Neuroscape : a web-based informatics infrastructure for classroom & research use
by Edward Duncan Smith Kennedy

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
This thesis describes Neuroscape, an Informatics client system that provides classroom access to a variety of remote data sources for inquiry and discovery based learning. There are educational laboratory packages that provide pre-configured experiments using conditioned data sets. Using these lessons can be instructive, but they are also quite restrictive for the student, due to the determinism of the outcome. There are potentially available alternative sources of data. Most research laboratories maintain large quantities of data collected from their experiments, which could be made available to other researchers and students. However, accessing this data in a consistent manner is quite challenging because the data is not categorized and stored in a consistent manner. Neuroscape provides a simple and innovative solution to this problem.

The Neuroscape system, which has a two-tier distributed object architecture, provides a Java Applet user interface accessible via the World Wide Web. The user is provided with tools for making queries of whatever data sources are available to the system, and tools for manipulating the results of those queries. A Neuroscape system can be extended to suit most any field of study that benefits from accessing remote sources of information via inquiry. Neuroscape provides a modular architecture that is predictably extensible. The modules represent the primitive capabilities needed for inquiry of a body of knowledge. The depth of their capabilities are bounded by the capabilities of their implementor. Since the student is free to explore using whatever capabilities are provided by the modules connected to the system, and the modules can be used with each other, the possibility for real discovery is in the student's hands.

Neuroscape:
A Web-Based Informatics Infrastructure for Classroom & Research Use

by

Edward Duncan Smith Kennedy

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY - BOZEMAN
Bozeman, Montana

April 1999

APPROVAL

of a thesis submitted by

Edward Duncan Smith Kennedy

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Gary Harkin — (Signature) — Date: 4/13/99

Approved for the Department of Computer Science

J. Denbigh Starkey — (Signature) — Date: April 13th 1999

Approved for the College of Graduate Studies

Bruce McLeod — (Signature) — Date: 4-14-99

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University - Bozeman, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature E.D. Smith Kennedy

Date 4/13/99

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

This thesis describes Neuroscape, an Informatics client system that provides classroom access to a variety of remote data sources for inquiry and discovery based learning. There are educational laboratory packages that provide pre-configured experiments using conditioned data sets. Using these lessons can be instructive, but they are also quite restrictive for the student, due to the determinism of the outcome. There are potentially available alternative sources of data. Most research laboratories maintain large quantities of data collected from their experiments, which could be made available to other researchers and students. However, accessing this data in a consistent manner is quite challenging because the data is not categorized and stored in a consistent manner. Neuroscape provides a simple and innovative solution to this problem.

The Neuroscape system, which has a two-tier distributed object architecture, provides a Java Applet user interface accessible via the World Wide Web. The user is provided with tools for making queries of whatever data sources are available to the system, and tools for manipulating the results of those queries. A Neuroscape system can be extended to suit most any field of study that benefits from accessing remote sources of information via inquiry. Neuroscape provides a modular architecture that is predictably extensible. The modules represent the primitive capabilities needed for inquiry of a body of knowledge. The depth of their capabilities are bounded by the capabilities of their implementor. Since the student is free to explore using whatever capabilities are provided by the modules connected to the system, and the modules can be used with each other, the possibility for real discovery is in the student's hands.

STUDENTS, COMPUTERS, AND LABORATORY SOFTWARE

The basic process used in scientific inquiry is familiar to anyone who has spent time in a science class. This process follows six steps. First, an unanswered problem is identified, and a hypothesis providing the answer to the problem is proposed. Then, an experiment is designed to test the validity of the hypothesis. Next, the experiment is performed, and data is recorded during the execution of the experiment. Analysis is conducted on that data according to the design of the experiment. Finally, some conclusion is arrived at as a result of the execution of the experiment and an analysis of the data.

In many fields of scientific study, there is an overwhelming quantity of data that has been collected from experiments. This data resides in databases and file systems in many different locations, and in many different formats. There is an enormous potential for exploration in these repositories of data. However, accessing this data in a consistent manner is either challenging or not currently possible because the data is not categorized and stored in a consistent manner. Many factors contribute to the problem of providing access to data. These include: the heterogeneous nature of scientific data collected even in a single laboratory; the inherent complexity of many data types; storage for a particular field is decentralized; and the lack of a universal taxonomy for categorizing all the data that is collected by scientists in even one field. All of these factors contribute to the prob-

lem of providing reasonable, uniform access to this tremendous mass of data. Systems that can solve this general problem have been given a name: Informatics systems.

Informatics Systems

There are three methods of implementing an Informatics system. These different methods are characterized by where the burden lies for providing the unifying organization and interface. The first method involves developing a universal taxonomy for categorizing all the data that is collected by scientists in one field, and fitting all of the data into some form of "container" that organizes all the data in terms of that taxonomy. The container can either be a centralized database for the data, or middleware that provides the appearance of a central database. This puts the burden on the data sources themselves to present a uniform interface.

The second method involves letting the data sources remain different by shifting the burden to present a uniform interface to the client. This client knows how to connect to different data sources, and present the appearance of a unified system to the user. Data could be manipulated using tools that could accommodate similar but different data types. An Informatics system designed in this method is clearly a less scalable solution, since the data sources remain heterogenous and the number of data types continues to proliferate.

The third method is a hybrid of the first two methods. It involves using the second method to provide the initial uniformity until the complexity of the first method can be solved. The first method is clearly the ideal method

that an Informatics systems should be implemented. If the method of accessing any data source were the same for all data sources, then the tools for accessing the data in those data sources could be simpler, which would provide a much more scalable solution. Unfortunately, using the first method is very challenging to implement if the data is complex. Successful Informatics systems that use the first method usually manage data that is quite homogenous and simple in nature. The Neuroscape system takes the second method to solving the problem that Informatics systems are intended to solve, and is easily adaptable to be usable in the third method as well.

Informatics and The Classroom

Students in science classes could benefit greatly from access to Informatics systems. In most classroom or laboratory settings, students interact with computers in static and restrictive ways. Students use computers to examine data sources that are used as references, which may also be used to perform "cookbook" exercises. These references contain reasonably large quantities of information that are potentially very helpful to the user, but the way that the data is presented to the user is typically very restrictive, for two reasons. First, the organizational framework of the data is usually inflexible. Second, the information in the reference is "packaged" so that it can be viewed or manipulated in very limited ways, which diminishes its overall utility to the end user.

The organizational framework most commonly used to provide a method of subdividing the information into smaller divisions is a hierarchical index, to attempt to help the user to find the information for which he or she is

searching. Clearly, an index of any kind limits the way that a user can discover information that is useful to her. When presented with an very large quantity of information (as is typically present in one of these references), a more sophisticated method is necessary. Search capabilities are a good place to start, but if the search criteria are very limited, only a small gain in functionality has been achieved. Moreover, by restricting the search functionality to the data that is present in that one reference source, the user's ability to further understand connections and possibly establish or discover new connections is virtually eliminated.

The other problem involves the way that a body of data may be "conditioned" for use by the end user. It is not uncommon for the information in the reference to be packaged so that it can be viewed in special but very limited ways by the end user. This also limits the user's capability to discover new things from that body of data. Frequently, in a valid attempt to make the information more engaging to the user, the information provide a higher level of interactivity, but at the expense of usability in other contexts. For example, the user may watch a QuickTime movie that shows a "how-it-works" presentation of some process, but the original data that was used to create the movie in the first place may not have been included in the reference source. Thus, the user can only get information about the content of the movie and not about the original data source.

Both of these problems set a limit on the level of true investigative capability and possibility for real discovery. These limitations result in a finite number of paths that the student can follow in her exploration of the available data in the reference source or exercise. Frequently, activities or lessons based on this kind of reference source are themselves restricted in

the types of tasks the student can be asked to perform. The student will be asked to perform some activity that has a specific result that she is trying to achieve, much in the same way a person uses a cookbook to ultimately achieve the goal of making something more complicated to eat. The student is frequently graded on how well they followed the recipe. Partial credit is awarded to those that seem to be on the right track, but identifying whether or not a student is on the right track can be difficult.

Neuroscape: Informatics For The Classroom

Neuroscape is a system that attempts to solve these shortcomings, by bringing a discovery based Informatics client to computer classrooms. By providing an infrastructure where a system of inter-operable modules may be used together to query data sources, organize the results of these queries, and analyze those results in meaningful ways, the student user has a greater opportunity to discover and understand scientific concepts. Software technology empowers students to explore a potentially vast world of data, using an expandable array of modules that provide functionality that can potentially be used in ways that may not have been intended by their creators. Neuroscape is a system that acknowledges and accepts the existence of heterogeneous data types from different and remote sources. Moreover, by providing a way of easily extending the functionality of the system, Neuroscape can accommodate different data sources and the information they contain, and handle different data types in similar or common ways. This system, then, creates a set of bridges between different data types, data sources, and analysis tools.

The Neuroscape system provides interactive laboratory classroom software that uses discovery and exploration as its way of helping students to understand a scientific discipline. This is achieved by using dynamic data sources, or sources that can grow and extend over time. It is important to emphasize that these data sources are not limited to the filtered or prepared data commonly used in most educational systems. On the contrary, real research data stored in databases or file servers, whether open or requiring authentication, are still potentially available to users of Neuroscape. Any arbitrary data source is available to Neuroscape if it is connected to the Internet. All that is required to access a new data source is the creation of a Module to access that data source, and it has become a part of the system. Moreover, data can originate from diverse sources seamlessly.

Thus, a user of Neuroscape has the potential to not only have a greatly expanded field of information to interact with, but also has the potential to make connections across data sources and use data from different sources in unified ways. This capability, which is known as data mining, is very powerful. Neuroscape allows the user with access to a Neuroscape Applet to be mining data by interacting with a real Informatics system whose limitations are defined by what Modules are plugged into it.

Data Sources are seen by the user as being "smart", in the sense that they are aware of what is in them. Users acquire data from data sources by formulating questions, or Queries, which are posed to the Data Sources. Since the actual "intelligence" that the Data Source seems to possess is actually present within the Neuroscape Module that provides an interface to that Data Source, the burden for intelligence is actually on the Neuroscape

system and the set of Modules attached to it.

The Result of a Query is returned to the user for examination and manipulation. A Result is a collection of references to discrete blocks of data, each of which is called a DataFile. Examination and manipulation can be performed in two ways. First, the attributes of one or more DataFiles can be examined directly. Second, the Results can be examined in a Viewer, which provides a context that is specifically relevant to that particular DataFile's type, or in some kind of suitable intermediate format. Specific viewers can allow more detailed examination of particular data types in ways that more general Viewers might not provide. Alternatively, it might be desirable to view DataFiles of different but similar types in some type of environment where they could both be represented in some common way, for the purpose of comparison.

An important feature of the system is the ability of the user to revise a Query that has been previously made. If the user is not happy with a Query's result, or wishes to further explore by making further Queries of a data source, previous queries may be revised, or used as starting points for the new inquiry. In the process, the original Query may be kept or discarded, according to the user's decision. This ability is significant because it allows a line of inquiry to be pursued with a level of continuity and organized discovery that is not commonly found in other systems. Further, if all of the Queries are kept in this process, any of them represent a possible branching point in a line of inquiry. This ability to pursue further lines of inquiry, coupled with the ability to perform branching inquiries, is one of the most empowering capabilities Neuroscape offers the user.

Another important feature is the ability to treat Results as new Data

Sources, for the purpose of inquiry and the generation of new data. The capability to derive new data from existing data means that the number of data sources can grow as a result of using the system. If the user can create her own new data simply by using the system, then not only is the user empowered, but the potential for growth of information is expanded simply by having users use the system. This will become even more important with the addition of persistent storage facilities for the user's work, because then the derived data will remain in existence and be usable by other users as well.

Although this system was originally designed for use in a Neurophysiology laboratory, Neuroscape's architecture is flexible enough that it can be adapted to almost any field of scientific study where students wish or are asked to make queries of bodies of knowledge. Other areas in the Biological sciences, Chemistry and Psychology are several that come to mind, but Neuroscape's usability is not limited to these fields. Neuroscape could also be adapted for use in the Social Sciences, such as Epidemiology, Anthropology, Economics, or Business. The extensibility of the system allows instructors to develop new Modules that suit their particular field of study without having to redevelop the infrastructure that Neuroscape provides. The extensibility and flexibility of the Neuroscape architecture also allow the system to be used for actual research work in addition to classroom use. The set of Modules attached to the system is the key to its usability in other fields.

