```
∃x ∃y  ( GO(x) ^ PERSON(Dickson) ^ PLANE(y) ^ CITY(Calgary) ^
         AGENT(x, Dickson) ^ INSTRUMENT(x, y) ^
         DESTINATION(x, Calgary) ^ PAST(x)
       )
```

Figure 41. Example of a Conceptual Graph as a First-order logical formula.

From the representations mentioned above, diagrammatic representations are (arguably) the easiest to recognize and interpret by humans. As illustrated in Figure 39, jKSImapper has the ability to graphically depict conceptual graphs as diagrams. However, due to its non-specialized nature, the system does not provide any assistance (and it does not enforce any constraints) for the elicitation of formal graphs. As a result, extending the functionality of jCMap to handle conceptual graphs is an area of future development that can demonstrate the usefulness of the system when applied to a specific domain.

Another example of the application of jKSImapper and jKSImapplet is found on current collaboration efforts undertaken between the Knowledge Science Institute and the Department of Mathematics, Statistics and Computer Science at the University of New England, Australia. Such efforts may result on the unification of systems into a distributed environment for the elicitation of centralized knowledge bases. In this context, jKSImapper and jKSImapplet will act as client elicitation agents to create and edit knowledge structures maintained by an Extendible Graph Processor system (Garner, Tsui, Lui, Lukose and Koh, 1992) performing as a server process. Future developments may

also extend the functionality of these systems to allow the translation of conceptual graphs into alternate forms of knowledge representation.

6.2.1.2 THE HABANERO ENVIRONMENT.

The Habanero Environment (NCSA, 1997) is a groupware framework aimed to support multi-user Java systems that share resources over the Internet.

Habanero is defined a client/server system composed of a server process coordinating client systems, each supporting a collection of Java applications. Client applications currently shipped with Habanero include multi-user versions of a whiteboard program, a text editor, and an audio chat for voice communication. To support such systems, Habanero implements networking and synchronization mechanisms that allow to share state, data and events generated by users manipulating client software systems.

To maintain a consistent state among client applications, Habanero implements mechanisms that allow clients to route user events to a server, which will broadcast such events to all the participants on a session. Since such runtime structure is similar to the one implemented on jKSImapper, it is feasible to adapt and extend the present system to operate as a Habanero client application. This circumstance will help to increase the awareness of concept maps as collaborative tools, and will allow jKSImapper to be under the scrutiny of different communities of users, some of which might find it suitable for extension on their particular domains.

6.2.2 IMPROVING FUNCTIONALITY.

To improve the functionality of a system means to enhance the mechanisms supporting the operation of such system. In the case of the developments presented on this work, such enhancements are achieved by adding and refining algorithms and methods to increase their performance and usability.

The following sections have been included as a critique of the limits, weaknesses and imperfections detected on the implementations described on this work. Proposed

enhancements will be addressed under three different sections: Human-Computer interfaces, Multi-user issues, and Miscellaneous improvements.

6.2.2.1 HUMAN-COMPUTER INTERFACE.

Human-Computer interfaces are generally described as information channels that allow users and computers to communicate (Lewis and Rieman, 1994). Usually, these communication channels on a system are composed of menus, windows, the keyboard and the mouse.

One method to discover the strengths and weakness of a user interface, is to count the number of keystrokes and mental operations (decisions) required for the tasks the user intends to achieve. This will enable estimation of task times and identify tasks that take too many steps.

