CSImage Technical Summary

Michael Ochs[1] and Joseph Callahan[2]
[1]Bioinformatics and [2]NMR and Medical Spectroscopy
Fox Chase Cancer Center
Philadelphia, PA 19111
27 April 2000

Description

This file is provided for those curious about the internal design of *CSImage*. It is not meant to be a thorough technical document, but merely to provide some insight to the philosophy followed during the design. The thinking which went into the use of design patterns for implementing some of the features in *CSImage* was heavily influenced by Bruce Eckel's Design Patterns and Java seminar (http://www.bruceeckel.com/).

Design

CSImage is designed to be an easy-to-use, powerful system for manipulating spectra and images. Previous similar analysis systems have generally been limited to specific data types and operations, which makes their usefulness in research limited, since researchers often wish to explore new methods of data analysis. *CSImage* has been designed using object oriented principles and design patterns to avoid these limitations.

CSImage incorporates object oriented principles throughout its design. The data objects use an inheritance hierarchy, with an abstract class *csiObject* at the top, so that all method invocations can be handled by polymorphism and run time type identification. Instance variables within the objects are not accessible from the outside, so that modification of variables occurs through well-defined (and easily traced) interfaces. User variables are incorporated into the data classes through composition by the inclusion of an instance, *localVar*, of the *UserVar* class in a *csiObject* object. In this way all data objects can be modified to include variables of interest to the user.

In addition to adding new data variables, users can create new operations. In order to make this as easy as possible, a design pattern known as *Visitor* is used to tie all major operations to the data. In the visitor pattern, all operations are called through a method in the data objects (which is present in *csiObject*) named *AcceptVisitor*. *AcceptVisitor* takes a *csiVisitor* object as its argument. The class *csiVisitor* is an abstract class which contains a series of methods *Apply*(*csiObject*), one for each data class. These methods are overridden in classes which inherit from *csiVisitor* and which implement *Apply* for the specific data class. If a method is not overridden, *csiVisitor* provides a simple dialog informing the user that the method is not implemented for that data type. Again, polymorphism and run time type identification provide the mapping of operations to data types.

Perhaps the most innovative feature in *CSImage* is the use of design patterns and Java to solve the problem of reading external data files with unknown formats. In the scientific community computers are not nearly as standardized as in the business community. A number of

"legacy" systems running spectrometers or medical imagers use nonstandard file formats, sometimes with nonstandard byte orders or float encoding systems. It is impossible to handle all data formats and all encodings within a typical system, because no single developer knows all the formats. However, Java's reflection mechanism coupled to the *Strategy* design pattern allows a novel solution to this problem. For reading external files a format descriptor file (fdf) is used to specify which class to read the data into, what byte order to use, what float encoding to use, which methods to call, and where in the external file to find the data. The Java reflection mechanism is used to instantiate the necessary classes and to retrieve the necessary methods from these classes. The *Strategy* design pattern is used to provide multiple methods of reading bytes or handling float encodings. In addition, custom classes are allowed so that a user can provide a new float decoding routine or byte ordering routine. Finally, although six specific orderings for real and imaginary values are provided, custom classes are also allowed for handling the ordering, since people are always more creative than a single system designer (or a roomful) can anticipate.

Implementation

In order to maximize the usefulness of *CSImage*, the application is written in the Java2 language as defined by Sun Microsystems. This language is portable across almost all presently used computing platforms. In addition, the program uses the Swing GUI components from Sun's Java Foundation Classes (JFC), so that a standard look and feel is present. The JFC however does allow a pluggable look and feel (PLAF), so that users may configure the GUI to appear and respond as a Motif, Windows, Macintosh, or proprietary interface rather than in the JFC default manner.

The overall structure of *CSImage* involves several "packages" or functionally connected classes. The *gui* is one such package. Other packages include *math*, for the complex variables as well as the mathematical operations, *io*, for reading and writing data to permanent storage, *data*, for encapsulating the data objects as described in the preceding section, and *visitor*, for maintaining methods which modify the data. In addition, there are classes designed for internationalization held in the package *string*. The *string* package holds classes with a series of static Strings giving all displayed text, including button labels and menu items as well as informative text fields for dialogs and the operational history. By replacing these strings with user defined values, the appearance of *CSImage* can be changed to the user's own language.

