An Extensible, Interface-Based, Open Source GIS Paradigm: MapWindow 6.0 Developer Tools for the Microsoft Windows Platform

Harold A. Dunsford Jr.[1], Dr. Daniel P. Ames[2]

[1]Dept Geosciences, Idaho State University, Idaho, USA, hadunsford@gmail.com
[2]Dept Geosciences, Idaho State University, Idaho, USA, dpames@gmail.com

Abstract

In an open source GIS software development paradigm, the availability of a robust, extensible, programming architecture can be critical to project success. Indeed, though tiered-code software development is used throughout the industry (proprietary and open source) modularity, extensibility, and internal transparency are absolutely essential when writing code in a largely volunteer, spatially distributed, programmer community. This provides future and co-developers with a common platform that can be extended without fear of breaking other parts of the code – hence saving time and development costs, and improving the code base at a greater pace. In light of this, the MapWindow GIS project is undergoing a major restructuring to a more fully modular and extensible architecture in the context of interfaces for three-dimensional, spatio-temporal modeling. Specifically, we developed a set of base interfaces for all low level data access objects, visualization (2-D and 3-D), and geoprocessing tools; and developed a set of plug-in managers for each type of interface to handle loading new functionality directly from .NET assemblies (DLL's) at run time. The result is a set of core components that provide base GIS functionality in the .NET environment, but can be easily extended by third party developers through the implementation of the base interfaces. Nomenclature for the majority of the interfaces is derived from standards published by the Open Geospatial Consortium (OGC). From the perspective of multi-platform support, a platform independent core can be extended with platform specific modules (e.g. for Mono on Linux, or Windows Mobile).

1. Introduction

The purpose of the MapWindow project is to provide a developer toolkit for the Windows platform and a fully functional mapping application. (Ames et al 2008) Much of the success of the MapWindow project has been its easy-to-use plug-in programming environment allowing third party developers to quickly extend the application using simple languages such as VB.NET and C#. The current plug-in extensibility architecture in MapWindow is based on an "application-wide" design where each plug-in implements an interface that provides direct interaction with and access to all elements and components in the full MapWindow application. This provides a simple "single interface" approach that has proven quite usable, but has some critical deficiencies. Specifically,

developers who only need to use one or two components from the full system, should not be required to implement such a large and inclusive interface. More specifically, a plug-in that only provides a new data type, or one that only provides a new geoprocessing tool, should not necessarily require implementing an interface with hooks to the GIS map, legend, or toolbar. Indeed, every custom GIS application is generally different enough that no one plug-in interface would be suitable for every conceivable configuration of modular components. The architectural solutions introduced in this paper (and implemented in the source code of the MapWindow 6.0 project) address improving the extensibility model itself by creating a series of plug-in manager components (and a much more granular set of interfaces), each one handling a narrow focus of extended functionality.

We presume that the easier it is to extend a GIS platform, the more usable it will be for scientists and engineers who are not programmers. To cater to this need, proprietary software (e.g. ArcGIS) supports writing plug-ins that extend the capabilities of the software without having to re-compile the binaries or be expert programmers (Greenberg 2007). Many open-source GIS software projects also benefit from a plug-in environment because it offers a layer of insulation between the core application required by all users, and the newest, least stable code that is actively being developed. It is therefore useful to include the concept of plug-in extensibility into a modular framework that mirrors the modular component development itself. MapWindow GIS is only weakly copyleft, allowing the use of components in both open and closed source applications, and so designing the plug-in architecture itself to be modular gives greater flexibility to developers using the MapWindow developer tools (Greenberg and Fitchett 2001).

Component development presents the interesting challenge of trying to anticipate all ways people will use the software, extend it, or incorporate it into projects of their own. The solution is non-obvious because most accepted paradigms for extensibility make the assumption that a project will lie entirely within the open-source or proprietary domains. It is also challenging to develop an architecture that is equally suitable to open-source development (and hence maximum transparency) and proprietary derivative software (usually with restricted control). The open-source aspect of the project also means that the code enabling this extensibility can not be overly complex or future programmers will likely abandon it the first time it conflicts with changes to the project layout or design.