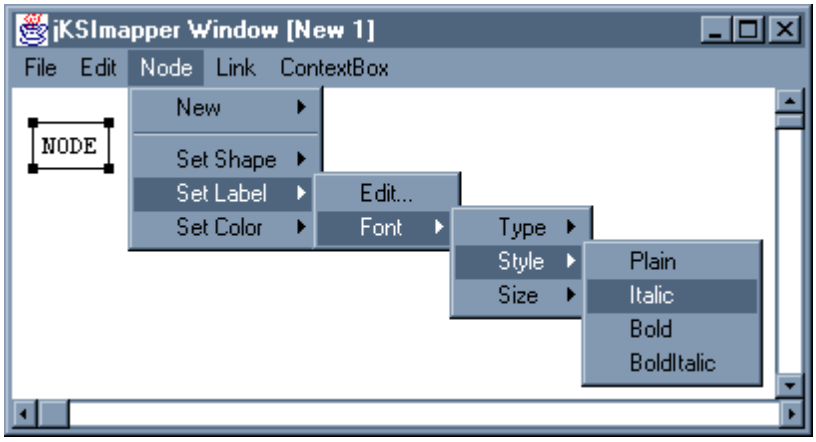


Figure 42. Font style modification using menu commands.

In the case of jKSImapper, frequently used operations include creation, edition and manipulation of graphical objects. From these operations, modifying an object's location is the less step-consuming task, since this operation is performed just by clicking on the target and dragging it to a new location. On the other hand, creation of objects and the edition of their attributes are the most step-consuming operations, since they rely on commands nested under several sub-menu levels. Of existing operations, the modification of an element's font style represents one of the most inefficient operations found on the

system. As illustrated on Figure 42, this task is achieved by selecting the object to modify (e.g., a node), and by traversing a group of sub-menus until reaching the desired font style to change. As a result, a total of six decisions are required to successfully complete this command. Under human-computer ergonomics, such number of decisions reflects a poorly designed interface.

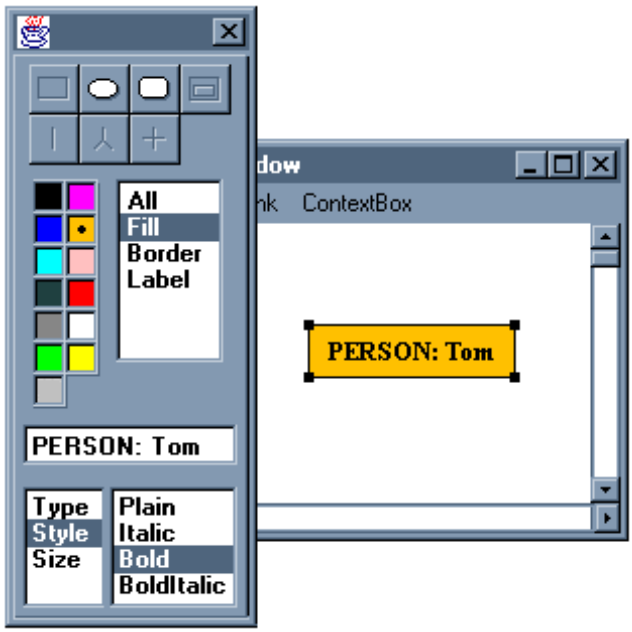


Figure 43. jKSImapper Toolbar.

At this point, it is true that a better solution for selecting commands needs to be provided. One feasible answer to this problem is achieved by implementing a toolbar allowing fast selection of frequently used commands. Such a toolbar may be organized as illustrated on Figure 43. In this figure, a jKSImapper toolbar is shown as an independent window, but it can also be implemented as part of the display area of the system.

Functions provided by the toolbar may allow the creation and modification of objects. This dual functionality will be defined by the existence (or the absence) of selected elements on an elicitation.

If multiple objects are selected, the toolbar will reflect the common denominator values for attributes of those objects. In the case of the example shown on Figure 43, a

rectangular node labeled “PERSON: Tom” is selected. As a result, the node’s characteristics are reflected on the toolbar’s controls, which can be used to modify the attributes of the node. In this case, buttons located at the top of the toolbar can be used to change the shape of the node to ellipse or rounded rectangle. In the case of this example, the remaining buttons are shaded since they do not reflect options applicable to the selected object. Additional controls included on the toolbar are: a color palette that works in conjunction of a list box to modify color attributes on the targeted object; an input line for editing the label of the object; and a pair of list boxes that are used to select the selected element’s font attributes and their values, respectively.