The Neuroscape system is also not limited to running on one particular computer architecture. As will be discussed later, all components of Neuroscape are implemented in 100% Pure Java, to ensure complete plat-

form independence. With this system, both the teacher and the students win. The teacher is free to run the system on whatever architecture she has available to her. More importantly, the students can access Neuroscape clients from their teachers, but potentially other teachers as well.

FLEXIBLE ARCHITECTURE ALLOWS FLEXIBLE USE

The Neuroscape system's ability to access heterogeneous sources of data and provide adaptable and flexible tools to interact with this data are made possible by both the core architecture and its extensibility. Neuroscape's architecture was conceived using object-oriented analysis and design (OOA/D) techniques. These techniques were extremely helpful in identifying the responsibilities of the components of the system and the way those components should interact.

It was very important to identify sufficiently simple primitives that represent fundamental abilities or actions. These primitives, represented in Neuroscape as Modules, allow a particular task to be characterized by a flow of interaction between a number of these Modules. Thus, Neuroscape's Modules provide building blocks for developing more complex operations or tasks. Some seemingly simple tasks are actually composed of a fairly complex interaction between several or a number of different types of Modules. The task may also involve several iterations of the process described above, where Queries are made, Results are gathered, and further Queries can be made of those Results. The addition of a significant set of new functionality may involve the implementation of a number of Modules. Neuroscape's Modular architecture allows the functionality of the system to be extended or changed by changing the Modules that are added.

If these Modules are to be manipulated by the Neuroscape system in

predictable ways, they must be capable of giving the Neuroscape system meaningful information about themselves. They must also provide a common way of performing their function, so that Neuroscape does not have to analyze the Module to attempt to discover how it is to perform its function. In object-oriented programming (OOP) terminology, a class' ability to perform tasks or provide information about itself is provided by its list of Methods. By implementing a hierarchy of abstract parent classes, the need for a consistent communications interface may be enforced. The design of a given Module type's interface is also important because inter-Module communication is also important.

To permit open communications between Modules and the core Neuroscape system to occur, Modules should be implemented in such a way that the dependency between specific Module implementations is minimized. It is especially important for allowing just about any Module to communicate with another Module. Dependency results from the situation where one Module provides its functionality via an individual and unique interface, and another Module that uses that nonstandard interface. Clearly, this is not a situation that should be encouraged. By minimizing dependency of this type, new capabilities coming from interactions between different Modules that were not previously intended remain possible. A greater level of dependency will result in a less flexible system. This principle seems like common sense, but it is important to recognize what constitutes dependency in a modular system that is designed to maximize flexibility. Clearly, there is no way to eliminate all dependency. But by trying to minimize it, the potential for flexibility is increased.

Java and Neuroscape

Many of the fundamental capabilities of this system are possible because the core of the system is implemented in the Java programming language, and Java's capabilities were crucial to this project. To quote a very helpful book, Java is a "simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multi threaded, and dynamic language".[1] In addition to these characteristics, Java is a popular language, which makes it likely that the source has a reasonably long usable life. Many of the reasons why these characteristics are so helpful are interrelated. All of these characteristics made Java the right language to use for implementing Neuroscape.

The Neuroscape system was designed using object-oriented design and analysis techniques. The object-oriented method of designing and developing software has been said to be the single most important method in improving the efficiency of software development found so far. Developing the design of a system in this way allows the software developer to characterize the solution to a problem as a number of responsibilities that can be given to objects, or software entities of various types, known as classes. The quick and intuitive way that a developer can subdivide the responsibilities of a system into groups that are related, and then design classes of objects that can take on these responsibilities, is a great advantage of object-oriented design. Another great advantage is that the object-oriented paradigm is based on the notion of objects communicating with one another via messages. The significance of this characteristic will become clear shortly.

Once a design has been created using OOA/D techniques, it makes sense to use an object-oriented language to implement the design. There are several popular object oriented programming languages in common use today, including Java, Ada95, and C++, which is perhaps the most commonly used of all of them. Ada95 and C++ are more complex, offering a broader array of object-oriented capabilities or greater expressive power than Java in some respects. For instance, C++ allows multiple inheritance, which would have been very helpful in certain instances in the implementation of the Neuroscape design. However, it was not strictly on a languages' object-oriented expressive capabilities that a choice was made.

Java is a language that supports distributed computing behavior. The Neuroscape system was intended from the start to be used to implement distributed software systems, where blocks of the functionality could be located on different computer systems, and new functionality could be loaded across a network. Networking capability, in both low-level (socket) and high-level (remote procedure call) forms, is a fundamental and essential part of the Java language. It has also been of great interest to the author. There are features of Java that make it unlike any other popular language for programming networked applications. Unlike other languages (most notably C and C++), Java's data types are defined according to a precise specification, and are fixed in length, no matter what platform they are used on. This is an important feature, especially when using remote procedure call (RPC) systems, which almost always require the use of their own rigidly defined data types. It is a trivial issue to map the RPC's types to the native data types of Java. In Neuroscape we use Java RMI for RPC, but other systems could also be used.

Portability and platform-independence were two primary design criteria for the Neuroscape system. Java's platform-independent characteristics are well publicized, and so are its problems. There were some problems regarding compatibility between different versions of Java, but these were very minor. Another primary design characteristic was to make the user interface Web accessible. At the moment, Java is the only practical way of providing this capability. Java is a compiled language, but the executable code that is generated is for an imaginary (virtual) machine that is typically emulated in software. A Java Virtual Machine (JVM) is the common name for that emulator. This allows Java executable code, which is known as Byte Code to be executed on any machine that had a JVM installed on it. The Byte Code is executed using an emulator, so Java is considered an interpreted language. But it is really a hybrid language, since the Java source itself is not interpreted by the Java Virtual Machine. A JVM is built into most modern Web browser software. Portability came at the expense of performance, however, since the JVM was always emulated in software. In the past two years, compilers have been built into JVMs that translate the Java Byte Code into instructions native to the platform running the JVM. These compilers, called Just-In-Time compilers, have had a great effect on the speed of execution of Java code. Thus, the performance concerns of using Java have become less of an issue. In particular, Java programs that have been executing for a long period of time on a computer running a JVM that has a Just-In-Time compiler will have probably translated all of their executable code into native instructions. So Java can now be used for back end computing with little impact on execution efficiency, and provide better performance on the front end as well.

Neuroscape and Java's Dynamic Capabilities

One of the most important features of Java for implementing a system such as Neuroscape is that it is a dynamic language. This means that the resolution of dependencies are performed closer to the time that the code is executed. In Java, as in other object-oriented languages, there is a distinction made between when a class is loaded, and when an instance of that class is created. The loading of a class is like loading the blueprints for making something, and the creation of a new instance is like building that thing from the blueprints. An important part of the way Java loads its classes is that, if it finds something in the blueprints that it does not recognize, it is not a problem until a request for the construction of that object is requested. This is usually called late binding. By comparison, a more static language requires that dependencies must be resolved closer to the time that the language is compiled, or translated from source code into executable form. If a programmer defines a class A, and uses within that class another class B, the Java Virtual Machine will not look for the implementation of class B until an instance of class B needs to be created, either on its own, or as a part of instantiating class A. This makes Java very flexible, because it means that, even if the implementation is not available at the time the class is loaded, it might be available when the class is instantiated. A consequence of this late resolution of dependencies is that there must be more services provided to a program at execution time, to allow this resolution to occur. Since Java Byte Code is executed on a JVM, the JVM can provide these services to all Java programs. A fundamental aspect of the design of Neuroscape is that the functionality can be loaded and extended

while the system is running.

The Neuroscape system takes advantage of Java's dynamic features, such as execution time loading of new classes, resolution of dependencies when a class is instantiated rather than when it is loaded, and Java's Reflection API, which allows a program to examine any loaded class specification. These are very powerful additions to an object-oriented language, because it allows for a great deal of adaptability. The dynamic loading and late resolution of dependencies pervades not only the way that Java was used to implement Neuroscape's design, but also how the components of the Neuroscape system perform their own duties in providing services for the user.

An interesting result of both Java's portability and dynamic qualities is that the system that the software is developed on need not have support for run-time system dependencies that do exist, either between versions of Java or for other API packages. The entire Neuroscape system was implemented using Metrowerks' CodeWarrior Academic Pro Release 4, running on an Apple Macintosh PowerBook 3400c/240. This includes development of test Modules to evaluate the usability of new Java software technology, such as Java3D. Although no run time support for Java3D exists at the time of this writing, executable files could be created on the Macintosh, and they could be executed on systems that do provide run-time support for Java3D, such as Sun Microsystems' own Solaris and Microsoft Windows 95/98/NT. The portability of Java executable files is real and the technology works.

THE COMPONENTS OF THE NEUROSCAPE SYSTEM

A formal description of the Neuroscape system requires a characterization of the system's components. Since the design and implementation of Neuroscape was done in an object-oriented manner, it should be noted that much of the terminology used in the characterization will be done using terms that are commonly used to describe object-oriented systems. If you are not familiar with these terms, an excellent starting point is "An Introduction to Object-Oriented Programming" by Timothy Budd.[2] Some of the terms are specific to the Java language's implementation of object-oriented constructs. A good basic reference to the Java programming language is "Java in Plain English" by Brian Overland.[3]

There are two families of components: Modules and Managers. There are also a limited number of other components that provide similar functionality but are not related to the first two categories in terms of the class inheritance hierarchy. The core of the Neuroscape framework consists of the Neuroscape Applet and six Managers. All of the components of the core framework oversee specific service domains, and provide basic services to each other and the user. If Neuroscape was implemented as a monolithic application on a single computer system, all components would communicate locally. However, the distributed architecture of this system provides its special capabilities.

This description of the Neuroscape system requires the definition of

some terms. What has classically been referred to as the "client" side will be called Applet-side, and the "server" side will be called ProxyServer-side. This is because the overall system is not really a client and a server, but an application that is tightly bound yet distributed in its execution location. Although in this system's initial implementation many of the Managers are run on a single system, they could all just as easily be run on separate systems. Further, it is possible to have the roles of "client" and "server" reverse in distributed object environments.

Some classes will seem trivial from an implementation point of view, since they provide very little functionality themselves. Nevertheless, what was developed was a framework for how these are to interact. That a given class provides little functionality on its own does not detract from its utility within the system. Its place in the inheritance hierarchy is a greater indication of its significance in the system. Also, some components in this description will be mentioned before they have themselves been defined. This is inevitable in a system where there is a complex relationship that exists between a number of components. However, by noting both the description of a component in its own definition and the component's mention in other components' definitions, a complete picture of the system will be presented.

Modules: Controlled, Predictable Extensibility

Neuroscape provides an architecture that can be extended in a controlled and predictable way. Functionality is extended by creating new Modules. There are 5 types of Modules: Query, DataFile, Viewer, Translator, and

Computation. Each module type represents a primitive set of functionality that can be used in conjunction with other modules to achieve tasks. All direct subclasses of the Module class are themselves abstract classes, and their abstract interface is what allows new functionality to be added to the system in a predictable and controlled manner. Modules are first managed in the Neuroscape system by the ModuleManager.

The Module class is the parent class of all Modules. This class, which is abstract, provides little functionality itself. But it is the parent of all Module types, and is significant for that reason alone. It is also available for extension in the future. It provides a way of getting a formal Name for the instance of this type, and maintains a list of errors of the problems that the object has encountered. This list can be retrieved for examination, and it can also be purged to save space.

Query Module

A Query Module is the Module type that provides a smart interface into a Data Source. Its capability includes three interrelated tasks. First, it provides the functionality needed for the user to formulate a query ("ask a question") of a number of particular Data Sources, and for interpreting the result of that query. Making the query will result in the creation of a list of references to particular DataFiles that have been registered with the DataManager. A DataFile's content is either present as a result of the query, or requires additional requests from the Data Source, and will possibly be loaded later. Second, the Query Module also must provide the functionality for trying to retrieve the content of a DataFile that is in any result list that it can generate. Finally, a query maintains the QueryPanel used to submit

the query, and the list of DataFileReferences that are the result of submitting that query.

