Fiswidgets
A Graphical Computing Environment for Neuroimaging Analysis

Kate Fissell,*1 Eugene Tseytlin,1 Daniel Cunningham,1 Karthickeyan Iyer,1 Cameron S. Carter,1 Walter Schneider,1,3 and Jonathan D. Cohen1,2

1University of Pittsburgh, Pittsburgh, PA; 2Princeton University, Princeton, NJ; 3Psychology Software Tools, Inc., Pittsburgh, PA

Abstract
Current fMRI neuroimaging software programs offer the researcher a wealth of analysis methods and tools. However, the incompatibilities in user interface, data format, and computing environment in these tools make it difficult if not impossible for most researchers to take advantage of the full set of resources available for neuroimaging analyses. We describe a graphical computing environment, Functional Imaging Software Widgets (fiswidgets), which was developed to address these interoperability and usability problems. This environment provides a desktop style framework into which 100 subcomponents from a number of widely used fMRI analysis software packages (e.g., AFNI, AIR) are incorporated. It is an open-source, extensible environment available for reuse and modification by other software developers. A discussion of the design criteria (open architecture, modularity, wrapper technology, commercial utilities) that support such loosely integrative computing environments, and the problems entailed in maintaining them (development overhead, distribution logistics, format incompatibilities, graphics vs scripting tradeoffs, and appropriate acknowledgment of software developers) follows.

Index entries: Fiswidgets; AFNI; AIR; tal; interoperability; graphical user interface; neuroimaging software; graphical scripting.

Introduction
Since the emergence ten years ago of functional magnetic resonance brain imaging (fMRI) as a tool for neuroscience, psychology and psychiatry research, a large number of software tools for data analysis and visualization have been developed to support this
research. Most of the packages are Unix-based software written by scientists active in the field of MRI imaging, and are distributed freely under a number of different copyright conditions. The primary advantages of this kind of distributed software development effort are that scientists from a diverse set of disciplines (e.g., physics, statistics, and neuroscience) each bring their own specialized areas of expertise and interest to bear on the problems, and thus a large body of highly sophisticated software is quickly developed. The primary disadvantage is that the tools written by different laboratories are difficult to integrate. We have been developing a graphical computing environment, Functional Imaging Software Widgets (fiswidgets), to address these interoperability and usability problems in current neuroimaging software. This environment provides a desktop-style framework into which subcomponents from a number of widely used fMRI analysis software packages can be incorporated. The key informatics problem this project addresses is how to provide researchers with an integrated, consistent, robust set of analysis tools that have been, and continue to be, developed by a very diverse and loosely integrated group of scientists.

This paper will first discuss the fiswidgets computing environment as one implementation of a model for integrating neuroimaging software. Second, it will discuss the software design considerations faced in establishing a software model that lends itself to collaborative, distributed software development and list some of the problems and issues that arise in attempting to implement and maintain that model. It should be noted that this discussion is presented in the context of a scientific research environment. Thus, some points are not relevant or valid for commercial software development. It has been our experience that neuroimaging scientists, both out of necessity and interest, want to be active participants in the software development process, and we are interested in methodologies that will foster that engagement.

