Astronomical Data Analysis Software and Systems IX
ASP Conference Series, Vol. 216, 2000
N. Manset, C. Veillet, and D. Crabtree, eds.

SPECVIEW: An Interactive Java Tool for Visualization and Analysis of Spectral Data

I. Busko

Space Telescope Science Institute, Baltimore, MD 21218

Abstract. This paper presents the general design concepts used in the development of *specview*, a Java 1–D spectrum visualization program capable of handling, among others, the data formats generated by Hubble Space Telescope instruments. In the design process, particular emphasis was given to the set of classes that define and handle spectral data, so they can be used not only for building the initial visualization tool, but could serve as well as a foundation stone for a more capable application enhanced with data analysis abilities.

1. Introduction

The *specview* Java tool was conceived as a testbed for evaluating the use of Object Oriented programming techniques and the Java language to solve astronomical data analysis problems. Its first inception was as a 1–D spectrum visualization tool. However simple the initial requirements were, considerable effort was put into designing a comprehensive set of classes to describe spectral data and its properties. The goal was to produce a set of classes that would support the future upgrade to a full blown data analysis application with minimal recoding.

Usual OO practice mandates that one should strive for the highest degree of encapsulation. That is, an object should not expose its contents to the public, but rather should expose an interface that describes the object's *capabilities*, including e.g. the ability to create a screen depiction of itself. Thus in *specview*'s case one should require that objects that store spectral data should be able to drawn themselves on screen. In other words, the simplest design would require graphics code that is buried inside these spectrum–holding objects.

On the other hand, another requirement of *specview* was that it should be able to handle spectral data from different instruments and wavelength regimes in a uniform way. That is, no matter where the data comes from, its screen depiction should be the same. Conversely, a single spectrum could have more than one simultaneous screen representation. This suggests that the graphics code should be independent of the particular details of the spectrum holding classes.

These two conflicting requirements led to a mixed design based on the Model–View–Controller (MVC) paradigm, where objects have screen depictions that are entirely independent from their other attributes. This however applies only to classes that require a common view for subclasses, such as spectrum–

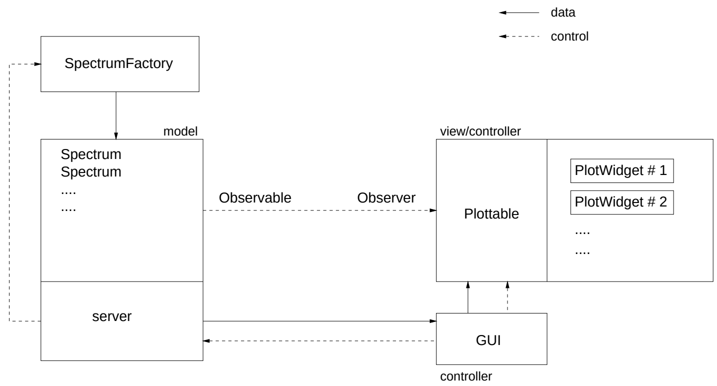

Figure 1. The MVC structure of *specview*. See text.

holding classes. Other types of objects in *specview* can have internal GUI code and be truly encapsulated.

Several concepts presented in the *Design Patterns* book (Gamma et al. 1994) were extensively exploited in *specview*'s design. Design patterns such as *Composite*, *Decorator*, *Command*, *Singleton*, *Observer* and *Abstract Factory* in fact make the backbone of *specview*'s design.

2. Main MVC Design

Figure 1 depicts the MVC relationships that exist in between the main elements of *specview*.

The Model consists basically of server methods that access a storage area where instances of a **Spectrum** class are kept. Upon receiving a command for creating a new spectrum, the server resorts to native methods in a factory class. These were presented in previous work (Busko 1999). **Spectrum** is an abstract class and appropriate subclasses are created by the factory depending on the file structure and/or instrument type. Except for the native code in the factory, and the subclass declaration files, the remaining code in *specview* can be made entirely unaware of the internal storage details in each **Spectrum** object, such as an echelle or multi–instrument structure.

The Controller and View are depicted close together in Figure 1, as in most MVC implementations the boundary between these two realms is quite fuzzy. The Controller object keeps the main GUI, but the plotting widgets (windows) have their own set of lower–level controlling elements (such as zoom controls). The plotting widgets are made generic and unaware of the existence of a **Spectrum** class by means of the **Plottable** interface. Any object that implements this interface can be rendered by the plotting widgets. **Plottable** attributes can be dynamically attached to **Spectrum** objects by the way of a *Decorator* pattern, thus making possible the rendering of any **Spectrum** object created by the factory. This design approach allows more code reuse since the plotting code in its entirety can be reused to depict other unrelated objects (such as tables), as long as they can be decorated with **Plottable** attributes.