Query objects are one of only two Module types that are regularly passed between the Neuroscape Applet and any of the remote Managers. Not surprisingly, Query objects are absolutely fundamental to the successful operation of Neuroscape. The ability to build a single query or a series of queries is based entirely on the capabilities these Modules implement. Their ability to function successfully affects almost every other facility in Neuroscape.

One consequence of this mobility is that the functionality provided by a Query Module, aside from its utility interface, is divided into two sets. The first set is executed on the Applet-side, and depends on the presence of a user interacting with it. The other set gets executed remotely from the Applet. There is an area of intersection between these two sets. The methods in this intersecting area could be executed either local to the Applet or remotely. If a method is remotely executed, it needs to be invoked by some other software entity that can accept the Query object and invoke its method. This other software entity is the QueryManager. The QueryManager partially acts as a proxy execution environment for the Applet. It invokes the Query's methods for the Applet. This flexibility could prove to be useful in the future, but for now is not used, and all of the methods in the remote set are executed remotely.

Queries are formulated by gathering information from the user. This information is gathered via a graphical user interface "form", known by the system as a QueryPanel, that must be filled out by the user. The GUI components on this panel and the way they are interpreted to formulate the

query for the Data Source, is also provided by the individual implementation of a Query Module. So the interface provided by this class has methods for formulating the query and submitting it. Specifically, there are methods for getting that Query's QueryPanel, asking the Query if its QueryPanel is ready for submission, resetting the state of the QueryPanel to a default state, and for submitting the query. There are also methods for retrieving the content of a particular DataFile, and for getting the Query Module's ResultList.

DataFile Module

A DataFile Module encapsulates the functionality needed to store some block of data, plus the attributes of that data or the entity that the data represents. Also, functionality is provided to evaluate the correctness of the data according to some standard syntax specification, to evaluate the data's usability within the system. DataFile Modules will contain data that has been retrieved from some Data Source. Since Data Sources can be remote databases, remote file systems, or a collection of other DataFiles, they must be quite simple and generic in their use. Attributes are stored in a data structure that is openly accessible by all Modules, but with contents that may have significance only to certain other Modules.

These Modules themselves will never be transported to the Neuroscape Applet. Instead, pieces of DataFiles that are needed at various times will be provided via this Module's public interface. The interface provided by this class consists of methods for getting, setting and clearing the contents of the buffer that holds the data, and methods for getting information about the buffer or its contents (size of data in buffer, attributes, correctness). DataFile objects will be managed by the DataManager. They will be created

either by the DataManager when a Query registers its references with the DataManager, or when a Query directly inserts the result of making its query into a new DataFile object, which is given directly to the DataManager for registration.

Viewer Module

A Viewer Module is a Module type that encapsulates the functionality needed to provide a method of displaying the content of a DataFile, directly or indirectly, in a way that is meaningful to that particular data type. Viewers can show data to the user in either a textual way (such as a text window), a graphical way (such as a 3-D simulation or a 2-D graph) or perhaps both. The services they provide can be practically anything, but they must be able to accept as input data of a particular type or types that are made available via DataFiles retrieved as a result of a Query. Access to Viewer Modules is provided via the ViewerManager.

Dependency can be a significant problem when creating a Viewer Module. The way a Viewer provides its functionality is dependent on what format the user would like presented by it. As a result, the level of meaningfulness a Viewer can provide for a given DataFile type will vary according to the complexity of the Viewer Module's implementation and the nature of the information that it can interpret. Basically, a Viewer Module implementation will encompass a trade-off between providing a broad array of services for viewing a small number of similar DataFile types, and providing a smaller but common array of services for a larger number of DataFile types.

Viewers that land at either end of this spectrum are useful in different situations. A Viewer that can display one particular DataFile type and allow

detailed manipulation of that type in a very customized environment is helpful for a detailed examination of that one type. Clearly, though, it has controls and other features that are not applicable to other types. Alternately, one could make a viewer that could accept some type of intermediate format into which other types could be translated, which would be helpful for comparative analysis of some kind and would possibly be useful for providing rudimentary capabilities for any appropriate type in a shorter amount of time. A more general type of viewer would probably not be as helpful for detailed analysis of a given DataFile type as a viewer designed specifically for that one type. This will depend mainly on the complexity of the DataFile type in question. If both flexibility and power is wanted, a viewer can be implemented to provide both capabilities, but the complexity of that Viewer Module will increase dramatically. A Viewer of this type will encapsulate the functionality of several special purpose Viewers rolled into one.

In terms of the Neuroscape system and its pervasive spirit of open communication and minimized dependency, the compromises that need to be considered might seem to be a liability. But since Neuroscape has a modular architecture, if the system is used in certain kinds of work that require a Viewer that is more focused in its usability, then it can be incorporated into the system easily. Alternately, a Viewer that can show a type into which other types can be translated easily can provide functionality that allows viewing of a broader range of DataFile types with potentially less effort than implementing a different viewer for each type. Also, it can be useful for comparison of different data types within a common space.

Viewer Modules provide a relatively simple common interface. There are methods to show, hide and unload a DataFile by reference. There are

methods to retrieve a list of names of DataFiles that are loaded, shown, or hidden. Additional functionality pertaining to the manipulation of the DataFile once it is loaded into the Viewer is provided in one of two ways. The Viewer can provide additional controls itself, which implies that the number of types the Viewer can display and manipulate is more restricted (possibly very restricted). Alternately, additional controls can be provided by a Translator that translates a DataFile into some other type.

Translator Module

A Translator Module is the Module type that encapsulates the functionality to allow translation of a DataFile of one type into an equivalent representation in another type format. As was mentioned above, the Translator Module also can provide a GUI Panel with controls that can manipulate the translated representation of a DataFile in ways that are meaningful to the nature of the input type and possible in a viewer that can display that data translated into the output type. Due again to the drive to minimize dependency, the Translator provides this panel of additional controls because it is responsible for translating a DataFile into another format, and the controls are needed to manipulate constructs that appear in that representation as a result of translation. Only one Translator Module would need to be written for each input/output type pair. At the time of this writing, two Translator Modules that have the same input and output types cannot concurrently exist in the TranslateManager.

Translator Modules are used indirectly via the TranslateManager interface. Translators are never referred to directly by name. Rather, they are characterized by their input type and output type, which together repre-

sents their signature. The TranslateManager categorizes all Translator Modules according to their signature. As was stated above in slightly different terms, two Translator Modules with the same signature cannot both be accessible by the TranslateManager. The interface provided by a Translator is simple. Methods are provided for getting the input and output types, for translating a DataFile's DataBuffer, and for returning the GUI Panel that is to be used for control of a translated DataFile within a Viewer.

Computation Module

Derived data sources are an important capability of Neuroscape. Computation Modules form one part of a derived data source, the other being the contents of DataFiles and other parameters used as input to the Computation. A Computation accepts a list of data of some type, performs some kind of computation on it, and returns the result of that computation. This type of data source is different from other types of data sources because the source of the information is internal to the Neuroscape system. Derived data sources are important because there are types of data that either simply can't be pre-calculated because it would be impractical to store all possible pre-calculations in an external source, or was not done because its importance was not foreseen. Thus, this data is derived from other sources dynamically, on a per-request basis.

Access to Computation Modules is provided indirectly via the public interface of the ComputationManager, much in the same way that the TranslateManager acts as an organizer and arbitrator of access to all Translators. Thus, Computation Modules are reclusive, and stay within the confines of the ComputationManager. Since they are simply wrappers around

some type of function, they never need to be instantiated. Their interface is provided as a static (class) method. Their interface is probably the smallest of all Modules, providing one method for performing the computation.

DataFileReference - A Module Proxy

A DataFileReference is not actually a Module. It is a proxy for an instance of a DataFile Module that exists in the DataManager. A DataFileReference serves as a lightweight transportable representation of a DataFile being maintained by the DataManager. There are many instances when DataFiles need to be manipulated or examined in the Neuroscape system, but it is acceptable to examine or manipulate an object that has all the information about a DataFile, without having its DataBuffer and other data maintained internally by a DataFile Module implementation carried around as well.

There is another benefit of using DataFileReferences. All DataFile instances are centrally located in the DataManager. Having multiple DataFileReferences referring to a single DataFile mean that pointless redundancy of effort or storage space will be minimized. This situation can happen in two contexts. First, if a user made two separate queries that were identical, and the Results were actually DataFiles, then a duplication of storage would occur. If the DataBuffers were filled for each of them, the problem would be compounded. Second, if multiple users are all making queries of a common data source, only one copy need initially be gathered, which means that responses for other users will be faster, because the DataFiles have already been created.

The interface provided by the DataFileReference class consists of meth-

ods for acquiring certain information about the particular DataFile this DataFileReference represents. Methods for getting the DataFile's name, type, the name of the source Query used to get the DataFile, timestamp of last tested availability, and the DataFile's Attributes allows a relatively complete yet lightweight description of the DataFile to migrate throughout the system. Truly, the DataFileManager is the workhorse communications object of Neuroscape. It is passed back and forth between Managers on both the Neuroscape Applet and ProxyServer sides of the system.

Managers

There are six types of Managers in the Neuroscape system. Each contributes some type of organizational or logistical service to the overall system. Five of the Managers are all directly related in the inheritance hierarchy, and the sixth is a Manager that provides logistical support to the other five. Also, five are made available as Java RMI remote objects, and are thus ProxyServer-side, while the sixth is Applet-side.

Manager Base Class

The Manager base class is an abstract parent class representing a software entity that manages the use of, and access to, a set of Modules of a particular type. Each manager has a specific set of services it provides to the Modules it manages, or tasks it performs on those Modules, or both. It also provides core functionality that all managers can use, and an abstract interface that all subclasses of this class would need to implement. The Manager class has only two children: ModuleManager and ServiceManager.

Similar to the abstract Module class, very little actual functionality is provided by the Manager class. In fact, the interface is currently made up of utility methods. However, as with the Module class, since it is a parent of all Managers, it provides a place for future extension and expansion, and most of its importance is as the root of an inheritance hierarchy. For now, there are methods for maintaining, accessing, and clearing a list of errors that the Manager has experienced. There is also a utility method for having additional debugging console output written to the console if a boolean flag is set.

ModuleManager

The ModuleManager is a Manager that loads and registers all of the Modules and their dependent classes into the Neuroscape system at start-up time. Registration involves testing to see if a particular class loaded by the ModuleManager is actually of one of the Module types. If it is, then it adds its name to the list of the Modules for that type. There is a list maintained for each type. Other Managers get their Modules from the ModuleManager, rather than loading them themselves. Its interface consists of one method that returns a list of Modules given some particular type. The ModuleManager is relatively simple from the outside, but its job is key to the way that the Neuroscape system works.

ServiceManager Base Class

The ServiceManager base class provides an abstract parent class for all of the Managers that comprise the real core of the Neuroscape system. Again, this is a simple parent class that is as much a place holder in the inheritance hierarchy as it is a class that provides actual functionality. Ser-

viceManagers maintain a timestamp of the last time they asked the ModuleManager for a list of Modules of the type that they are to manage. Its interface consists of four methods. One returns a copy of the list of names of Modules being maintained. Another returns the timestamp of the last time the ModuleManager was asked for a fresh list of Modules of that type. The third method tells the ServiceManager to get a fresh list of Modules of its particular type, and the last method returns a particular Module by name.

QueryManager

The QueryManager is the entry point through which any new external data coming into the Neuroscape system must first pass. It is the I/O controller for the system, and its services are tightly bound to the functional capabilities of the Query class. Since Query Modules represent the specific interface between a data source and the Neuroscape system, it is logical that the QueryManager would be the I/O manager. The QueryManager is made available to any number of Neuroscape Applets as a Java RMI remote object. It is possible that a single QueryManager could be an I/O bottleneck into the Neuroscape system if enough Neuroscape Applets were making requests at one time. However, other services would degrade in quality as well, and this would indicate it is time to make another QueryManager available on a different host.