fMRI brain studies collect very large quantities of binary image data (multi-subject studies can easily require 20GB or more) that undergo a number of processing stages, including noise reduction, signal processing, image processing, statistical analysis, and rendering and display computations. Because of the large, unwieldy nature of the data collected, it cannot be analyzed with most commercial image processing or statistical packages. Therefore, researchers have of necessity developed their own software (Gold, 1998) to efficiently implement algorithms specific to fMRI analysis, and to handle the specific formats of the image data, as well as the various pieces of semantic information (e.g., subject ids, statistical parameters, and links to anatomical data) that accrue at each processing stage. Much of this software is Unix command-line programs written in C or C++, with Motif, Tcl/Tk, or Java graphical user interface (GUI) support (Laird, 1998), or packages built in a programming environment such as MATLAB (The MathWorks, Inc.) or IDL (Research Systems, Inc.); there are also a number of Windows and Macintosh programs. Some packages focus on one particular problem domain (e.g., image registration), others offer an integrated solution for complete dataset analysis. A listing that includes information about most of these programs is available at the Image Analysis Tools Registry maintained at www.cma.mgh.harvard.edu/tools. The algorithms implemented in this software are of very high quality and of interest to a wide group of researchers. However, incompatibilities in user interface, data format, and computing environment may preclude some researchers from using some tools, make some analysis methods far more time-consuming and intricate than necessary, and result in redundant and less robust tool development. Depending on the complexity of a package, it may take a
researcher weeks or even months to gain expertise in using its core functions. Part of this learning curve is owing to the inherent complexity of fMRI analysis. However, a significant portion is also owing to the complexities of package usage and format requirements, as well as the large number of distinct steps required for a complete analysis. A researcher typically does not use these tools on a daily basis, but rather needs to access dozens or hundreds of tools periodically over the course of a year, and may need to delegate processing tasks to research assistants inexperienced in scientific computation. Fiswidgets attempts to provide a platform that provides enough integration and commonality across packages to significantly reduce the logistical overhead of using packages from a number of different laboratories, and that does not impose any constraints on the rapid, independent development of those packages.

**Fiswidgets Computing Environment**

**Overview**

Fiswidgets is an open architecture, Java-based software development project that integrates a number of fMRI analysis packages in a modular, extensible, graphical computing environment. This environment facilitates the interoperability and usability of software developed by independent laboratories. It allows the neuroscience researcher to dynamically construct, execute, monitor, and log multi-step and iterative data processing sequences made up of analysis programs from a number of different packages. The fiswidgets software was developed to address the following specific issues: 1) integration of modules from different software packages; 2) need to efficiently execute iterative and batch mode processing jobs; 3) difficulty in using Unix command-line programs; 4) data format incompatibilities.

The design strategy of the fiswidgets architecture is to invest programming effort in a computing infrastructure that is sophisticated from a computational perspective, with functionality such as Client/Server support, parallel and iterative execution and other flow control constructs, graphical interfaces, etc., so that new methods which may be quite sophisticated from a scientific and algorithmic perspective, but relatively simple from a data processing and control perspective, can be put immediately on-line in this infrastructure and access its additional functionality. Examples of these kinds of architectures include: 1) The AVS/5 and AVS/Express products from Advanced Visual Systems, Inc. that let the user assemble customizable processing and visualization modules in a visual programming environment for data analysis and exploration. Modules are supplied by AVS, and, there is a repository of over 700 user contributed modules. 2) SCIRun (Parker, 1997), a “computational workbench” for interactively steering and visualizing lengthy and complex simulations in computational medicine. Examples specific to neuroimaging include: 1) MEDx, a system developed by Sensor Systems Inc. (www.sensor.com), which incorporates modules from the SPM (Friston, 1995), FSL (Smith, 2001), and AIR (Woods, 1998) packages; 2) VoxBo (Kimberg, 2001), a general linear model-based fMRI analysis package designed to interoperate with 3rd party packages; 3) The LONI Pipeline (www.loni.ucla.edu/NCRR/Software/Pipeline.html), developed at the Laboratory of Neuro Imaging at the UCLA School of Medicine that permits researchers to construct processing sequences that access supercomputing resources in a dataflow driven processing model; 4) An example on a smaller scale is the plug-in feature of the AFNI (Cox, 1997) system, in which an API is provided for developers to write modules that can be easily incorporated into the existing graphical display. The most significant differences between these systems and the fiswidgets architecture are: 1)
Fiswidgets focuses on interfaces to command-line programs and batch mode processing, not sophisticated scientific visualization tools, which are already available in several fMRI packages; 2) The fiswidgets data processing model is simple; it is intended to permit researchers with little experience in scientific computing to easily build their own processing streams, not to use processing streams created by programmers; 3) The fiswidgets architecture was designed from the ground up as a decentralized environment that depends on 3rd party packages to supply the scientific components; there is, therefore, no inherent bias towards using components from a particular analysis methodology; 4) The fiswidgets architecture provides not only an infrastructure, but also a large set of components that have a consistent behavior via the fiswidgets interfaces, but can run outside of the infrastructure. These components are in themselves a resource. Developers can incorporate them in their own infrastructures or in modified versions of the fiswidgets infrastructure. And, users can run any component through its graphical interface as a single, standalone entity.