The toolbar can be used to create new elements if no objects are found to be selected. In this case, new objects will have their attributes initialized to the values provided by the controls in the toolbar. This technique can speed the creation of multiple objects using non-default values.

However, the toolbar approach is not suggested as the ultimate solution for human-computer interaction, and it certainly contains weaknesses. One of such limitations is found on the creation of new elements while existing objects are selected. This circumstance represents a problem, since the toolbar presented does not accommodate (yet) the functionality to switch states from modification of selected items to creation of new items.

Additional enhancements to the user interface can be achieved by implementing pop-up menus and in-place edition of labels. These techniques can make explicit the operations available for an object (in the case of pop-ups) and substantially simplify the modification of objects’ attributes. The implementation of these approaches is not defined in this thesis, and they are left for future development.

6.2.2.2 MULTI-USER ISSUES.

Groupware awareness is another topic that requires improvement. Groupware awareness is defined as the dynamic knowledge of changing environments, which are updated and maintained as the environments evolve (Gutwin, Greenberg and Roseman, 1996).

There are features that can be used to provide awareness on multi-user environments. For example, there are mechanisms to identify the members collaborating on a multi-user session. However, techniques vary from displaying a list box listing the names of the participants, to the implementation of individual cursors for each participant and global viewers displaying the location of each participant under the entire domain of the session. The suitability of such techniques depends on the context on which the target application will be used.

In general, groupware techniques allow participants to be aware of the presence of other participants (e.g., by using global viewers displaying each member's cursor); they allow participants to be aware of the operations performed by each collaborator (e.g., using authorship and concurrency control); and may allow communication between members of the session (e.g., by using text areas for message exchange, or using voice transmission processes). However, as desirable as these functions may be, the implementation of some of them may require high bandwidth network connections. In the case of jKSImapper and jKSImapplet, such networking power may not be available, since these applications were designed to work on the Internet, which is a networking environment that does not make any assumptions with regard to speed and reliability.

The identification of participants and concurrency control are issues that need to be implemented on jKSImapper. Identification of users can be achieved by maintaining a list of members joining the server. This information can be made available to all the clients for display (e.g., using a list form). On the other hand, concurrency control can be achieved by using centralized locks for displayed objects. Such locks can be maintained by the server, and will be represented on client systems by using visual feedback (e.g., using colors or patterns to indicate granted locks).

6.2.2.3 MISCELLANEOUS IMPROVEMENTS.

Miscellaneous improvements encompass enhancements that do not fit under previous sections. Up to the moment of writing, two topics have been identified as candidates to such improvements. These topics are:

- *Paint by clipping*: Every program that interacts with a user has to implement mechanisms to display the state of the information handled by the system. Under normal circumstances, such state will evolve as a result of the operations performed by the user. Therefore, systems will also need to implement methods to update the information that is displayed to reflect those changes. In short, two mechanisms have to be implemented on interactive systems: one, to display the current state of the information; and another one, to update the information that is displayed to reflect changes resulting from user interaction.

In the case of jKSImapper and jKSImapplet, the model described above was supported. Thus, methods were implemented to allow the painting of the graphs maintained by the system, and to allow the updating of such graphs in response to user events.

However, from these methods, the updating mechanism is not implemented efficiently in the systems presented, since they just trigger the painting of all the displayed objects after the execution of each command generated. This deficiency is evident as the complexity of a concept mapping elicitation grows. For example, in a concept map composed of one hundred objects, the movement of one of those elements will require the painting of the remaining ninety-nine objects, regardless of whether such movement affects or not their visual representation.

A solution to this problem can be found by implementing a technique called paint clipping. This technique is based on the delimitation of display areas affected by executed commands, and restricting the updating only to those objects displayed within such affected areas. In practice, the algorithms implementing this technique will impose greater processing loads to small concept maps, if compared with the current

implemented algorithm; however, this technique will increase the painting speed as the number of elements on the elicitation increase.