A new user of an open source GIS project might ask the question, “Why write a plug-in when you can simply download the source and modify any aspect of it to suite your needs?” However, plug-in extensibility can be beneficial for open-source projects by appealing to non-programmers that want to make small improvements, and by insulating the core application from unstable new additions. Even commercial software can benefit from a modular, re-usable plug-in architecture because the original company might be the authors of future derivative work based heavily on existing components.

The remainder of this paper presents a brief background of the MapWindow project, a summary of how plug-in architectures are used in other GIS platforms, and a brief overview of the plug-in approach currently being used in the newest MapWindow GIS 6.0 development effort.

2. Background

2.1 MapWindow

The MapWindow project started in 1998 at Utah Water Research Lab (UWRL) at Utah State University as an alternative to using ESRI MapObjects LT 1.0 for custom GIS applications. MapObjects LT did not support vector editing or raster data management – both key requirements of an important UWRL project. To meet these needs, UWRL researchers created the core MapWinGIS.ocx component, an ActiveX control that provides low level access to geospatial data which can be embedded in third party GIS software (Ames et al 2007). This ultimately led to the development of the extensible MapWindow GIS application to avoid the replication of common GIS software features (e.g. legend, preview map, etc.) This project moved to Idaho State University and was released as fully opens source software under the Mozilla Public License 1.1 (Mozilla 2008) in January 2004 as "MapWindow GIS 4.x".

The MapWindow 6.0 project inherits the strong foundation and legacy of several years of development, but is focused on modular components written strictly in C#. Since everything written for MapWindow 6.0 is in a managed, Dot-Net language, it will be far more portable in terms of exporting the project to the compact framework (used in Windows Mobile) or working with web applications (in ASP.NET). The MapWindow 6.0 version of the project began in the summer of 2007 as an effort to develop a topology toolkit for MapWindow 4.x. In the Fall of 2007, the team added the Net Topology Suite into the project. This required a thorough re-design of the underlying objects. New components were created from scratch to use the new objects, and the result was called MapWindow version 6.0.

2.2 Other GIS Extensibility Architectures

ArcGIS

The ESRI ArcGIS object model, when viewed as a diagram, sprawls out in a panoply of interconnected objects, each with a very precisely defined role. (Zeiler, M 2001) This is not open source software but demonstrates implementation of an extensibility architecture through various interfaces. In the Visual Basic for Applications (VBA) Macro development environment that is associated with ArcGIS versions 8.0 and later, directly accessing an object does not expose the majority of its properties or methods. An example is the MxDocument object. In order to access all of its properties and methods, one must first dimension a new IMxDocument interface, and point it at the object. After doing this, a host of new methods and properties appear that enable writing of extensions. The disadvantage of this approach is that working with all the extra interface declarations makes it more complicated for novice developers. Also, ArcGIS plug-ins can not be used without the proprietary ArcGIS software. One way to supply both the restricted behavior desired by proprietary plug-in developers, or the more extensive control demanded by those in the open-source environment is to use something called explicit interface implementation (MSDN 2008). As an example, the active ESRI Document object is responsible for returning a Map object through a Map property. Through explicit interface implementation, an IDocument interface could

have a different return type for the same property. Instead of returning a Map object, it could return an IMap interface. Therefore, the complex architecture can match the restricted plug-in architecture, and accessing members that are several levels deep becomes far simpler.

GRASS

Modularity is a major strength for GRASS. As is noted in *Open Source GIS A GRASS GIS Approach*, GRASS 6 is written in the ANSI C programming language and hosts more than 350 modules for management, processing, analysis and visualization of GIS data. (Neteler, M and Mitasova, H 2008) The GRASS development team is actively developing new graphical user interfaces, but the core project is built around a series of completely independent core libraries. The opportunity for open-source style extensibility in this setting is obvious. One simply needs to add a new library to the list of libraries and to extend GRASS. There is also a recognition that not every user will be an expert coder. To support intermediate level programmers, GRASS accepts script programming. UNIX Shell, PERL and Python scripts are supported, allowing repetitious tasks to be handled through scripting languages. GRASS demonstrates that a successful, open-source software project must have a framework that allows for future development and expansion over a long period of time. In our plug-in paradigm, the ever expanding collection of core libraries would exist as separate dynamic linked libraries, or dll files linked to the core application through the use of a common, interfaces.