The fiswidgets architecture (Fig. 1) consists of four parts: 1) the set of Java graphical user interface (GUI) wrappers; 2) a Java toolkit used to develop the GUI wrappers and available to other programmers so that other labs can write their own GUI wrappers and plug them into the fiswidgets environment; 3) a desktop style graphical computing environment that incorporates the GUIs as subcomponents and permits them to be linked into batch style processing streams; 4) utility programs such as medical image format converters and data display tools. The current fiswidgets release (fiswidget-1.5, Nov. 2001) is available for download from http://neurocog.lrdc.pitt.edu/fiswidgets.

Subcomponents

The fiswidget desktop currently contains 100 analysis subcomponents including: almost all (31) of the AIR-3.08 (Woods, 1998) image registration applications; 22 AFNI (Cox, 1997) applications (3dDeconvolve multiple regression, 3dfim+ correlation analysis, 3dcalc image calculator, et al.); 10 tal (Frank, 1997) applications developed independently by the Brainvox/tal software team, University of Iowa, (tal_gwcsf image segmentation, tal_regress multiple regression, tal_stat descriptive statistics et al.); random effects ANOVA and ROI utilities from the NIS package (University of Pittsburgh, Princeton University); Fourier analysis and CovariateManager applications (University of Pittsburgh); modules for reconstruction of MRI images; and format conversion, graph plotting, table display, and image-display utilities. As a convenience for the user, and to help insure that proper credit is given to package

![Fig. 1. Overview of the fiswidgets architecture. Components consist of the underlying application plus a fiswidgets wrapper. The wrapper creates a GUI that is integrated as a subcomponent of the graphical computing environment. [CFO]](image-url)
authors, we have added a component to each package that lists the publications that can be cited for that package. Screenshots of the subcomponents are online at neurocog.lrdc.pitt.edu/fiswidgets/fisdocs/browse.html.

Each subcomponent consists of two parts: a Java graphical interface wrapper that lets the user specify all the command-line parameters to the application and run it, and the application itself, usually a C/C++ program or a shell script. This application is installed without modification as per the conventions of its particular package; if the package has already been installed on the system there is no need to recompile or change the installation in any way. In addition to providing the graphical interface, the wrapper augments the application with a layer of additional data management and display functionality. For example, many of the AIR-3.08 programs process only one image per invocation; the fiswidgets interface permits invocations over entire directories or lists of images. Likewise, some of the AFNI programs process one 4D dataset (3 spatial dimensions plus time) per invocation, whereas the researcher might prefer to process the data in independent subsets over the time dimension; the GUI provides a convenient interface for doing this. The GUIs also check for errors, such as file overwrites or illegal parameter specification, log the processing steps, and display program results in tabular or graphic format. All of this functionality is provided without changing the executable package and without the user needing to know if the work is being done by the wrapper or the underlying program, thus providing a more uniform “look and feel” across packages.

**Toolkit**

The fiswidgets wrapper toolkit contains a set of high-level graphical elements, thread control methods, and utilities that permit rapid development of Java GUIs that include a number of built-in features such as an HTML help system, logging of program output messages, and parameter save/load. It is a general purpose toolkit for writing UNIX command-line wrappers, and contains no elements that are specific to the neuroimaging applications for which it has been used. It is designed to be used by programmers who want to create fiswidget subcomponents, not by the end users of the applications. The use of a toolkit, instead of programming the wrappers directly in the Java language, provides a layer of protection against changes in the Java language, because only the fiswidgets toolkit needs to be updated to accommodate changes, not each fiswidget wrapper. Also, it provides an efficient way for all fiswidget subcomponents to have a similar look and feel, without explicit collaboration and consultation among different programmers using the toolkit. The toolkit has been tested and run extensively on Silicon Graphics/IRIX, Intel/RedHat Linux, and Sun Microsystems/Solaris systems, and is being ported to the Macintosh/OS-X and Windows (with Cygwin Unix emulation) systems.