- *Copy and paste*: Copy and paste are functions based on a temporary buffer used as a container of information. Copy reflects the act of storing information on the buffer, and paste represents the action of retrieving information from the buffer. This buffer can be used by a single system to duplicate existing information, or by multiple systems to share compatible information.

  In the case of the concept mapping systems described on this thesis, the implementation of a copy and paste mechanism may allow one to copy selected objects to a buffer, from where they can be retrieved by the same or by a different elicitation session, to duplicate concept map segments, or to copy concept map segments between sessions, respectively.

  Buffers can be implemented global to the operating system or local to the application. The first case is supported by most operating systems, which provide a buffer that is shared by all the applications been executed. On the other hand, local buffers are implemented for exclusive use of an application. Due to the portability characteristics of Java, it is not clear at this point whether access to system buffers will be supported or not. This circumstance will lead to the need of implementing local buffers for the concept mapping systems described on this work.

6.3 THESIS SUMMARY AND CONCLUSION.

The goal of this research was to evaluate Java as a suitable programming language for the Internet by using it to implement a concept mapping tool system able to support distributed user environments on the Internet and the World Wide Web.

To accomplish such task, the presented research introduced, during early stages, the notion of concept maps, followed by an overview of the Internet and the World Wide Web. The Web was described as a hypermedia environment capable of presenting Internet resources under a multimedia interface. From such resources, executable code

was identified as the most versatile resource available, due to its ability to provide complex services as a response to users' interaction.

At this point, leading techniques that provide downloadable executable code on the Web were identified and analyzed based on their capabilities to support portability, security and functionality issues. Techniques evaluated were Java and ActiveX.

This evaluation was followed by a description of the characteristics found in the Java programming language, including an overview of the networking features that have positioned Java as a suitable programming language for the Internet and the World Wide Web. This discussion was followed by a comparison between Java and C++ at the language level.

The leading step towards production is based on a thorough understanding of the tools to use. In the case of this research, its first part was devoted to understand the features of the tool, which is the Java programming language, and the context of its use, which is the Internet and the World Wide Web. Once these topics were covered, the production of the system was started. As stated in the aim of this work, a concept mapping tool was to be implemented to support multi-user concept mapping collaboration on the Internet and the Web.

The process of producing such tool required the study of previous concept mapping developments as a guidance to the definition of the requirements for the system. Such requirements were defined based on levels used for the analysis of concept maps (abstract, visual and discourse perspectives), and on the functionality that the system has to provide to support multi-user environments (based on the CSCW interaction modes). The definition of the requirements for the tool was followed by an overview of the functionality implemented on the systems developed, which were named jKSImapper and jKSImapplet. These systems perform as Java concept mapping tools for the Internet and the Web, respectively.

The last part of this research was devoted to describe in detail the jCMap class hierarchy, the runtime system architecture for the systems implemented, and the lessons learned

while porting C++ code to Java. Class hierarchies are structures composed of defined entities ordered to reflect the specialization of state and behavior. In the case of the this research, jCMap was presented as a class hierarchy composed of more than 60 classes designed to support jKSImapper and jKSImapplet. The operations implemented on these classes were described according to the areas on which they perform (which are Behavioural Graphic, Visual Graphic, User Interface, Command Handling, Networking and Server, and File Storage).

However, class libraries are not a suitable reflection of the relationships existing among objects at runtime. Thus, a description was required of how the objects collaborate to provide the functionality expected for the systems implemented. Such relationships between components in a program define the runtime system architecture for the system. In the case of this research, three architectures were presented; one for jKSImapper, one for jKSImapplet, and a third one for the server process.

This discussion was followed by a description of the lessons learned while porting C++ code to Java, and a section describing areas of future development.