Quantum GIS

Another open source desktop GIS with a well established plug-in architecture is Quantum GIS. The Quantum GIS plug-in architecture is similar to the MapWindow GIS approach in that new functionality is provided through compiled binaries that are referenced by the software at runtime. A powerful and useful concept in Quantum GIS is the idea of a “Data Provider” plug-in that allows developers to specifically extend the data formats that can be supported by the software. (QGIS Project 2008) This is a relatively new concept for the MapWindow developer team and will be a key part of the new MapWindow 6.0 design. The plug-in architecture for QGIS works by using python scripts or C++ files that satisfy certain criteria, i.e. hosting a particular script file with the name plugin.py and writing a script to match specific method names or schema. Our idea extends this basic concept by also introducing the componentized managers.

3. Interfaces

In software engineering, an interface specifies the public properties, methods, events and indexers that should be found on a class. Unlike an abstract class, an interface does not contain any code itself. One advantage gained by interfaces is interchangeability. Two very different classes can be passed into a single method that works equally well as long as both classes can answer the right questions. New classes can be created that work directly with pre-existing methods simply by having them implement the right interface. By using interfaces, the framework is extended to

function with classes that may not even exist yet. Using interfaces also does not prevent the use of more conventional inheritance in order to rapidly start from an existing class.

Our use of an ICoordinate interface creates a convenient example of interchangeability. The Coordinate and Point classes both implement the same interface. The interface simply specifies that there will be a double valued property for each ordinate: X, Y, Z and M. The Coordinate class basically provides the code that stores the double valued X, Y, Z and M values and not much else. A Point class, on the other hand, can answer questions about topology. For instance, it has an Intersects method to determine if the point falls within the area of a polygon. The Point class also inherits from a geometry base class. The C# language does not support multiple-inheritance, and so we might normally have to copy the values from a point to a new coordinate in order to use methods that require a coordinate parameter. By implementing the same ICoordinate interface in both the Point class and the Coordinate class, either class can be used for a method that requires an ICoordinate.

We can also imagine a scenario where we would have to work with time as well as space (Ott T and Swiaczny F 2001), (Egenhofer, MJ and Golledge RG 1998). With the new ICoordinate interface, new temporal features could simply implement the existing interfaces like ICoordinate. Unlike fields, properties can do more than simply storing a value. Properties work like accessor methods and can have very complicated behaviors. For instance, the implementation for the X value on a temporal feature could automatically check against a time value and return an appropriately interpolated X coordinate. The benefit is that the temporal feature can be passed directly into methods that were designed to take an ICoordinate. Using interfaces, therefore, provide an excellent tool to allow future classes to work with existing code.

3.1 Performance Characteristics

The price of using property accessors instead of public fields is that properties are slower. It can be imagined, therefore, that adding an Interface might make this step even more indirect, and therefore even slower. We created a simple test to evaluate the performance impact using several different scenarios for building a Coordinate class or structure. The test performed 1,000,000 vector additions using the various coordinate classes. The structure with fields (instead of properties) had an average performance of around 42 ms and was at least three times faster than the other data structures. There was a steady, but very small performance penalty from the structure (with properties) which has a performance of 125 MS to the class with virtual properties being accessed through an interface, which was about 155 MS.