The graphical elements of the toolkit are built on top of the standard Java AWT (Flanagan, 1997) and Swing classes and contain high level widgets such as integer and float fields that accept default values and labels and prohibit non-numeric input, directory browsers that permit multiple file selection from different directories, dialog boxes, radio buttons, checkboxes, etc. Developers are not limited to using the elements in the fiswidgets toolkit; they may add any graphical element they construct from the AWT or Swing classes to a fiswidget graphical interface via the fiswidgets “generic” component class.

The thread control methods enforce a clean separation of display and control functionality in fiswidget wrappers. Each wrapper consists of two Java classes, one to display the GUI on the computer screen and respond to
user input, and one to manage the execution of the command-line program. From a flow control perspective, the only function of the display class is to collect the command-line parameters from the user and pass them on to the execution class. Because all of the command-line information is encapsulated in one class, it will be possible to extend the system to use alternative classes to collect and transfer this information to the execution class (e.g., graphical interfaces written in other languages, or non-graphical interfaces to formatted text files). After collecting the parameters from the display class, the execution class performs as many sanity checks on the parameters as possible, and initialization steps such as creating directories, uncompressing files, etc. Then, the appropriate command-line string is built and the application program is invoked through the Java Process class. The Process class spawns the job as a regular UNIX process outside of the Java Virtual Machine (JVM) environment. The program output that would have been displayed in the xterm or other UNIX window had the program been invoked directly from the command-line is captured and displayed in a scrollable output window and also appended to a log file. The exit status of the program is returned by the Process class, and any errors are displayed to the user.

The toolkit provides a number of utility functions that are either automatically built into each fiswidget graphical interface or are methods available to the programmer for use in coding the execution class. The built in functions include 1) a logging facility to record datesamps, the exact command-lines that
Fiswidgets were invoked, and the program output; 2) a help menu item that invokes a browser to display HTML documentation files; 3) a user editable configuration file (similar to a Unix login file) to specify and customize various system parameters (For example, this file contains the full Unix pathname pointing to a bin directory for each package installed in the fiswidgets environment. This permits users to easily switch between different package versions without updating the wrappers.); 4) a save/load button that permits the current parameter settings to be saved to disk and reloaded. These are saved in a binary file using the Java serialization class, a general method for saving any Java object. However, this format prohibits the user from inspecting or editing the saved parameter settings directly, so, the save/load mechanism has been re-implemented to use Extended Markup Language (XML) (Hoque, 2000) format. This feature will be available in the next release of fiswidgets (June 2002) and provides additional functionality to annotate and display parameter sets. The programmer utilities include methods that provide UNIX-like functionality, such as querying available disk space on a partition, decompressing files, sleeping, generating temporary filenames, Java specific utilities such as making “endian correct” read operations, and pulling resources out of jar files, and miscellaneous tools such as sorting and string manipulation operations.

Desktop

The fiswidget desktop (Fig. 2) is a graphical framework that permits fiswidget subcomponents to be dynamically linked together to create reusable, multi-step processing streams. The user selects subcomponents from drop-down menus for each package, orders them in a graphical flow display with drag-and-drop rearrangement, edits parameter settings by opening each component in the desktop panel area, and starts the job via one run button. The applications are invoked in serial; the completion of one component triggers the start of the next. There are currently two flow control constructs to modify the serial execution flow. First, there is a “for-each” widget (Fig. 3) that permits the user to execute portions of the flow multiple times, changing selected parameter settings automatically at each iteration, much like a programming language “for-each” loop. This can be used to repeat the same analysis over multiple subjects. The user sets variable names and associated lists of parameter settings in a spreadsheet, and enters the variable names instead of the parameter settings in the subcomponents’ GUI fields; the desktop makes the appropriate string substitutions before each iteration. Second, there is a breakpoint facility so that the user can halt the flow and inspect intermediate results before continuing with the remaining steps. Additional flow control options will be added to the desktop, such as support for high level parallel operations and scheduling jobs to run at specified times. Entire desktop flows can be saved and later reloaded in the desktop using the same save/load mechanism described previously for individual components, thereby permitting not just individual application settings, but also entire processing sequences to be saved and reused. A desktop “find and replace” mechanism for global string substitutions assists the user in changing parameter settings for re-use of processing streams.