The QueryManager provides three distinct sets of services to the Neuroscape system. First, it manages and arbitrates access to new instances of Query Modules. Second, as was described briefly in the description of the Query Module, it provides a remote execution environment to the Applet where a Query's remote methods can be called. Finally, it provides an

interface for use by the DataManager, which manages all data stored internally to Neuroscape. The first set of methods of the QueryManager's public interface consists of a method for getting a list of instances of available Query Modules. This is used by the Neuroscape Applet as the first step in constructing a "pad" of new query forms for the user to browse. The second set, which provides the proxy execution environment, consists of a method that accepts completed queries from the Neuroscape Applet and submits them for the Neuroscape Applet. When the action is completed, the results are examined by the QueryManager, and the DataManager is contacted to register the Query's references (ResultList) or register a new DataFile (ResultItem), depending on if the query returned any results. The Query is then returned to the Neuroscape Applet. The third set consists of two methods that actually call class methods of a Query Module with a particular name (which is maintained in the DataFile and DataFileReference classes as the variable Source), in order to either test the availability of the content of a result, or retrieve the content itself.

DataManager

If the QueryManager is the I/O controller for Neuroscape, then the DataManager provides Neuroscape's main memory. Any named block of data that is maintained by the system is stored within DataFiles that are all managed and maintained by this ServiceManager. Thus, any new data or references to data that are coming into the system must be registered with the DataManager. Copies of a DataFile's DataBuffer can be requested, and they can be translated if it is necessary and a suitable Translator exists in the TranslateManager. But the place that the main copy of the data lives

within the system is the DataManager.

New DataFiles are added to the internal DataFile storage structure through the act of registration. As was explained earlier, the result of a Query Module performing its makeQuery() task is either a single new DataFile or a list of DataFileReferences. If the result is a DataFile, then that DataFile is registered by submitting it to the DataManager, and a DataFileReference for that DataFile is returned to the Query. If the result of a query is a list of DataFileReferences, then the list is submitted to the DataManager for registration. In this case, the DataManager will test the availability of each of the references' data content through the interface provided for this purpose by the originating Query. If the availability test succeeds, a new DataFile object is created for the type that the DataFileReference specifies. If the type is not known, then the type used is the default UnknownFileType DataFile subclass, provided for just this purpose. After the DataFile is instantiated, a DataFileReference for the new DataFile object is created and appended to a list of new DataFileReferences that are available. However, if the availability test fails, no DataFile is instantiated, and thus no new DataFileReference is created and added to the list. The result of registering a list of DataFileReferences is the list of new DataFileReferences that refer to DataFiles in the DataManager whose data is available for loading.

Notice that only the availability of the data to be loaded is tested. The DataFile's DataBuffer is not actually loaded at this time. Loading does not occur until the first request for that DataFile's DataBuffer arrives. The DataManager delays loading the data until it is needed because it doesn't know if any user will want the content of that DataFile. This scheme com-

plies with the earlier mentioned goal of the system to delay action until it is first requested. Once a DataFile's DataBuffer has been filled, the DataBuffer's usefulness to a user has been proven, and the DataManager will allow the DataFile to keep it for an indefinite period of time. A more complex caching scheme might be better under higher loads, and methods have been provided by the DataFile to allow this to occur. For now, simplicity seemed best.

A single, read-only copy of a DataFile is maintained by the DataManager for use by each of the Neuroscape Applets that are in communication with it. There are two advantages to having a single central data repository within the Neuroscape system. The first advantage is that it minimizes unnecessary resource usage. No reasonable networked software system casually passes data files of any type across a network connection. Network links tend to be comparatively slow, and data files tend to be large, and this is especially so with scientific data. Also, regardless of the computer and networking technology used and the size of the data being manipulated, any well-designed software system should use resources as efficiently as is practical. The nature of the tasks that Neuroscape performs and the design of Neuroscape as a distributed architecture rather than a monolithic application both led to the use of a centrally located DataManager. The primary use of Neuroscape is as a tool for interactive discovery and inquiry. Exploration is characterized as much by where the explorer tried to go but had to go another way. If the path to the goal were clear and wrong turns were never taken, exploration would not be necessary. But the user should not pay with a degradation in performance for exploration. Large masses of data should be downloaded only when the user really wants it,

and not every time that a result comes back from a query. If the files are loaded on demand, and the first inquiry the user makes yields a result the user is happy with, and the user chooses to load all of the results into a viewer, then the result is exactly the same from the user perspective as if they had all been loaded whether he wanted them or not. But, as might be more common, if the user made a query and was disappointed with the results, she should not have to wait for all of the data attached with that result to load before making a new query.

The other advantage is that it minimizes a duplication of effort on the part of the Neuroscape Applets, much in the same way that an HTTP proxy server limits the pointless duplication of requests by caching DataFiles. If one user running a Neuroscape Applet makes a query and gets a result, all of the elements of that result that are available will have DataFile objects constructed for them. If another user running a separate Neuroscape Applet makes a query and gets a result list whose intersection with the first user's result list is non-empty, those entries that are within the intersection have already been tested, and duplication of effort for those will be eliminated. An analogous savings of effort is achieved when one user has requested the DataBuffer of a particular DataFile, and then another user goes to access it. It has already been introduced into the Neuroscape system, and is more readily accessible.

The Query Modules being manipulated by each Neuroscape Applet using this DataManager maintain their own list of DataFileReferences. So a copy of the DataBuffer is not necessary until the user wishes to view it in a Viewer or use it as a data source in a derived data query. Either way, there are never changes that are directly made to the DataFile's DataBuffer, so

the complex concurrency and coherency problems that arise from systems that provide centralized resource access will never occur as a result of DataFile access through the DataManager.

The interface provided by the DataManager is quite simple. There is a method for registering a list of DataFileReferences, and one for registering a single DataFile. Both of these methods return DataFileReferences if registration was successful. The method for registering a single new DataFile should always return a new DataFileReference because the DataFile will always be registered. There are also several methods for getting a copy of a given DataFile's DataBuffer corresponding to a given DataFileReference. This interface enforces the "write-once, read only" storage characteristics that the DataManager is intended to provide.

ViewerManager

The ViewerManager is unique among ServiceManagers in that it resides Applet-side. This means that there is a separate ViewerManager for each Neuroscape Applet. A ViewerManager is instantiated and started by a Neuroscape Applet when the Applet is initialized. There are 2 reasons for the placement of the ViewerManager Applet-side. First, a Viewer presenting data to the user is inherently an activity that is conducted Applet-side, and not remotely. With a few exceptions, two users will usually not want to share a display. An exception to this is the recent interest in computer collaboration. Collaboration capability was not a part of the initial design of Neuroscape because the Neuroscape system was intended as a tool for students in laboratory classrooms. Also, adding this capability would certainly add to the complexity of the system, because the possibility for mul-

tiple user access opens up many issues, such as access privileges and authentication. Nevertheless, it should be possible to add it into the system while maintaining the integrity of the original design goals.

The way that the ViewerManager exists Applet-side involves Java's dynamic abilities, as described above. An important part of the Java platform, and a significant reason for Java's utility in networked object oriented software systems, is its ability to delay binding of executable functionality until that functionality is needed. The importance of this description of the late binding characteristics is that some Viewer Modules may need to bind to implementations that have some platform dependency. This is especially true of graphics functionality, which is one of the biggest hurdles in the way of platform independent visualization capability from a single code base. Since Viewer Modules can be loaded into the ViewerManager Applet-side simply by using the same call to ModuleManager.getModules() that all other ServiceManagers use, it was a clean solution to a potentially messy problem.

The second reason for putting the ViewerManager Applet-side was to ensure efficient use of resources while still maintaining a reasonably friendly public interface. In this case, the resource use being minimized is network traffic. Every time that a Query is submitted to the QueryManager, and the result that comes back is not empty, the ViewerManager is asked for a list of Viewers that can be used to show data types that match the results' types. Since the ViewerManager simply passes back a list of names, the network time needed is relatively small, and little savings is realized. Other savings occur because Viewer Modules, which ultimately are Java classes, are brought Applet-side, rather than instances of Viewer Modules. This

difference is analogous to the difference between faxing somebody a blueprint for an object, and sending them the actual object. It will always be cheaper to send the blueprints. It is possible that the system that is Applet side is slower at making that object, but perhaps it is not. As was mentioned above, it is also possible that the system where the ProxyServer is running is not able to make that object, but the system running the Applet is. Ultimately, the Viewer must be able to be instantiated and used Applet-side.

The last reason was simply to experiment with having one of the ServiceManagers be Applet-side. All of the Managers were designed so that it would not matter whether they were running on the same system as the Neuroscape Applet, or remotely half way around the world. Any or all of the Managers should be able to be moved around however it is deemed necessary or most advantageous for better performance or whatever criteria is important.

The interface provided by this Manager consists of two methods. There is a method that returns a list of names of Viewers that are usable for a given type. The list includes Viewers that can display DataFile of a given type directly, and Viewers that can display data of a type for which there exists a translator that can convert the DataFile's type into the Viewer's type. There is also a method for getting a new instance of a Viewer Module by name. Both of these methods are called by a particular ResultWindow object in the Neuroscape Applet, and both are involved in the process of providing a selection of different Viewers the user can activate, and activating it.

TranslateManager

The TranslateManager arbitrates requests for translation service through its public interface. It never allows direct access to a Translator. Instead, it takes requests for translation and forwards them to the appropriate Translator. Again, as was stated above in the characterization of Translator Modules, two Translators with the same signature cannot both be accessible by the TranslateManager. Its public interface also includes two methods that provide lists of either input types given an output type, or output types given an input type. These two methods constitute specific types of queries (unrelated to the queries a user makes) that can be made to retrieve information from the TranslateManager about the Translator Modules it has registered. At start up, the TranslateManager gets its Translator Modules from the ModuleManager, and then sets up a set of data structures that cross-reference the Translator Modules it has registered. These data structures allow quick responses to the two latter methods.

ComputationManager

In a way, the ComputationManager is very similar in its design to the TranslateManager, except that it is simpler. Like the TranslateManager, it arbitrates access to Computation Modules by never allowing direct access to them. The Modules that they manage are quite similar as well, in that they are both wrappers for functions of some type. Translator Modules are wrappers for parsing functions that accept a blob of data of one type, and return an equivalent representation in another type. Computation Modules are also wrappers for functions, but they encapsulate more generic functions, and, as was described above, they accept as input a list of param-

eters packaged in a Java Vector, which is a container class that can contain a list of heterogeneous data sources of arbitrary length. The ComputationManager's role in the system is to provide a way of managing Computations, which are used by Query Modules as alternate data sources for acquiring data that is derived from other data, and not permanently stored. The ability to generate new data from data that is currently available provides an important way of extending the capabilities of the Neuroscape system.

It is because of the context dependency of the input to a Computation that the ComputationManager can be simpler, because much more of the burden of testing the input lies in the hands of the implementation of the Computation Modules themselves. Thus, the ComputationManager provides only one method in its public interface. This method is peformComputation, and it simply takes as input a name of a computation and returns the result of the computation as an array of bytes.

Neuroscape Applet

The Neuroscape Applet is the user's interface to the Neuroscape system. Making the interface available as a Java Applet permits it to be used by any user using any computer with a Java-enabled Web browser that implements a certain level of Java functionality. Examples of Web browsers are Netscape Navigator 4.5, Netscape Communicator 4.5, and Microsoft Internet Explorer 4.0.1 or 4.5. Java capabilities of the Web browser must provide compatibility with the Java 1.1 Abstract Windowing Toolkit (AWT), the Java 1.1 AWT Event model, and support for Java RMI (Remote Method

Invocation). It should be noted that the Microsoft JVM used in the Windows 95/98/NT version of Internet Explorer is not able to run Neuroscape, since Microsoft decided to not provide support for Java RMI.

It is also the primary executable entity operating Applet-side (hence the term). The Applet establishes the communication between itself and the ServiceManagers. It also starts an instance of the ViewerManager Applet side. The Neuroscape Applet provides a windowed, Graphical User Interface (GUI) that allows the formulation of queries, and the examination, manipulation, presentation and organization of the results of those queries. The interface will be described in more detail in a later section, but it is important to recognize that the GUI components are as integral a part of the system as the Modules and Managers. They also were created to provide maximum flexibility to the user. There are three primary window types that a user will encounter: the Main Control window, a QueryDialog window, and a number of ResultWindows.

The Main Control window is the root window of the Applet. All user action starts by interacting with this window. It provides a basic starting point, and administrative capabilities. Here will be found a button for making a new Query, GUI components to examine and manipulate the list of ResultWindows, and a Help button. It was intended to be simple, and it is.