The *gui* package includes a number of low level components in addition to the main display components. These components provide items such as sliders with editable endpoints, canvases for displaying images, and complex dialog boxes. Each component is designed as a Java Bean to allow visual creation and maintenance of the GUI. The GUI includes a number of display windows and menus. The main window includes a *JPanel* which holds canvases used for drawing images, while secondary windows provide a history of operations applied to the data and a list of files presently within the project. In addition, when a large Fourier transform is performed (as defined by the user), a window with a progress indicator appears to allow the user to follow the progress of the Fourier transform. Transforms are performed in separate threads so that the user may continue work on other data if he desires to do so while a long Fourier transform is running.

The *data* package includes not only the csiObject class, but also the data classes which inherit from it: *Spec*, for spectra, *Image2D*, for two dimensional images, and *csiData* and its children, for multidimensional spectral data sets. Third parties can extend these classes through the *UserVar* class. This class is instantiated as *localVar* in any data object, so that it is always present. Variables together with *set* and *get* methods for them can be added in *UserVar*, however complex manipulations of the data should be handled using visitors. Since the *csiObject* class includes the method *AcceptVisitor*(), which essentially returns the present object to the Visitor with the *Apply* method invoked, addition of new data manipulation routines is quite easy.

The *visitor* package includes the *csiVisitor* class together with classes inheriting from it, which define the methods for manipulating data: *FilterVisitor*, for applying filters to the data (implemented in the k space domain), *PhaseVisitor*, for adjusting the phase in the data, *ShiftVisitor*, for shifting the data, and *FFTVisitor*, for performing fast Fourier transforms. These routines generally take the required portion of the data object (usually the array and some variables defining time or space) and call static mathematical methods contained in the *math* package. The data object is then updated as appropriate. The *csiVisitor* class contains default methods for all known data objects, which leave the object unmodified and inform the user that the operation is undefined for that data type through a dialog box. These methods are overwritten in the specific visitor objects. This permits a user to create a new visitor for a single data class and have the existing program handle mistaken invocation of this visitor on an improper data class.

Some additional points on the addition of new data types and operations need to be made. Because *CSImage* is designed to allow modification of all displayed text, a String describing the new operation needs to be entered into the *OperString* class. By using the Java reflection mechanism, *CSImage* can then bind the visitor to the menu item.

The *math* package contains all mathematical operations that can be performed in a static manner on the arrays in the data objects together with a class *Cpt*, which defines a complex variable and the standard mathematical operations for it. Although the class names are in most cases similar to those in the *visitor* package, they operate not on data objects but on arrays and variables as they are purely mathematical functions.

The *io* package contains methods that read and write data to a block device, such as a hard drive, or send data to a printer. These routines include routines to read data in a variety of formats. The *io* routines use a format description file (fdf) to specify the location of the variables of interest in the unformatted file, the size of the variables in bytes, the byte order used for the data (big or little endian), and the floating point encoding method, IEEE or VAX. Additional methods for handling byte order and float encoding are supported by use of the *Strategy* design pattern with classes *UserInt* and *UserFloat*, which allow substitution of the conversion algorithms. The standard format for output is the *CSImage* format that uses Java routines for reading and writing, which guarantees portability. However, other formats are supported in the initial version including the old Felix format and ASCII. Additional formats can again be generated by the user by creating additional visitors.

Finally the *string* package contains all the Strings for display of menus, buttons, dialogs, *etc*. Strings for new menu items created by addition of new visitor classes are inserted into the *OperString* class and appear on the *User* menu.

Goals

It is hoped that *CSImage* will provide a framework for future development of data analysis and display programs. Far too often in software development, the easiest path is taken without an overall picture of what the final design should be. This occurs because developers take a spiral approach to design, adding new features as an application grows without a clear strategy for doing so.

Object oriented design is one approach that attempts to limit the errors that naturally occur as a project grows over time. Through encapsulation and the use of well-defined interfaces, error propagation is contained. Furthermore, encapsulation allows the swapping of one component for another with the same interface, which permits constant improvements to subsets of a program (much as an FFT replaces a DFT to get faster computation). *CSImage* is an attempt to use object oriented design from the beginning to provide an extensible and modifiable system for data visualization and manipulation.

CSImage is not meant to be a final standalone product. It is intended to provide users with a GUI designed to handle most of their visualization needs together with an underlying framework for data manipulation. This underlying framework has been designed from the beginning to allow easy extension so users can add functions they need. Science and software design are two fields where the future is always changing, hopefully *CSImage* is prepared to handle such change and to permit users to change it.