Over the next year, Java RMI-based Client/Server capability (Orfali, 1997; Lewandowski, 1998) will be added to the fiswidget desktop specifically to separate the execution of the graphical user interface wrappers from the execution of the C/C++ application programs, so that users can run the graphical interface on their local machine (e.g., PC or Macintosh) and invoke jobs on
Fig. 3. Example of the fiswidgets desktop “for-each” loop and the command-line invocations logged to the output files for the first loop iteration. The AirAlignlinear GUI generates both the alignlinear and reslice command lines; FormatConvert and Afni3dDeconvolve generate the fconvert and 3dDeconvolve commands. [CFO]
remote Unix servers. This will improve performance by taking the CPU and memory load of running the graphical interfaces off the server machine, and, give users faster interactive response by running the interface directly off their local machine. It will also provide unified access to multiple platforms, so that, for example, some processing steps can be performed on vendor specific machines or high performance servers. In this design, both the data and the data processing programs reside on the server side. We do not plan to implement a model in which the client routinely transfers imaging data to a server, and receives back processed data. Given the magnitude of imaging datasets, this would be a highly inefficient processing strategy for most computations. Tools for such bulk data transfers as are necessary (e.g., initial download of data, copying to an archival machine) will be provided via a subcomponent that runs the ftp protocol.

The desktop is one of several distinct computing environments in the fiswidgets architecture that aggregates individual fiswidget components into graphical displays that give the user unified access to multiple packages. It is the most complex environment, specialized for batch mode processing. The architecture also includes a simpler “wizard” framework that permits a set of applications to be linked together so that the user can execute them in a step-by-step, “first-next-last” fashion. And, toolchest and workbench frameworks group together subcomponents by package or by application type (e.g., pre-processing applications or ANOVA based tools) in button panel displays so that individual applications can be easily accessed for stand-alone use. Any subcomponents created with the fiswidgets tool kit can be incorporated into these environments without modification or recompilation. The object structuring of the Java programming language guarantees that once fiswidget wrappers have been compiled successfully, they will interoperate successfully in the larger GUI frameworks.

Utilities

The fourth part of the fiswidgets architecture provides a collection of general purpose utilities, not specific to any analysis package. The most important of these is a data format conversion program that currently supports Analyze, AFNI, GE image (read but not written), and one locally used format. It is a C++ program with a fiswidget interface. We plan to add support for MINC, raw (user specifiable), and some BrainVoyager (Goebel, 1997) formats in the near future. The user needs to insert this utility at the appropriate places in the batch processing jobs when using subcomponents with incompatible formats. A medical image viewing utility, ImageJ (rsb.info.nih.gov/ij), developed by the Research Services Branch of the National Institute of Mental Health is also invokable in the fiswidgets environment. It is useful for quickly inspecting results (rather than viewing them in a full orthographic viewer), and since it is written in Java, it can be installed without recompilation or porting to any machine that runs the fiswidgets architecture. In addition to these, there are utilities for text editing, spreadsheet viewing, graph plotting, and sending e-mail. These are very simple tools, designed to help people unfamiliar with the UNIX system, so that they do not need to transfer data back to their PCs to accomplish routine editing and display tasks.

Discussion

Design Criteria

The fiswidgets architecture is designed to integrate a number of software packages that are written by different laboratories with minimal collaboration, and, that are continually under revision and update. The following design criteria are key in reaching that goal: 1) Open architecture and open source; 2) Modularity and extensibility; 3) Software wrappers and powerful command-line inter-
faces; 4) Embedded commercial off the shelf (COTS) products.