Overall, this thesis has presented a usable implementation of a client/server concept mapping tool developed using the Java programming language. This tool can perform as a standalone application or as an applet embedded on Netscape browsers. Such developments, which are named jKSImapper and jKSImapplet, have been demonstrated to support the elicitation of concept maps on single and multi-user collaborative environments under the Internet and the World Wide Web.

REFERENCES

Adobe Systems, Inc., (Adobe, 1985), “PostScript Language Reference Manual," Adobe Systems, Inc., Addison-Wesley, 1985.

Axelrod, R. (Axelrod, 1976). “Structure of Decision,” Princeton, New Jersey: Princeton University Press. 1976.

Ball, S. (Ball, 1996), “SurfIt! - A WWW Browser,” Cooperative Research Center for Advanced Computational Systems, Australian National University, 1996. Available at: http://pastime.anu.edu.au/SurfIt/

Berners-Lee, T., Cailliau, R., Frystyk, H., and Secret, A. (Berners-Lee, Cailliau, Frystyk, Secret, 1994), “The World Wide Web,” Communications of the ACM, August, 1994.

Berners-Lee, T., Masinter, L., and McCahill, L. (Berners-Lee, Masinter and McCahill, 1994), “Uniform Resource Locators (URL),” December 1994. Available at: http://sunsite.auc.dk/RFC/rfc/rfc1738.html

Berners-Lee, T., and Connolly, D. (Berners-Lee and Connolly, 1995), “Hypertext Markup Language 2.0,” November 1995. Available at: http://sunsite.auc.dk/RFC/rfc/rfc1866.html

Booch, G., (Booch, 1994). "Object Oriented Analysis and Design with Applications", Addison-Wesley Publishing Company, Second Edition, 1994.

Borenstein, N. and Freed, N. (Borenstein and Freed, 1993). “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies,” Request for Comments 1521, September 1993. Available at: http://sunsite.auc.dk/RFC/rfc/rfc1521.html

Borgida, A., Brachman, R.J., McGuiness, D.L. and Resnick, L.A. (Borgida, Brachman, McGuiness and Resnick, 1989), “CLASSIC: a structural data model for objects,” Proceedings of 1989 SIGMOD Conference on the Management of Data. pp.58-67. New York, ACM Press.

Cox, B., (Cox, 1986), “Object Oriented Programming: An Evolutionary Approach,” Addison-Wesley, 1986.

Crocker, D.H. (Crocker, 1982). “ARPA Internet Text Messages,” Request for Comments 822, August 1982. Available at: http://sunsite.auc.dk/RFC/rfc/rfc822.html

Dean, D., Felten, E.W., and Wallach, D.S., (Dean, Felten and Wallach, 1996), “Java Security: From HotJava to Netscape and Beyond," IEEE Symposium on Security and Privacy, Oakland, California, USA, May 6-8, 1996. Available at: http://www.cs.princeton.edu/sip/pub/secure96.html

Ellis, C.A., Gibbs, S.J., and Reln, G.L., (Ellis, Gibbs and Reln, 1991). "Groupware: Some Issues and Experiences," Communications of the ACM, 34(1), 1991. pp. 39-58.

Gaines, B.R., (Gaines, 1991). "Modeling and Forecasting the Information Sciences," Information Sciences, 3(22), 1991. pp. 57-58. Available at: http://ksi.cpsc.ucalgary.ca/articles/BRETAM/InfSci/

Gaines, B.R. (Gaines, 1993). "Situated Action Solution of a Resource Allocation Problem as a Classification Task Represented in a Visual Language," International Journal of Human-Computer Studies, 1993. pp. 243-271.

Gaines, B.R., Kremer, R. and Flores-Méndez, R.A., (Gaines, Kremer and Flores-Méndez, 1996), “Concept Mapping Development at the Knowledge Science Institute,” Poster track at the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, Canada, November 9-14, 1996.