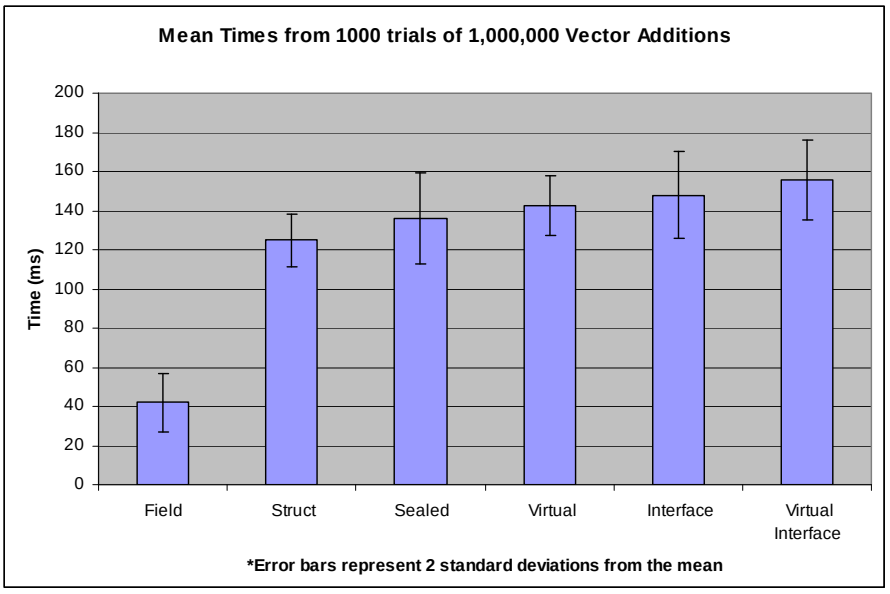

Figure 1. Data Structure Time Comparison

Though this test clearly shows that directly accessing fields is faster than using properties, it also shows that once the decision is made to use property accessors, the switch to using virtual or interface based properties is not significant. The architectural restrictions required when using fields instead of properties can also carry a hidden performance cost. Complicated coordinates, like the previously mentioned temporal case, would have to be written as a completely different class than the one being used during geometry calculations, and so all the ordinates would have to be copied from the new class to the geometry class and back, even if those coordinates were not going to be used in the calculation. The extra copying would negate the performance benefits apparently gained by using fields instead of properties. Since we are attempting to develop a framework that is centered on extensibility, we chose to use an ICoordinate interface with a Coordinate class that has virtual properties for the X, Y, Z and M values. This allows the widest range of strong-typed classes to be used directly in the existing topology methods.

4. MapWindow 6.0 Extensibility Architecture

4.1 Interfaces

Some key interfaces under development are as follows. Data Interfaces including IRaster and IFeatureSet as generic forms of raster and vector data. IFeatureSet is comprised of IFeature objects consisting of attribute data as well as geometric data. These data interfaces are not rendered directly, rather for drawing, an ILayer interface is introduced to bring together existing datasets with ISymbolizers. The symbolizer interface specified colors, drawing styles, and other visual characteristics. Layers are organized into a collection called an IMapFrame, which also specifies a visual extent (2D) or perspective (3D). Topology is controlled through a set of classes derived from the Java Topology Suite that satisfy the OGC simple feature architecture (Open GIS Consortium (OGC) 2006), (ISO19107:2003). Geometry classes including Envelopes, Points, LineStrings, Polygons and various Multi-Geometries have each been created, and both simple and complete interfaces have been developed for each of these. The IBasicGeometry interfaces convey all the

information needed to create a feature, but don't require topological methods. Implementing these basic interfaces from a new data format is trivial compared to attempting to re-implement the entire suite of topology methods.(Boissonnat, J-D and Yvinec M 1998) To make it easier to use topology methods when starting with a simple interface, we have also equipped each of the existing geometry classes with constructors that require the basic version of the same type.

4.2 Plug-in Providers and Managers

Some developers may wish to extend one aspect of the project themselves, but not want to support additional plug-ins. Other developers might want to open the options for data handling, supporting them through a plug-in like capability, while preventing any extensions that provide tools or custom drawing layers from being used. The model that we developed to support this wide range of abilities is called a manager component. Rather than having a single block of code in the MapWindow 6.0 application itself, a manager is a component and each component focuses on a specific type of interface extension. Each manager can then be configured at design time in order to specify the other components that it should connect with through its properties. The manager then knows enough about the application to substitute for passing some kind of pre-defined application interface to the various plug-ins. To use the manager, developers would first drag a manager component onto their application. Managers are components, but do not have a visible control, so they will appear at the bottom of the designer window, just like a DataAdapter or ImageList component. The properties of the manager component can be customized in the property window, enabling or disabling the various techniques for supporting plug-in extensibility.