Open Architecture and Open Source

An open architecture is defined as an architecture whose specifications are public, either through full release of the source code as in the Linux system (Raymond, 1999) or through the use of publicly specified protocols, such as the Common Object Request Broker Architecture (CORBA) (Orfali, 1997) and Model Driven Architecture (MDA) interoperability specifications developed by the Object Management Group (www.omg.org). Public specifications give third party developers full access to all technical information needed to enable them to create add-on products and enhancements that interoperate successfully with the original product. The emerging definition of open source code is much broader than simply permitting third party developers to read source code; rather, it permits any (commercial or research) developer full, free access to the code, with rights to alter and redistribute the source and executables. (See www.opensource.org for a full definition of “open-source,” written by The Open Source Initiative, a non-profit organization that has been a leader in managing and promoting open source development.) Most of the fMRI analysis software developed by research laboratories is already available in source code so that the users can see exactly how an algorithm is implemented. But, in a step that is not yet widespread, it would also be helpful if the code were available to developers for modification, repackaging and redistribution. This would reduce redundant code development, since programmers could leverage existing code when only a small modification is needed to a program, or a problem has already been solved in a subroutine that could simply be reused if permitted. It would also enable developers to provide pre-compiled, executable versions of programs from multiple sources as one single, easy-to-install distribution.

Modularity and Extensibility

A software system can be open architecture and open source, such that any other developer could in principle modify, reuse and extend it, but, unless the architecture is designed to be extensible, such modifications might be so cumbersome as to be unfeasible. For example, if an update entails re-testing an extensive amount of code, or if module dependencies are not well documented, the resulting system fragility discourages both integration efforts as well as the rapid release of new methods demanded in a research environment. The use of object-oriented technology (Schach, 1999) and well-documented Applications Programming Interfaces (APIs) can help to promote modular design. We chose to implement the fiswidgets architecture in the Java (Gosling, 1996) programming language, because it is a very robust, object-oriented language, and has extensive support for graphical interfaces and network communications (e.g., remote method invocation, and sockets).

Software Wrappers

A software wrapper is a specialized kind of application that does not compute a specific result, but provides a new interface between a program and the user, or between two programs (Venema, 1992; Fraser, 1999). The key feature of wrapper technology is that it can add essential functionality to a piece of software, without altering that software. We have been able to use wrappers successfully in the fiswidgets architecture because the packages we selected already had strong UNIX command-line interfaces in which all parameters are passed to an application at start time as a command-line string. The wrappers present a convenient interface to the user, construct and run the command-line string, and augment the application function with additional data.
management and display features. Wrapper technology also works well with programs that have a scriptable interface, in which case the wrapper generates the script instead of the command-line. It is difficult to wrap programs that pass parameter settings through user configurable input files, that present only a graphical interface to the user, or that have only a highly interactive (e.g., menu driven) interface that requires manual input at each processing stage. While such interactive systems are quite useful in coaching a new user through a series of steps, or permitting an experienced user to modify each processing step based on the results of the previous step, they are not efficient for rapidly processing large quantities of data, and are not amenable to integration in larger systems.

Commercial Off-the-Shelf Software

Commercial off-the-shelf (COTS) software, also known as “shrink-ware”, is software available for sale to the general public, that a user purchases and uses without modification. Software developers who use COTS components purchase the COTS software as well as a license to redistribute the component as an embedded part of their own product. The high cost of software development and the increasing use of the PC platform, for which a wide range of software is available, have driven the increased use of COTS products (Clapp, 2001). While COTS products are not free, given even modest economies of scale, they are significantly cheaper than designing, developing, testing, and documenting comparable products from scratch; and they can often provide more extensive documentation, support, and interoperability than custom made, “in-house” components, as well as the assurance of using known and trusted implementations of algorithms. They can play a very useful role in supporting fMRI analysis programs by supplying pieces of functionality that are not specific to neuroimaging analysis, and that are therefore of little interest to the scientists working on algorithm development, but are of great benefit to the users. These can include data display and management tools such as spreadsheets, databases, or graph plotting functions, and libraries for standard mathematical or statistical functions. We have purchased a commercial embedded spreadsheet application to replace the current simple fiswidgets spreadsheet and plan to release it by Fall 2002.