The QueryDialog provides a virtual pad of query forms. It displays the QueryPanel for each Query that is available at the time the window is opened. The user may flip through the pad by selecting any one of the choices is displayed in the selection component at the top of the window. A line of buttons is at the bottom of the window, labeled "Cancel" "Reset" and "Submit". For whichever Query's QueryPanel is being displayed currently, the

user may fill out the form. When the form is filled out sufficiently so that it is deemed submittable by the Query Module that owns it, the Submit button will be enabled, and the user can click on it to submit the query. Canceling the query throws away the window, and clicking the Reset button resets the topmost QueryPanel to its default state.

A new ResultWindow is created and displayed whenever the submission of a Query was successful in retrieving a list of results that is non-empty. This non-empty list will be displayed in a list display component in the window. There are controls for manipulating the list, and also controls for activating Viewers that are either usable on any of the types of the results in the list, or usable on results of any type. Currently, only one Viewer may be activated for each ResultWindow. There is also a space where the DisplayPanel from a Translator may be displayed.

ProxyServer

Strictly speaking, the ProxyServer is not an actual component of the Neuroscape architecture. It is a Java application that sets up the remote execution environment. It instantiates the Managers that exist as remote objects, calls their communications initialization methods, and registers them with the RMIRegistry on the host system where they will execute. If there is not an RMIRegistry running on that host system, it will attempt to start the system's, and if that does not exist, then it will start one internally as a last resort. If there are objects that are already registered with the names that the ProxyServer tries to use, it gives up because the Neuroscape Applet needs to know the names the ProxyServer used to register them. This

utility application would be unnecessary if either the ProxyServer-side managers could be started autonomously, or if CORBA was used instead of Java RMI, because, like DCE, the Managers would not need to stay in process all the time. The equivalent of the RMIRegistry would instantiate them on the fly. Changes to the RMI system for Java 1.2 should also provide this service.

THE NEUROSCAPE SYSTEM: A USER'S PERSPECTIVE

Rigorous specification is required for understanding the details of how a complex system like Neuroscape actually functions. But a description of the interface with which a user will be asked to interact is equally important. Here we present descriptions of the fundamental processes that a user will perform with Neuroscape.

Making a Set of Queries

Making a query is the first step to using the Neuroscape system. When the user wishes to pursue some new line of inquiry, she will click on the "Make New Query" button on the Main Control Window (fig. 1). When this button is clicked, the Neuroscape Applet will present the user with a QueryDialog (fig. 2). The user can select any one of the available Queries found in the Query choice control. If a data source is unavailable for query, a default QueryPanel will be displayed to the user (fig. 3). The QueryPanel for the selected Query will be displayed below, so that the user may fill it out. The state of the QueryPanel can be reset to its default state by pressing the Reset button. When the QueryPanel has been sufficiently filled out by the user, the Submit button will be enabled, and the user may then click on the Submit button to submit the Query for processing. If the Result of a query is non-empty, then a new ResultWindow will be created and presented

to the user (fig. 4).

Managing Results

Any Queries that have non-empty results will have a corresponding ResultWindow (fig. 3). The set of all ResultWindows will be listed in the Result list on the Main Control Window (fig. 1). This set can be managed via controls also provided in the Main Control Window. Results can be discarded or brought to the front. New Results are added by making a new query, if the query has a non-empty result.

Viewing Attributes of Results

The attributes of a particular DataFile can be viewed by selecting one of the elements in the ResultList in the ResultWindow and clicking on the "Attributes" button (fig. 3). This will display a window that presents the attributes of the DataFile that entry in the list represents.

Viewing Results with Viewers

Available Viewers can be chosen and activated by using the Viewer choice control on a ResultWindow. The Viewer choice control will only have Viewers that can display the DataFile types present in the Result, either directly or indirectly via translation. Currently, only one Viewer can be activated at a time. Once a Viewer has been activated, content from a DataFile can be loaded.

Getting Help

Clicking on the "Help" button on the Main Control Window (fig. 1) will bring up the Neuroscape Help page in the default browser for the user on the user's host computer.

AREAS FOR FURTHER DEVELOPMENT

The current version of the Neuroscape system is fully functional. As with any system, there are things that can be improved. Some of these recommendations for further development work on the Neuroscape system come from the fact that the system was implemented in only five months, by the author alone. The current version is as much a proof of concept as it is anything else.

Maturation of Results Presentation

The first additions to the system should be the expansion of the services the ResultWindow provides to present the result of a query to the user. The ResultWindow currently provides a very basic set of functionality. For instance, currently only one Viewer can be activated at a time for a particular ResultWindow. This limit was made simply to avoid having to implement the management of state information for multiple active Viewers by the ResultWindow. There is a co-dependent relationship between an active Viewer and its associated ResultWindow. If a Viewer is closed, the ResultWindow needs to know this so that subsequent attempts to load to it are not carried out.

Also, the ResultList presentation in a ResultWindow should display more information, such as some subset of the result item's attributes, and the

result items in the list should be sortable according to some set of criteria other than alphabetical order. This was a limitation that was set because a simple interface was provided. Clearly, with more concentrated effort, a more powerful interface could probably be developed for that list display.

From Java RMI to CORBA

Second, there could be benefits to using CORBA instead of Java RMI for inter-object communication. Since CORBA is a way of having objects implemented in any language communicate with each other, and Java RMI is a remote procedure call system specific to Java, moving to CORBA could have important benefits. New Modules could be implemented and incorporated into Neuroscape simply by putting a Module "wrapper" around existing programs. This is an important capability of CORBA. It should be obvious that this would contribute to the adoptability of Neuroscape by individuals or organizations that already have a significant investment in software written in languages other than Java. The utility of Modules of this type would be limited to the Neuroscape Applet being run as an application on a machine. These "new" Modules would possibly be platform dependent. The Neuroscape Applet could be adapted to accommodate this possibility, by allowing it to be run as an application on a particular client machine would be very helpful anyway. There is a movement by Sun to tightly integrate CORBA into the Java core libraries with Java 2 (or Java 1.2). Neuroscape could reap significant benefits in terms of flexibility by converting to CORBA, and it would not be complicated to do so. For the sake of a broader audience using Neuroscape, this change in the implementation should

be explored early on.

Making Neuroscape Swing

It would also benefit the system if the GUI components used to construct the core interface were changed from the Java AWT component API to the Swing component API (also known as the Java Foundation Classes). There are several advantages to doing this. The most important advantage is a greater level of consistency in how support for the components is provided on different computing platforms. The lightweight peers in the Swing API aren't dependent on any underlying OS's native GUI component set for functionality like the AWT components are. With a couple of necessary exceptions, they are implemented entirely in Java. As a result, they inherit none of the liabilities of their equivalent components in the AWT, which rely on the native component set of the executing platform. Since the capabilities of these native components differ, there is a lack of consistency that is inconsistent with the desire for a uniform interface across all platforms. Also, the Swing API has a much greater array of pre-made components. This would allow the Neuroscape core interface to have a more professional appearance. It would also allow easier development of more sophisticated Modules. There was an effort to see if the Neuroscape Applet interface could be easily converted from using the AWT components to using Swing's counterparts. Unfortunately, there is not a strict one-to-one correlation between the functionality of all traditional Java AWT components and Swing components. There are differences in the naming, in the way that the functionality for those components is provided. Swing's additions to the

Java event model are also slightly different from their predecessors, and complicated by the fact that Swing component events are not thread safe. Finally, the Swing API is not widely supported by Web browsers' JVMs yet. This is an ongoing problem with developing for the Java platform, but one that hopefully will be resolved soon.

Neuroscape Module SDK and Testing Suite

A part of the Neuroscape system that would be very helpful to developers of Modules would be a Software Development Kit (SDK) or, even better, a suite of JavaBeans, to assist in the development process. It would be very helpful for getting other professors and other schools to use the system if they could develop new Modules themselves, without having to have experience with programming in Java. Creating a higher level development capability also be helpful because it would speed and possibly standardize the process of developing Modules by making it more efficient. However, the problem of providing non-programmers with the ability to program instantly is a difficult one. A partial solution would be limited in its capabilities, meaning that users of this Module construction system would possibly only be able to construct fairly trivial Modules. Still, this would be helpful. A couple of testing utilities, for testing the integrity of a newly made module, would also be helpful for the developer.

Persistent Workspace

Finally, incorporating some way to permit the persistent storage of the state of one's work across sessions in Neuroscape would be very helpful, to students using this in classroom settings. It would also be essential to using this system in a research setting. Adding this capability to the system is a nontrivial task because it means figuring out how and where to store this data, and also means implementing some form of authentication and access restrictions. In this second version of Neuroscape, adding this capability to the system would have prevented it from being completed in time, and it is an added feature to the core services that Neuroscape provides. However, Neuroscape will not be a truly mature system without it, and it should be folded in at some stage.

SUMMARY

Simple Informatics systems have already begun to touch the lives of a large number of people. The World Wide Web could be viewed as the first widely used Informatics system. Data is in standard formats, and a browser application is used to access these different data types, either directly or using plug-ins or helper applications. Initially, the simplicity of the World Wide Web was one of its strengths, because it encouraged rapid growth. But once the Web gained its critical mass and really caught on, the simplicity of the Web became its limiting factor, because it became large enough that a person had a hard time finding the information she was looking for. But since the core system was flexible enough, it was then possible to create ad-hoc services such as Web crawlers and search engines to help find the pages of information she is seeking. Search engines are now a part of the World Wide Web system.

The Neuroscape system will be extended in an analogous way when the data sources begin to share common organizational structures. Neuroscape makes no assumptions about the sophistication of the data source. More sophisticated data sources will mean that the development of Query modules will be easier. Standardizing on data file formats will mean Viewer modules will become more standardized as well. The number of translator modules will grow less quickly with the standardization of data file formats as well. This system can scale and adapt to the nature of the data sources

it connects to. Clearly there is a need to develop the modules, but the utility of the system should grow exponentially with the addition of new modules. All that is required is to create a critical mass of modules, and the system will be usable without the need to create new modules. There are still challenges posed by a system such as this, but it provides a first glimpse into a world where we can be asking questions of any data source and manipulating the answers in common ways.

REFERENCES

[1] Flanagan, David. *Java in a Nutshell,* O'Reilly & Associates, 1997, 3-8.

[2] Budd, Timothy. *An Introduction to Object Oriented Programming,* Addison-Wesley Publishing Co., 1997.

[3] Overland, Brian. *Java in Plain English,* IDG Books Worldwide, 1997.