Gaines, B.R. and Shaw, M.L.G. (Gaines and Shaw, 1995a). “Collaboration through Concept Maps,” Proceedings of CSCL95: Computer Supported Cooperative Learning. Bloomington, October, 1995. Available at: http://ksi.cpsc.ucalgary.ca/articles/CSCL95CM/

Gaines, B.R. and Shaw, M.L.G. (Gaines and Shaw, 1995b). "Concept Maps as Hypermedia Components," International Journal on Human-Computer Studies, 1995. pp. 323-361. Available at: http://ksi.cpsc.ucalgary.ca/articles/ConceptMaps/

Gaines, B.R., Shaw, M.L.G., Chen, L.L.J. (Gaines, Shaw and Chen, 1996), “Modeling the Human Factors of Scholarly Communities Supported Through the Internet and World Wide Web,” Journal of the American Society for Information Science, Winter-Spring 1996.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (Gamma, Helm, Johnson and Vlissides, 1995). “Design Patterns, Elements of Reusable Object-Oriented Software,”Addison-Wesley, 1995.

Garner, B.J., Tsui, E., Lui, D., Lukose, D., and Koh, J. (Garner, Tsui, Lui, Lukose and Koh, 1992), “Progress on an Extendible Graph Processor for Knowledge Acquisition, Planning and Reasoning,” Current Directions in Conceptual Structure Research, Tim Nagle, Jan Nagle, Laurie Gerholz and Peter Eklund (Eds.). Ellis Horwood, 1992.

Goldberg, A., and Robson, D., (Goldberg and Robson, 1989), “Smalltalk-80: The Language,” Addison-Wesley, 1989.

Goodman, D. (Goodman, 1993). “The Complete AppleScript Handbook,” New York, Random House, 1993.

Gosling, J., Rosenthal, D.S.H., and Arden, M., (Gosling, Rosenthal and Arden, 1989), “The NeWS Book," Springer-Verlag, 1989.

Gutwin, C., Greenberg, S. and Roseman, M. (Gutwin, Greenberg and Roseman, 1996), “Workspace Awareness in Real-Time Distributed Groupware: Framework, Widgets, and

Evaluation," People and Computers XI, Eds. A. Sasse, R.J. Cunningham, and R. Winder. Springer-Verlag, in Press. From the proceedings of HCI'96 (London, August 20-23, 1996).

Harrison, C., Chess, D., Kershenbaum (Harrison, Chess and Kershenbaum, 1995), A., "Mobile Agents: Are they a good idea?", IBM Watson Research Center, March, 1995. Available at: http://www.research.ibm.com/massive/mobag.ps

Kernighan, R., Ritchie, D., (Kernighan and Ritchie, 1978), "The C Programming Language," Prentice-Hall, New Jersey, 1978.

Kirtland, M., (Kirtland, 1996), "Safe Web Surfing with the Internet Component Download Service," Microsoft Systems Journal, pg. 65-73, July 1996.

Kremer, R., (Kremer, 1993). "A Concept Map Based Approach to the Shared Workspace," Master Thesis, The University of Calgary, June, 1993.

Kremer, R., (Kremer, 1996). "Towards a Multi-User, Programmable Web Concept Mapping "Shell" to Handle Multiple Formalisms," Proceedings at the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, Canada, November 9-14, 1996. Available at: http://www.cpsc.ucalgary.ca/~kremer/KAW96paper/kremer.html

Kremer, R. and Flores-Méndez, R. A. (Kremer and Flores-Méndez, 1996). "BNFs for KSI Map Storage Formats," Draft, 1996. Available at: http://ksi.cpsc.ucalgary.ca/local/software/storageBNF.html

Lapsley, A.Z., (Lapsley, 1995). "Development of a Mediator System on the World-Wide Web to Model the Concurrent Manufacturing Life Cycle," Master Thesis, The University of Calgary, July, 1995.