Design Issues

We have outlined above a number of design considerations that are key in developing an effective model for loosely collaborative software development. They have worked well for us in the fiswidgets architecture. However, we selected third party packages to work with that were stable, extensively field tested, and very well written. And, with the exception of the tal package, all the wrappers were developed by one group of programmers. Our goal was to seed the fiswidgets architecture with a critical mass of initial subcomponents. The architecture cannot continue to expand unless other developers contribute not only back-end subcomponents, but also fiswidget wrappers for those components, and are committed to supporting the widespread use of their software. However, the implementation and maintenance of this kind of software architecture does incur significant costs and raise a number of issues that may prove problematic for either the expansion of fiswidgets or the development of other similar loosely integrative architectures.

Development Overhead Costs

First, it takes more time to develop software components robust enough for general redistribution than to write programs to meet immediate local needs. Specifically, addition-
al time is needed to generalize a component (e.g., to handle more data types, work on multiple platforms, include more features, and report all errors), and to document it so that others may use it (Tracz, 1988, 1994) and, as the number of supported platforms and software versions increases, the length and complexity of the software testing process also increases. When a component is successfully generalized and used by a wider community, the number of questions and change requests will grow, thus increasing the amount of time a developer needs to devote to user support. If a repository of components is available, via an open-source license, development and maintenance costs can be reduced through software reuse. But, developers will also need to expend time to become familiar enough with the components to know whether an existing component is a good fit to the problem they are working on. We feel that these overhead costs are a worthwhile investment if they do in fact result in software that can be more easily used by more researchers, and also reused by other developers. However, not all developers can afford this investment, particularly in the beginning stages of software development.

Packaging and Distribution Logistics

Second, when a set of software tools is only very loosely integrated, and not unified into one package, the work required to install all the necessary components can become a strong disincentive for using such a mélange of programs. This is particularly the case in the UNIX environment, which may be unfamiliar and intimidating to users with a PC background. If a package does not provide executable files, users inexperienced in computer programming will need to struggle through the compilation process. Some packages require that additional packages be installed (e.g., scripting or GUI support, compilers, document formatters, math libraries, and license managers). The installation procedures for different packages are often significantly different, and frequent recompiles are required to keep up with recent updates and patches. This is a cost factor that might not be readily apparent to individual developers who are concerned only with the complexity of their own package’s logistics. One way to mitigate this problem is for developers to release binary distributions for most commonly used platforms. Another way is for developers to coordinate in creating streamlined installation and update procedures, or to use standard software installation managers (e.g., Silicon Graphics swmgr, Sun Microsystems pkgadd, and Red Hat rpm).

Data Format Incompatibility

Third, incompatibility in data formats is a major impediment to systems integration and interoperability. An enormous amount of effort is consumed, both on the part of software developers who must write conversion routines and multiple data i/o libraries and maintain legacy codes, and users who must reformat their data for different programs and hardware platforms. Data formats affect not just access to information in disk files, but also in-memory data structures and the ease with which programs can transfer data to each other either through traditional interprocess communication methods (e.g., message passing paradigms or shared memory) or networked communication (e.g., remote procedure call or distributed object technology). The problem of incompatible data formats is acute for neuroimaging software; there are a number of major MRI hardware vendor proprietary formats in use, several commonly used image formats, and a number of formats specific to just one or two software packages. The cost, both to user and developer, of converting between formats, is such that often routines will be re-implemented, rather than adjusted to interoperate with existing routines.
that use a different format. Ultimately, the solution is to evolve towards using a smaller number of formats. But, this is a difficult and time-consuming process, and datasets in legacy formats will survive for a very long time to come. This issue is addressed in the fiswidgets architecture by a format conversion sub-component that the user must explicitly insert at the appropriate places in the fiswidget desktop batch processing flow. The alternative would be to implement automatic data conversion, either by pre-appending the appropriate conversion routines into the batch processing flows, or performing this step internally in the wrappers. We decided not to pursue this alternative, for several reasons, although it could potentially provide a significant usability enhancement for the user. The primary reason is that the amount of work this would entail, given the number of packages, formats, and applications in fiswidgets, the complexity of the meta-data formats, and no guarantee that a particular package will not change its formats, does not warrant working on a fully automated solution when sufficient functionality is offered by a user invokable routine. Second, implementing automated conversion would necessarily add a layer of complexity that would be a barrier to quickly adding new components that use new formats or unanticipated variants of existing formats. We would like developers, both in our laboratories and in other laboratories, to be able to write subcomponents without needing to insure that they comply with a specific data format standard. Third, most of the command-line programs do not have an interface that allows them to read input data from memory, and since fiswidgets does not make any changes to those programs, a disk-based method of format conversion is needed. Given the large size of neuroimaging datasets, we felt it was best to leave management of the location and duration of intermediate datasets to the user. If at some point the community settles on a smaller number of formats, we plan to revisit both the issue of automatic conversion and in-memory transfer of intermediate data between subcomponents.