APPENDICES

APPENDIX A

SCREEN SNAPSHOTS

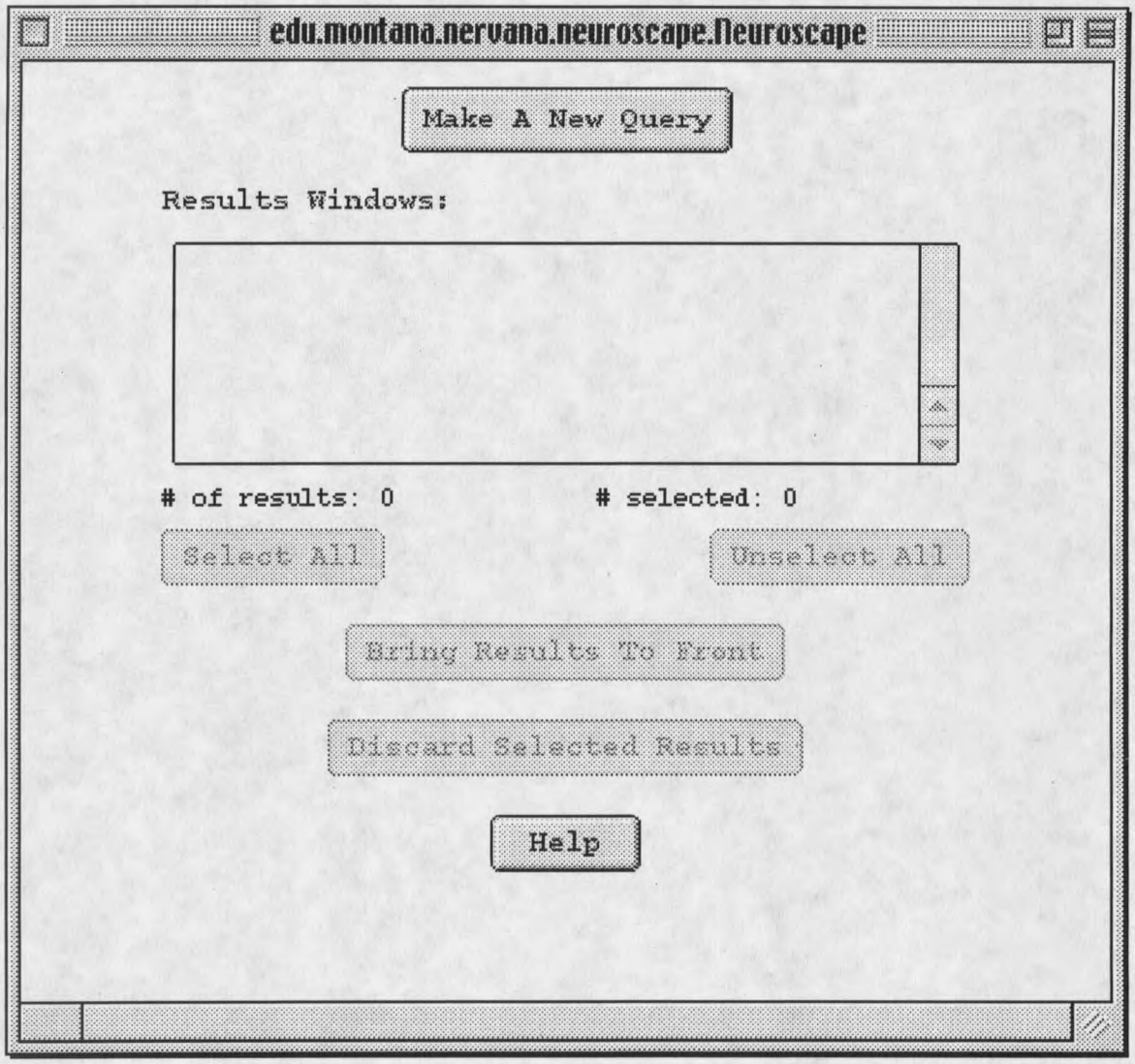

Fig. 1: Main Control Panel

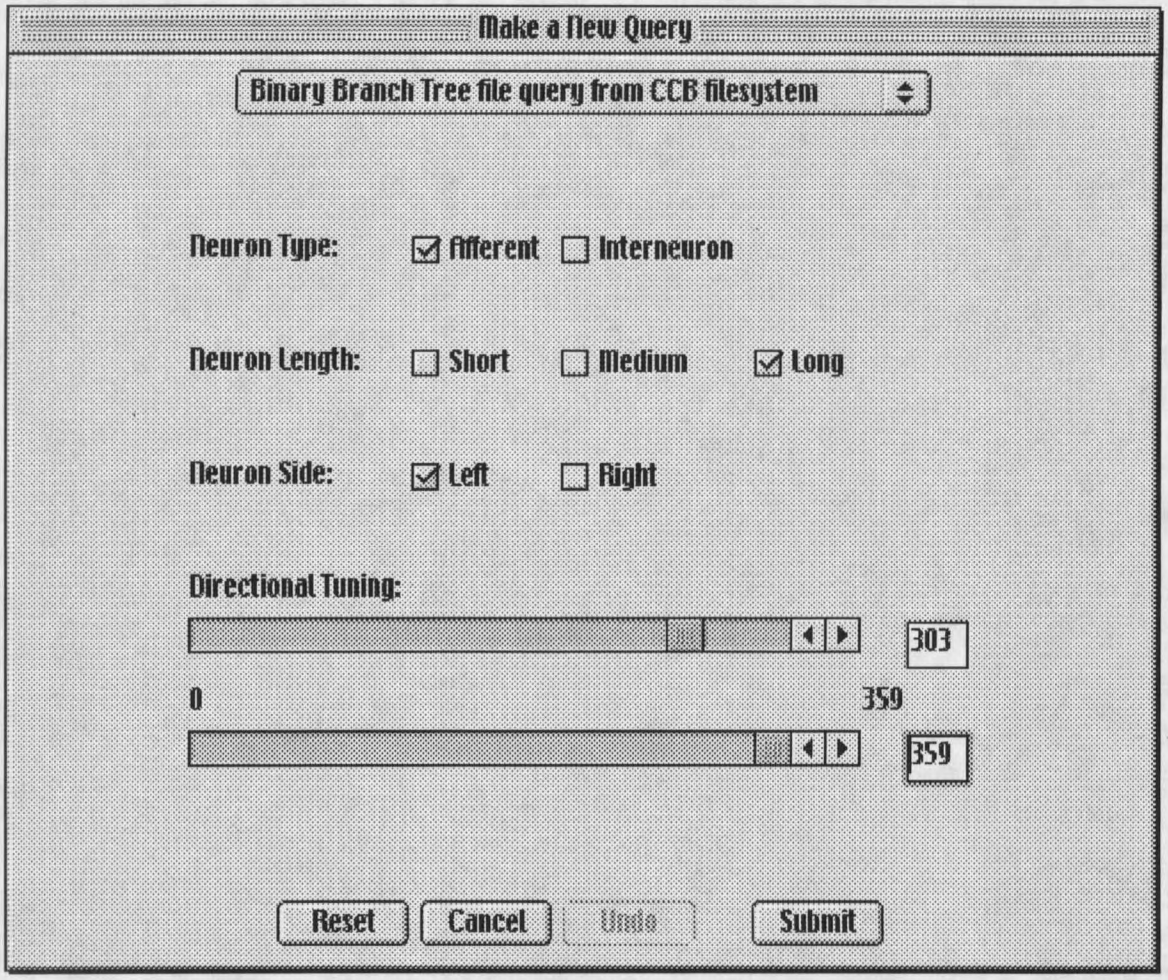

Fig. 2: Filling out an available Query in a QueryDialog window.

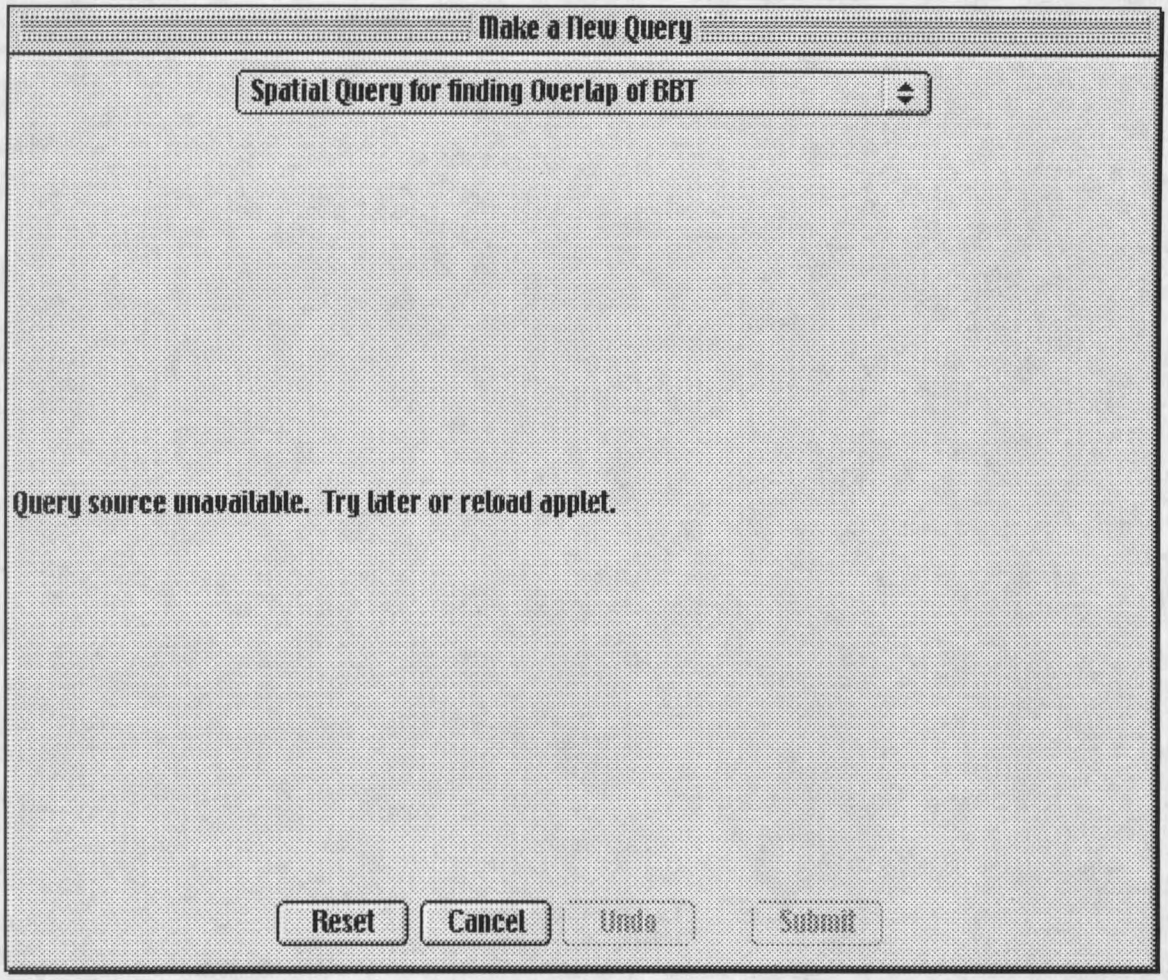

Fig. 3: Unavailable Query in QueryDialog.

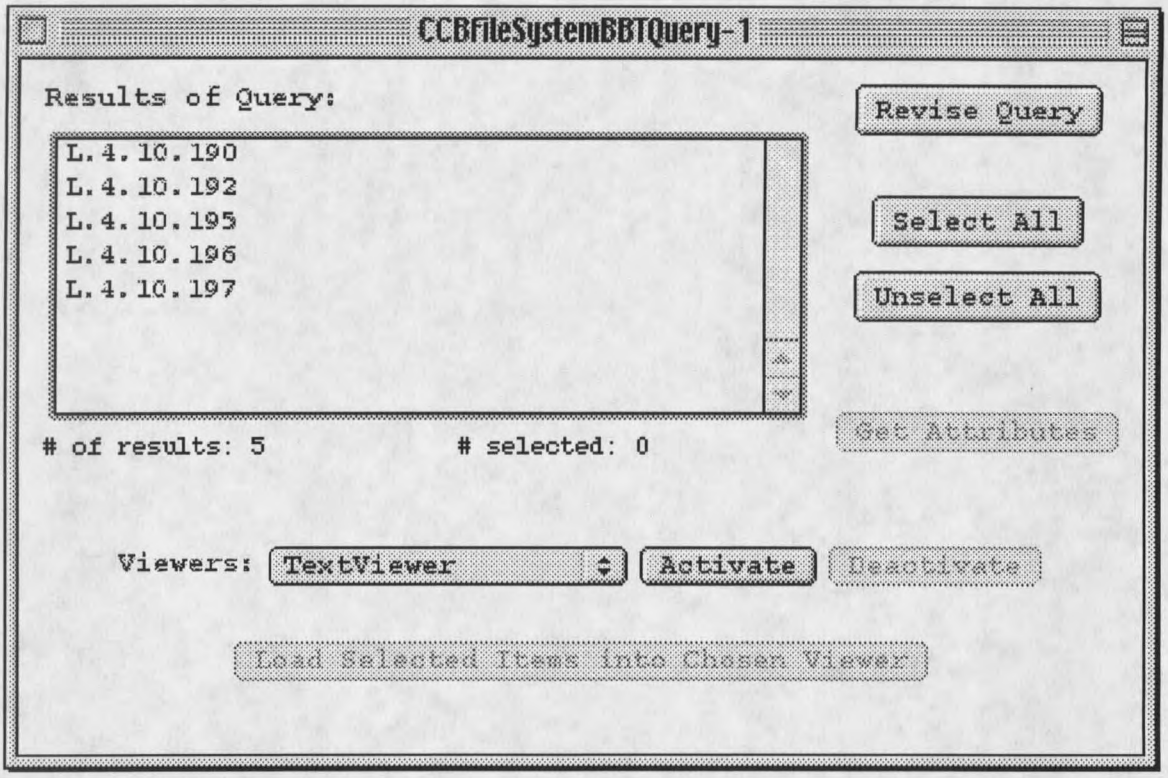

Fig. 4: A ResultWindow displaying the result of a submitted query.