Lewis, C., and Rieman, J., (Lewis and Rieman, 1994), "Task-Centered User Interface Design: A Practical Introduction," University of Colorado, 1994. Available at: ftp://ftp.cs.colorado.edu/pub/cs/distribs/clewis/HCI-Design-Book/

Lukose, D., (Lukose, 1996), "MODEL-ECS: Executable Conceptual Modeling Language," Proceedings at the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, Canada, November 9-14, 1996.

MacGuire, S. (MacGuire, 1993), "Writing Solid Code," Microsoft Press, 1993.

McNeese, M.D., Zaff, B.S., Peio, K.J., Snyder, D.E., Duncan, J.C. and McFarren, M.R. (McNeese, Zaff, Peio, Snyder, Duncan and McFarren, 1990). "An Advanced Knowledge and Design Acquisition Methodology for the Pilot's Associate," Harry G Armstrong Aerospace Medical Research Laboratory, Wright-Patterson Air Force Base, Ohio. 1990.

Microsoft Corp. (Microsoft, 1996a), "Visual Basic Scripting Edition," Microsoft, Corp., 1996. Available at: http://www.microsoft.com/vbscript/

Microsoft Corp. (Microsoft, 1996b), "Internet Explorer" Microsoft, Corp., 1996. Available at: http://www.microsoft.com/

Microsoft (Microsoft, 1996c), “Internet Component Download," Draft Paper, ActiveX SDK, Microsoft, Corp., May 1996. Available at: http://www.microsoft.com/intdev/sdk/

Microsoft (Microsoft, 1996d), “Windows Trust Verification Services," Draft Paper, Microsoft, Corp., February 1996. Available at: http://www.microsoft.com/workshop/prog/default.asp

Myers, A.C., Bank, J.A., and Liskov, B. (Myers, Bank and Liskov, 1997). “Parameterized Types for Java,” Proceedings of ACM Symposium on Principles of Programming Languages, January 1997, pp. 132-145. Available at: ftp://ftp.pmg.lcs.mit.edu/pub/thor/popl97/popl97.html

National Center for Supercomputing Applications, HTTPd Development Team (NCSA, 1996), “The Common Gateway Interface,” National Center for Supecomputing Applications, 1996. Available at: http://hoohoo.ncsa.uiuc.edu/cgi/

National Center for Supercomputing Applications, (NCSA, 1997), “Habanero Project," National Center for Supercomputing Applications, 1997. Available at: http://www.ncsa.uiuc.edu/SDG/Software/Habanero/

Naughton, P., (Naughton, 1996), “The Java Handbook,” Osborne McGraw-Hill, 1996.

NCompass (NCompass, 1996), “ActiveX Plug-in," NCompass Labs, Inc., 1996. Available at: http://www.ncompasslabs.com/activex/

Netscape Communications Corp. (Netscape, 1996), “JavaScript Authoring Guide,” Netscape Communications Corp., 1996. Available at: http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/

Netscape Communications Corp. (Netscape, 1996b), “Netscape Navigator," Netscape Communications Corp., 1996. Available at: http://home.netscape.com/

Netscape Communications Corp. (Netscape, 1996c), “LiveConnect Communication," Netscape Communications Corp., 1996. Available at: http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/

Netscape Communications Corp. (Netscape, 1996d), “The Plug-in Developer's Guide,” Netscape Communications Corp., 1996. Available at: http://home.netscape.com/eng/mozilla/3.0/handbook/plugins/pguide.htm

Norrie, D.H. and Gaines, B.R. (Norrie and Gaines, 1995). “The Learning Web: A System View and an Agent-Oriented Model,” International Journal of Educational Telecommunications 1(1) 23-41, 1995. Available at: http://ksi.cpsc.ucalgary.ca/articles/LearnWeb/EM95J/

Novak, J.D. and Gowin, D.B. (Novak and Gowin, 1984). “Learning How To Learn,” New York: Cambridge University Press. 1984.