**Graphical User Interfaces vs Scripting**

Fourth, there are a number of costs entailed in using graphical user interfaces. The main alternative to graphical integration of heterogeneous (different languages and calling conventions) modules is the use of a scripting language (Ousterhout, 1998) such as Perl, Tcl, or one of the UNIX shell languages. Graphical interfaces are less powerful than a scripted approach in the sense that the user does not have access to complex flow constructs such as if-then-else, switch, or while statements that can be driven by a very wide range of test clauses (e.g., program exit status, file existence, and type checking). Also, the user cannot easily access many standard UNIX utilities such as cat, grep, dd, and diff, that are often used to prepare or check data. Second, running the GUIs adds an additional memory and CPU load to the system, and performance may be prohibitively poor across some networked connections to servers. Third, the GUI components and infrastructure present yet another new and unfamiliar piece of software that the user must master. The first of these drawbacks is the most serious, as performance issues apply primarily to server-based architectures and can be addressed by Client/Server protocols, and the training issue is a cost that is recouped if the GUIs are in fact robust and friendly enough to reduce the amount of lower level application and operating system details the user must learn. We are beginning to address the issue of more powerful flow control constructs in the fiswidgets desktop with the addition of “for-each” loops, and plan to extend this with additional constructs, (e.g., parallel operations). The desktop also has a generic component that lets the user insert any Unix command-line into the desk-
top flow. However, graphical scripting environments like these will, by design, remain less powerful than text-based scripting languages as the intent is to hide complexity from the user.

Acknowledgment of Software Authors
Fifth, the authors and/or implementers of an algorithm or piece of code need to be acknowledged and credited for their work. In an open source code environment, this is not enforced, and in an integrated system the source of the subcomponents may be obscured. While there is no way to control how open source code may be used in general, we feel that excellent acknowledgment of the authors of software components will be made within the neuroimaging community. It is a research environment in which originality is respected. Acknowledgment of the source of a program or algorithm is also an indication of its quality; in general developers who have incorporated components from other sources in their systems have been more than eager to assure people that it is the original, unchanged code, used with the author’s permission. One example of both scientists’ willingness to contribute their code to other developers, and those developers actively advertising the participation of those scientists, is the MEDx system mentioned above, which incorporates modules from the AIR, SPM, and FSL packages.

Conclusion
The fiswidgets architecture has been developed as a resource both for neuroscience researchers performing fMRI analysis computations, and for the software developers writing the analysis applications. The use of a graphical computing environment to loosely integrate a set of independently developed applications significantly enhances their usability and interoperability for the end user. The availability of component-based computational infrastructures and utilities that can be shared across applications permits programmers to achieve that usability and interoperability at reduced programming effort and with minimal collaboration overhead or constraint on individual applications. Within the neuroimaging community, there is both a desire and a need for independent laboratories from different disciplines to continue to make their analysis tools available for widespread use. There is also now a growing need for computational tools to support that distribution; the fiswidgets architecture is an implementation of one possible model to provide that support.

Acknowledgments
Fiswidgets development is funded jointly by the National Institute of Mental Health, the National Institute of Neurological Disorders and Stroke, and the National Cancer Institute, under the auspices of the Neuroinformatics/Human Brain Project, grant #RO1 MH62006, Usability and Interoperability of Neuroimaging Software.

References
Woods, R. P., Grafton, S. T., Holmes, C. J., Cherry,