APPENDIX B

SYSTEM PROCESS FUNCTION DIAGRAMS

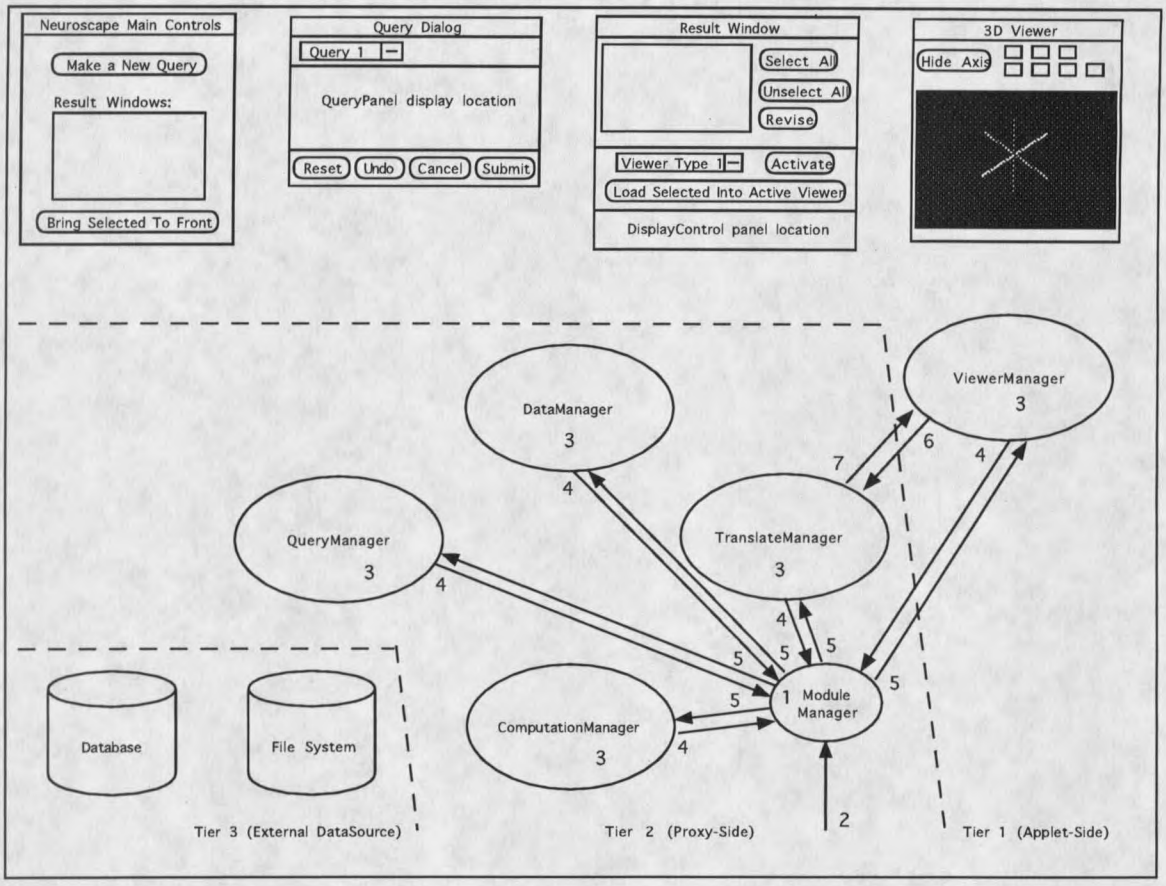

Fig. 5: Initialization of the Neuroscape Managers.

1. **ModuleManager**: This object is created.
2. **ModuleManager**: Loads and registers any modules that are in its subdirectory called "modules".
3. **ServiceManagers**: Object is created.
4. **ServiceManagers**: The object asks the ModuleManager for the Modules of the type that it manages that have been loaded.
5. **ModuleManager**: A list of Modules are returned.
6. **ViewerManager**: After retrieving its modules from the ModuleManager, it asks the TranslateManager for a list of translators that can be translated into types for which the ViewerManager has Viewers registered.
7. **TranslateManager**: Returns the list.

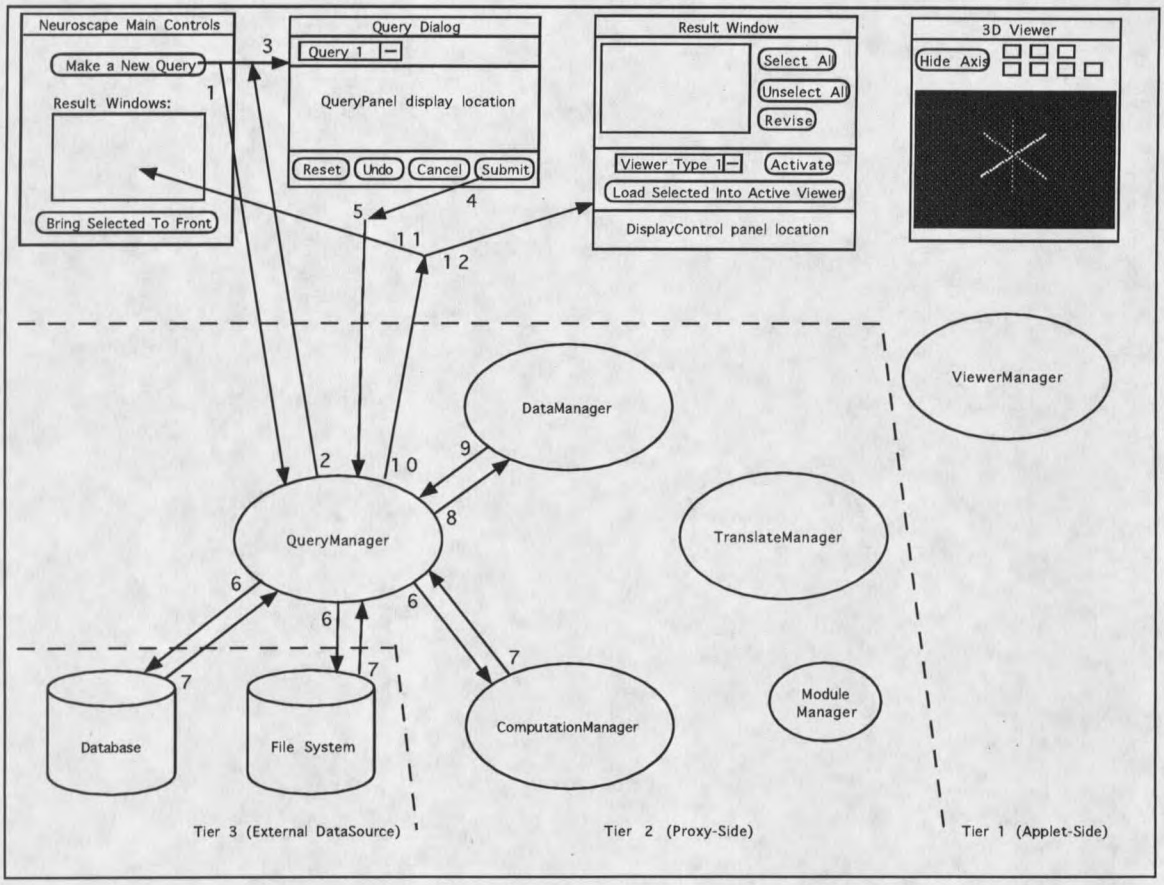

Fig. 6: Making a New Query.

1. **Neuroscape Applet**: Ask the QueryManager for a list of new Query instances, one of each type of Query registered with the QueryManager.
2. **QueryManager**: Create the list of instances, and return them.
3. **Neuroscape Applet**: Make a new QueryDialog, and pass it the list of Query module instances.
4. **QueryDialog**: Let the user enter her query. When the query is filled out properly, the Submit button will be enabled. If it is clicked by the user, return the Query that is currently displayed.
5. **Neuroscape Applet**: Submits the chosen query to the QueryManager for processing.
6. **QueryManager**: Receives the Query and tells it to perform its query action.
7. **Query**: Receives the result of performing its query action, and processes this result.
8. **Query**: Submits the filled DataFile or the list of DataFileReferences to the DataManager for registration.
9. **DataManager**: Returns a list of DataFileReferences that have been registered to the Query.
10. **QueryManager**: Returns the Query carrying the Result.
11. **Neuroscape Applet**: Add the name of the filled Query to the ResultList.
12. **Neuroscape Applet**: Create a new ResultWindow and give it the filled Query.

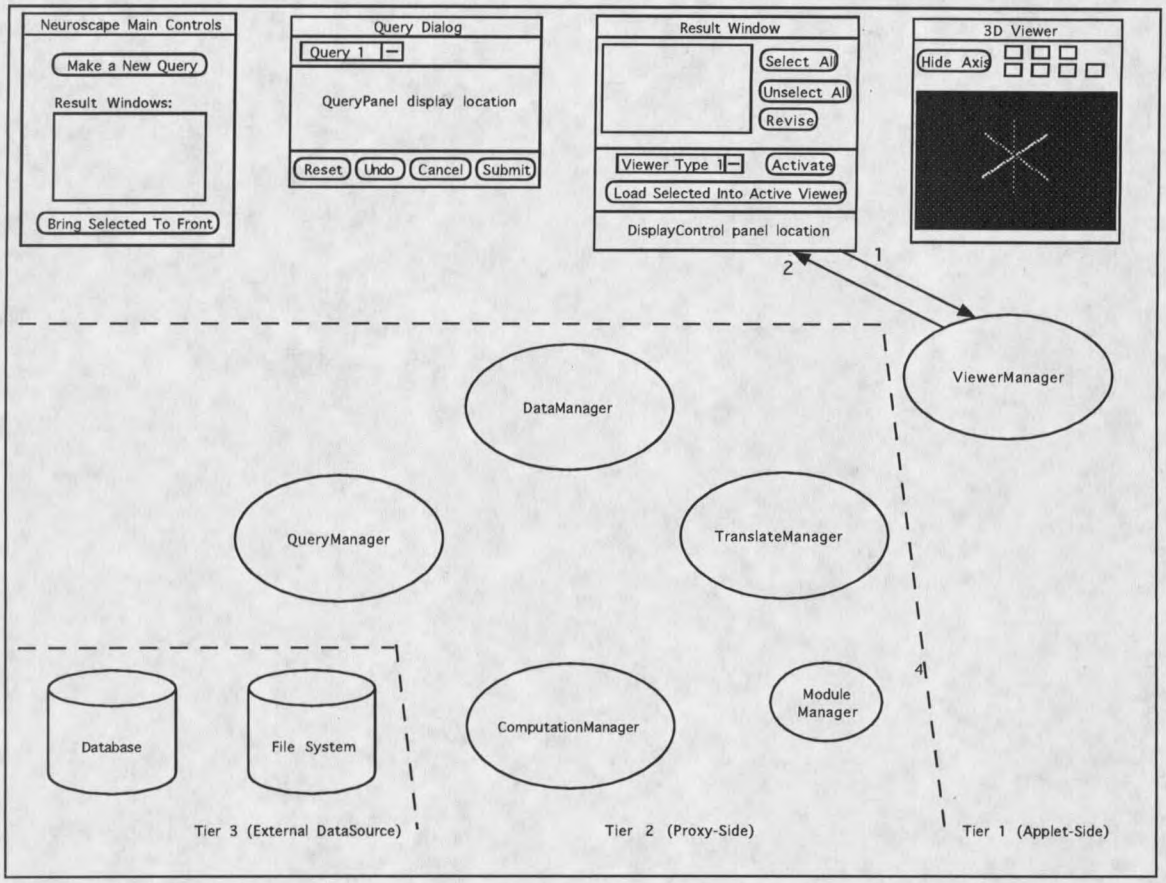

Fig. 7: Creation of a new ResultWindow.

1. **ResultWindow**: Upon creation, ask the ViewerManager for a list of Viewer Types that can be used to view the DataFile types in the ResultList of the Query this window manages.

2. **ViewerManager**: Return the list.