ObjectSpace, Inc., (ObjectSpace, 1996), “The Java Generic Library," ObjectSpace, Inc., 1996. Available at: http://www.objectspace.com/jgl/

Ousterhout, J. (Ousterhout, 1994), “Tcl and Tk Toolkit," Addison-Wesley, 1994.

Quillian, M.R. (Quillian, 1968). “Semantic memory,” Semantic Information Processing, M. Minsky, Editor. MIT Press: Cambridge, Massachusetts. p. 216-270, 1968.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. (Rumbaugh, Blaha, Premerlani, Eddy and Lorensen, 1991). “Object-Oriented Modeling and Design,” Prentice Hall, 1991.

Shaw, M.L.G., and Gaines, B.R. (Shaw and Gaines, 1995). “Comparing Constructions through the Web,” Proceedings of CSCL95: Computer Supported Cooperative Learning, Bloomington, October, 1995. Available at: http://ksi.cpsc.ucalgary.ca/articles/CSCL95WG/

Shaw, M.L.G., and Gaines, B.R. (Shaw and Gaines, 1996). “Experience with the Learning Web,” Proceedings of ED-MEDIA'96: World Conference on Educational Multimedia and Hypermedia, 1996. Available at: http://ksi.cpsc.ucalgary.ca/articles/LearnWeb/EM96Exp/

Shoffner, M., and Hughes, M. (Shoffner and Hughes, 1996), “Java and Web-Executable Object Security," Dr. Dobb’s Journal, Pg. 38-49, November, 1996.

Smart Ideas Technologies Inc., (Smart Ideas, 1996). “Smart Ideas,” Smart Ideas Technologies, Inc., 1996. Available at: http://www.smarttech.com/

Sowa, J.F. (Sowa, 1984). “Conceptual Structures: Information Processing in Mind and Machine,” Addison-Wesley, Reading, Massachusetts, 1984.

Stepanov, A, and Lee, M., (Stepanov and Lee, 1996), "The Standard Template Library," Hewlett-Packard Laboratories, February, 1995. Available at: http://www.cs.rpi.edu/~musser/stl.html

Stroustrup, B., (Stroustrup, 1991), “The C++ Programming Language,” Second Edition, Addison-Wesley, 1991.

Sun Microsystems, (Sun, 1995a), “The Java Virtual Machine Specification," Release 1.0 Beta, Draft Paper, Sun Microsystems, Inc., 1995. Available at: http://java.sun.com/docs/

Sun Microsystems, (Sun, 1995b), “HotJava: The Security Story," Sun Microsystems, Inc., 1995. Available at: http://java.sun.com/docs/

Sun Microsystems, Inc. (Sun, 1996a), “The HotJava Browser," 1996. Available at: http://java.sun.com/HotJava/

Sun Microsystems, (Sun, 1996b), “Frequently Asked Questions - Applet Security," Sun Microsystems, Inc., 1996. Available at: http://java.sun.com/sfaq/index.html

Sun Microsystems, Inc., (Sun, 1996c), “Java Development Toolkit," Sun Microsystems, Inc., 1996. Available at: http://java.sun.com/

van Rossum, G. (van Rossum, 1996a), “Python Reference Manual," Corporation for National Research Initiatives, 1996. Available at: http://www.python.org/

van Rossum, G. (van Rossum, 1996b), “Grail – The Browser for the rest of Us," Corporation for National Research Initiatives, 1996. Available at: http://grail.cnri.reston.va.us/grail/

White, J. E. (White, 1995), “Mobile Agents,” October, 1995. Available at: http://www.genmagic.com/agents/Whitepaper/whitepaper.html

Yellin, F., (Yellin, 1995), “Low Level Security in Java," World Wide Web Journal: Fourth International World Wide Web Conference Proceedings, pg. 369-379, November 1995. Available at: http://www.w3.org/pub/WWW/Journal/1/f.197/paper/197.html

Zimmermann, P., (Zimmermann, 1995), “The Official PGP User's Guide," MIT Press, 1995.