Plug-in developers would focus on the provider, rather than the manager. Supporting a new data file format to support shapefiles, for instance, would require implementing the IFeatureSetProvider interface. This supplies the methods for opening and saving an IFeatureSet. The IFeatureSet can use the existing FeatureSet class, but alternately could be a completely new class that simply can answer the same questions.

5. Summary and Conclusions

In summary, this project has developed a plug-in extensibility architecture that is modular in order to be used side by side with modular components. It uses a simple component-like drag and drop behavior that should be very familiar to visual developers. The project makes this possible by using interfaces for each of its classes. The interfaces allow for interchangeability, can provide restricted access to components, and consist of smaller interfaces that can be used where implementing the entire class is not necessary. These interfaces then allow for the addition of run-time classes identified by using system-reflection. Because the new classes implement pre-defined interfaces, other components in the architecture can interact with the new classes as if they had a programmatic reference to them. Because the entire project has been built to use interfaces, future developers can exchange literally any part of it with their own classes as long as the classes implement the same interfaces.

The steps being taken by this project show an exciting new trend for community code. While mapping projects are abundant, projects that set themselves up to provide components for use by other projects are rarer, and projects that allow those components to be used for free by developers of proprietary software are extremely rare. This approach may prove to be naïve, as many advocates of the GNU general public license (GNU 2007) warn. However, producing easy-to-use components that are specifically designed for an environment used predominantly by proprietary developers would be largely useless unless those proprietary developers could use the software. Attempting to create tools that can be used for multiple purposes also presents new problems and perspectives that can lead to solutions that benefit the entire community.

References

Ames, D. et al 2007. Introducing the MapWindow GIS Project. *OSGeo Journal* 2: 8-10.

Ames, D. et al 2008. 'MapWindow GIS.' *Encyclopedia of GIS.* Sashi Shekhar and Hui Xiong (Editors). Springer, New York, pp. 633-634.

Boissonnat, J-D and Yvinec M 1998, *Algorithmic Geometry* Cambridge University Press, Cambridge, United Kingdom.

Egenhofer, MJ and Golledge RG 1998, *Spatial and Temporal Reasoning in Geographic Information Systems*, Oxford University Press, New York, New York

GNU 2007, GNU General Public License, Free software Foundation, viewed 16, August 2008 <http://www.gnu.org/licenses/#GPL>

Greenberg S and Fitchett C 2001, 'Phidgets: easy development of physical interfaces through physical widgets.' In: Proc ACM UIST, pp 209–218

Greenberg S 2007, 'Toolkits and Interface Creativity', *Multimedia Tools and Applications,* vol. 32, no. 2, pp. 139-159.

ISO19107:2003, *Geographic Information – Spatial Schema*, International Organization for Standardization, Geneva, Switzerland.

Mozilla 2008, *Mozilla Public License Version 1.1* viewed 16, August 2008, <http://www.mozilla.org/MPL/MPL-1.1.html>

MSDN 2008, 'Explicit Interface Implementation' *C# Programming Guide* viewed 16, August 2008, <http://msdn.microsoft.com/en-us/library/ms173157.aspx>

Neteler, M and Mitasova, H 2008, *OPEN SOURCE GIS A GRASS GIS Approach,* 3rd Edition, Springer, New York, New York.

Open GIS Consortium (OGC) 2006, 'Simple feature access – Part 1: Common architecture', *Implementation Specification for Geographic Information*, Version 1.2.0 viewed 16, August 2008, <http://www.opengeospatial.org/standards/sfa>

Ott T and Swiaczny F 2001, *Time-Integrative Geographic Information Systems* Springer-Verlag, Berlin.

QGIS Project 2008, *Quantum GIS User and Installation Guide,* version 0.9.0, viewed 16, August 2008, <http://gisalaska.com/qgis/doc/user_guide_en.pdf>

Rotaru C 1999, *Extensibility Model*, version 1.00, Mozilla.org, viewed 16, August, 2008, <http://www.mozilla.org/projects/intl/extensibility-model.html>

Zeiler, M 2001, 'Applications and Cartography', *ESRI ArcGIS 8.1 Object Model*, vol. 1, viewed 16, August 2008, <http://edndoc.esri.com/arcobjects/8.3/Diagrams/ArcGIS%20Object%20Model.pdf>