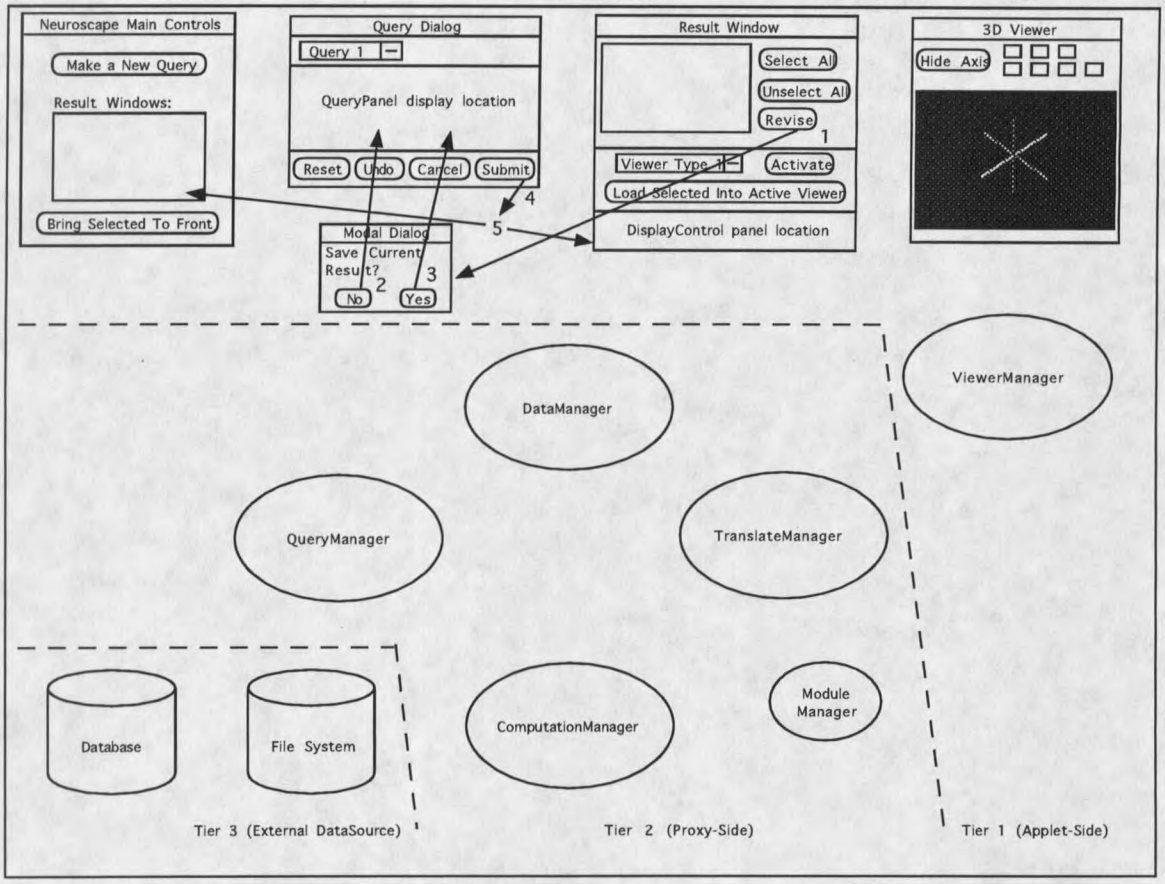

Fig. 8: Revising a query.

1. **ResultWindow**: When the Revise button is clicked, display a YesNoDialog asking whether to keep the current result. If the answer is "No", go to step 2. Otherwise, go tostep 3.
2. **ResultWindow**: Deactivate any Viewers this ResultWindow has activated. Give the Neuroscape Applet the old query to make a new query with.
3. **ResultWindow**: Clone the current query (which makes an identical copy but is nameless) and give that to the Neuroscape Applet to make a new query with. The list of Query modules going into the QueryDialog contains only the cloned query. Proceed as if it were a New Query.
4. **Neuroscape Applet**: When asked to make a new query, do so and submit it if asked.
5. **Neuroscape Applet**: On return, check to see if the Query is named. If so, then replace the old query with the new query. Otherwise, generate a new ResultWindow as usual.

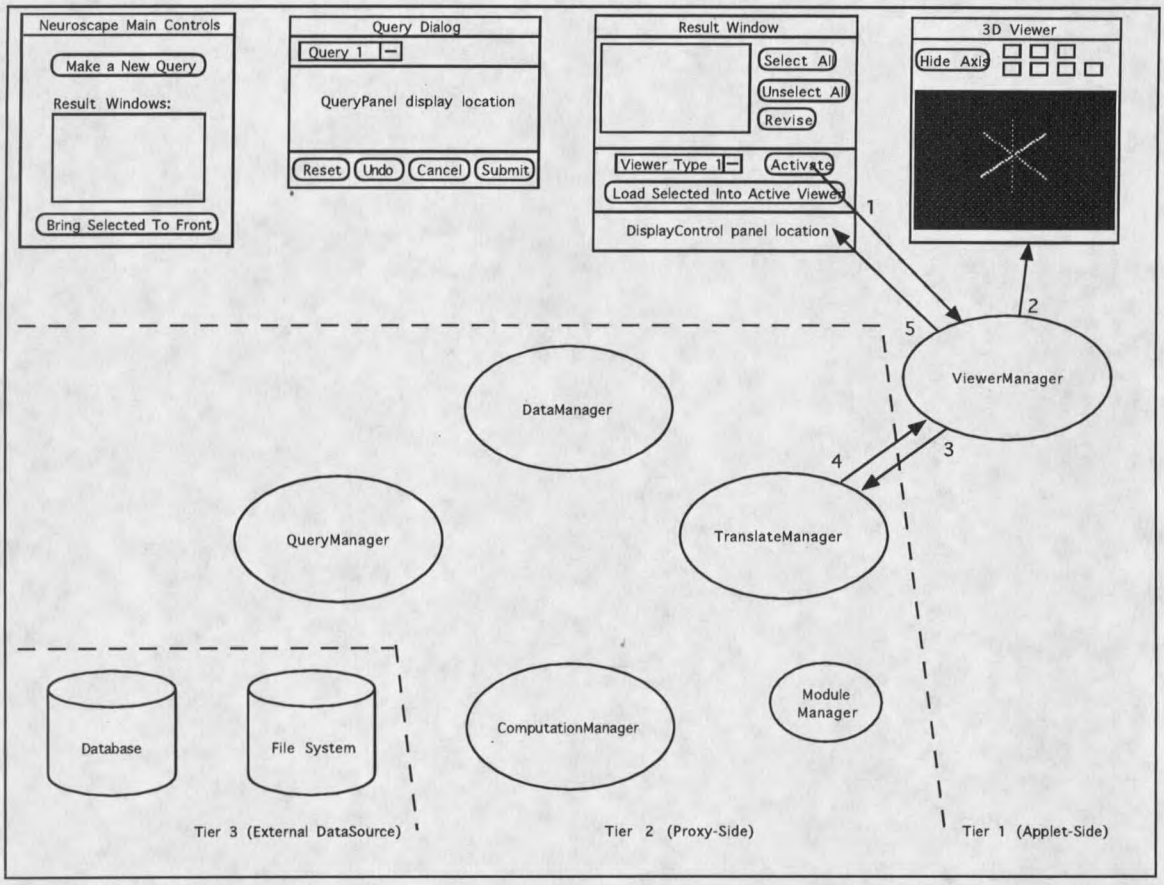

Fig. 9: Activating a new Viewer.

1. **ResultWindow:** Clicking the "Activate Viewer" button asks the ViewerManager for an instance of the chosen viewer.
2. **ViewerManager:** A new instance of that particular Viewer is created.
3. **ViewerManager:** Ask the TranslateManager for the associated DisplayPanel, if there is a need and if one exists.
4. **TranslateManager:** Returns the requested DisplayPane, if there is one.
5. **ViewerManager:** Establishes the connections between the DisplayPanel and the newly activated Viewer. Returns a reference to the Viewer, which has a reference to the DisplayPanel if there is one.

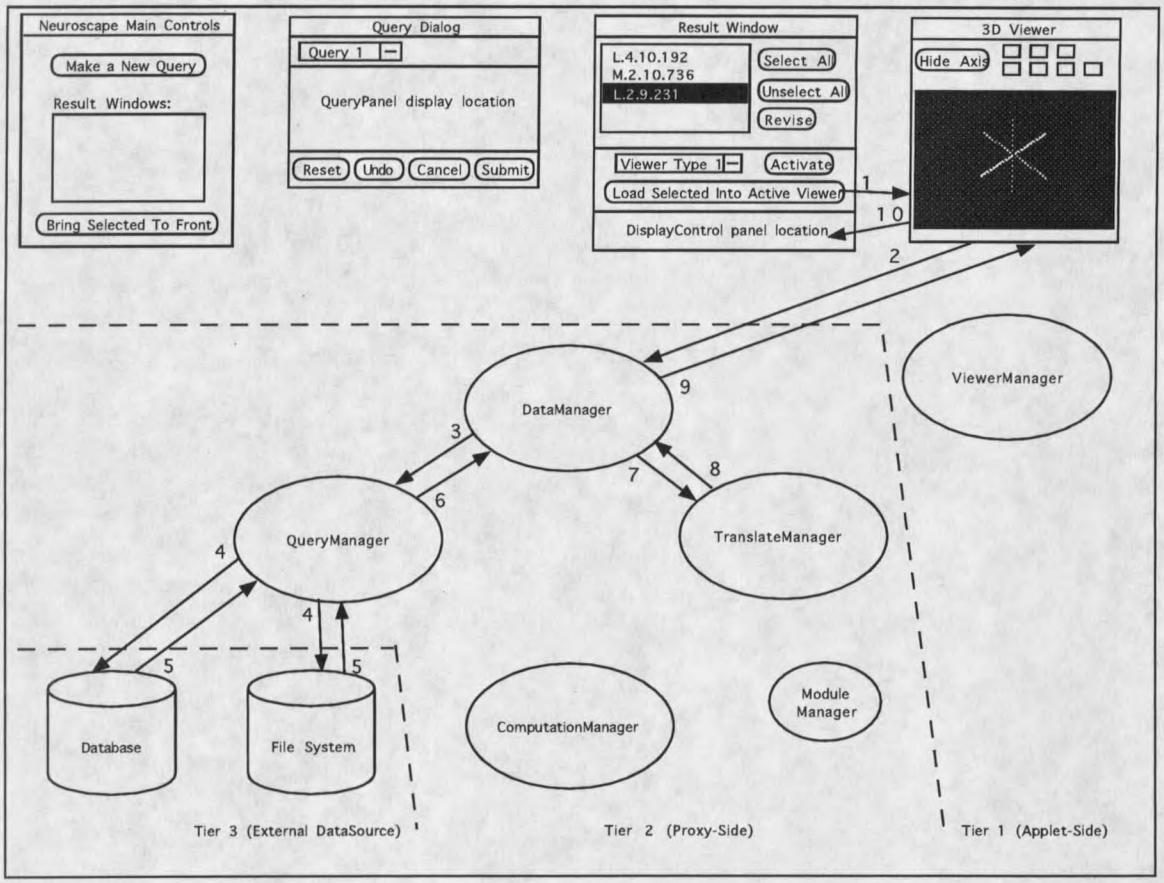

Fig 10: Loading Selected Items Into Active Viewer.

1. **ResultWindow**: Clicking the "Load Selected Into Active Viewer" button passes the list of chosen items from the Result List to the active Viewer.

2. **Viewer**: For each of the items in the list passed to it, ask the DataManager for the DataBuffer of the DataFile corresponding to the list item, in the given type.

3. **DataManager**: If the DataBuffer is not filled yet, then fill it by asking the QueryManager to fill it for you.

4. **QueryManager**: Check the source of the DataFile, and get that Query module to retrieve the content for that DataFile using its getDataFileContent() method.

5. **Query Module**: Retrieves the content and inserts it into the DataBuffer of the DataFile.

6. **QueryManager**: Return the filled DataFile.

7,8. **DataManager**: If the type requested by the Viewer is different than the DataFile's type, then ask the TranslateManager to translate it into the desired format.

8. **TranslateManager**: Returns a copy of the DataBuffer given it, translated into the new type.

9. **DataManager**: Return the DataBuffer, in the requested type.

10. **Viewer**: Update the state of the DisplayPanel on the corresponding ResultWindow.