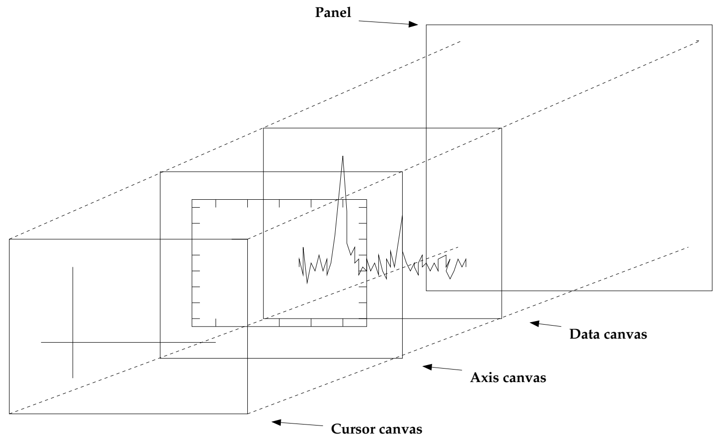

Figure 2. Building a plot using graphics elements. See text.

The View and Model are linked together by an *Observer* pattern. When the Controller creates a plotting widget associated with a given **Spectrum** instance, this instance is informed that it has that particular widget as an observer. From then on, whenever the **Spectrum** object changes its internal contents, the widget is automatically notified and can perform some action such as refreshing the plot. This mechanism ensures that the View is always in synch with the Model. It also allows more than one observer per observable, thus multiple views of the same spectrum can be kept in synch on screen.

3. Graphics

Graphics support is not directly related to the MVC architecture but is the single most important element of *specview*, which is basically a plotting application. The initial idea was to resort to existing scientific graphics packages written in Java. A quick survey showed that the few available were basically either literal translations of older, procedural code, or where developed in procedural style, making their adaptation to *specview*'s needs labor–intensive. Additionally, they did not exploit at all the new powerful resources available in Java's Swing and 2D packages.

Thus an effort was made to design and code from scratch a scientific plotting package that would at first satisfy the needs of the project, but then would provide capabilities to support other projects as well. The resulting work showed to be comparable, or even less, than the work that would be necessary to adapt any of the existing, older packages.

The graphics package design is based on two design patterns: *Decorator* and *Singleton*. It also uses *Observer* links for handling the cross–hair cursor.

Figure 2 represents the assembly process used to create a plot. Individual graphics elements such as the data plot, axis, labels, cursor, are plotted by separate objects whose classes implement a basic **GraphicCanvas** interface. The individual elements are stacked together using a dynamically–defined *Decorator* pattern. The pattern exploits the transparency and containment capability of

Swing components, as well as the internal mechanism used by Java's Abstract Window Toolkit to refresh widgets on screen. This results in very compact and modular code. Decorators can be combined at will for creating customized plots on the fly.

The cross–hair cursor is handled by a **GraphicsCursor** class implemented as a *Singleton*, that is, only one instance of the class can exist at any time. The **GraphicsCursor** object tracks the on–screen mouse pointer and uses *Observer* links to notify interested listeners of the current cursor position in world coordinates and of mouse/key events. Listeners can include the application code, which automatically gets notified of any cursor action and can listen to specific mouse/key presses. Special listeners are **CursorCanvas** objects, that use the notifications sent by the **GraphicsCursor** object to update the cross–hair position on screen. In this way multiple windows can handle copies of the cross–hair that move in synch in WCS space, helping the user to relate features displayed in separate windows.

4. Lessons Learned

The work done so far enables us to draw a few conclusions about the OO methodology:

- When dealing with complex, high–level objects, the MVC approach tends to save development time since the high–level plotting code has to be written only once. The savings are partially offset, however, by the time it takes to design a more complex interface to separate the Model from the View. Hopefully the current interface will stay in place and lead to further savings down the road when new unforeseen features are added to *specview*;
- The OO approach of encapsulating GUI code inside objects is appropriate when dealing with objects that need a customized screen depiction that may depend on such things as the particular subclass a object belongs to. In cases like this only the low–level graphics code can be shared among different classes;
- Time spent on the design phase is time well spent, and this fact is exacerbated in OO programming. Features that were carefully designed in well before the start of the coding phase ended up as remarkably short and easy to maintain code. Features that were tackled with the design–while–coding methodology tended to end up as larger and more convoluted code that demands more work to adapt to new requirements.

References

Busko, I. 1999, in ASP Conf. Ser., Vol. 172, Astronomical Data Analysis Software and Systems VIII, ed. David M. Mehringer, Raymond L. Plante, & Douglas A. Roberts (San Francisco: ASP), 417

Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1994, Design Patterns, Addison–Wesley Professional Computing Series