

# The GeoViz Toolkit: Using component-oriented coordination methods to aid geovisualization application construction

## Abstract

In this paper we present the GeoViz Toolkit, an open-source, internet-delivered program for geographic visualization and analysis that features a diverse set of software components which can be flexibly combined by users who do not have programming expertise. The design and architecture of the GeoViz Toolkit allows us to address three key research challenges in geovisualization: allowing end users to create their own applications on the fly, integrating geovisualization methods with spatial analysis methods, and making geovisualization applications sharable between users. Each of these tasks necessitates a robust yet flexible approach to inter-tool coordination. The coordination strategy we developed for the GeoViz Toolkit, called *Introspective Observer Coordination*, leverages and combines key advances in software engineering from the last decade: automatic introspection of objects, software design patterns, and reflective invocation of methods.

Keywords: Software design, coordination, geographic visualization

## Introduction

The GeoViz Toolkit (GVT) is a geovisualization environment that lets users easily create and modify custom palettes of coordinated exploratory and analytical tools. GVT users can build a custom geovisualization application by adding new views interactively, requiring no understanding of programming languages or technical details. Exploratory views such as scatter plots, choropleth maps, cartograms, and parallel coordinate plots, are augmented with spatial statistics tools like a spatial autocorrelation explorer, and cluster detection methods from Proclude (Conley *et al.* 2005) to help users develop and evaluate hypotheses. Behind the scenes, a sophisticated coordination architecture ensures that views can be added or taken away while retaining datasets, and visual characteristics like color and classification settings.

In the following sections we describe relevant prior work, the software architecture and design of the GVT with special attention to the *Introspective Observer Coordinator*, advantages that this coordination strategy enables, and an example real-world application using the GVT. We conclude with thoughts on our future plans for GVT development, dissemination, and evaluation.

Figure 1 about here.

**Figure 1:** The GeoViz Toolkit, showing voting data from the 2008 presidential election. Program available at <http://www.geovista.psu.edu/gvt>, source code available from <http://code.google.com/p/geoviz/>.

## Geovisualization Software Environments

Our interest in developing a new, user-friendly geovisualization environment is inspired by a wide array of previous work to develop geographic visualization and information visualization systems. Earlier examples of full-featured geovisualization application development environments include Descartes (Andrienko *et al.* 1999a) and its successor, CommonGIS (Andrienko *et al.* 2002), which were designed to automatically suggest tools and representations based on user tasks and available data, and GeoVISTA *Studio* which implements a visual programming interface that allows sophisticated users to assemble applications using a data-flow paradigm. The Improvise system (Weaver 2004) also provides highly-coordinated visualization facilities, as well as meta-visualization views. These environments offer multiple, coordinated views that feature dynamically-linked interaction behaviors that allow users to highlight, select, and brush spatio-temporal data. GeoVISTA *Studio* is an open source toolkit that provides the means to construct custom geovisualization applications using views and coordination mechanisms implemented as discrete JavaBeans (Gahegan *et al.* 2002). In *Studio*, users connect multiple JavaBeans together (for example, a choropleth map component, scatterplot component, classification component, and color scheme component) on a virtual canvas to construct an application. When connecting components, users have extensive options available to control how they coordinate data and representational variables. In practice, *Studio* works best for expert users who have a solid understanding of Java programming (Gahegan *et al.* 2008). The GVT is in many ways an evolutionary step beyond the GeoVISTA *Studio* application construction environment. The design philosophy for the GVT was to take the components developed for GeoVISTA *Studio* and wrap them in a coordination mechanism that allows non-expert users to simply select the views they desire and create a custom application.

The aforementioned geovisualization application construction environments draw from information visualization application construction environments like Snap (Adger 2000). Snap provides visualization tools as components that can be combined into custom applications quickly and easily by users with minimal understanding of programming principles. Other information visualization application environments like the InfoVis Toolkit (Fekete 2004), the Visualization Toolkit (VTK) (Azevedo *et al.* 2000), and prefuse (Heer *et al.* 2005) provide libraries of ready-to-use components that can be assembled and extended by software developers.

Additional motivation for our work on the GVT is to couple coordinated, interactive geovisual representations with advanced spatial analysis techniques. In previous research (our own and completed by others) we have learned that many of the users who might want to work with geovisualization software would like to see visually-led exploratory approaches augmented with quantitative spatial analysis methods to confirm or reject hypotheses they generate (Rinner 2007, Robinson 2007). In addition, we draw upon related software development efforts that have successfully coupled interactive geovisual representations with spatial analysis

methods like the Space-Time Analysis of Regional Systems (STARS) toolkit (Rey *et al.* 2006) and GeoDA (Andrienko *et al.* 2007).

### *Common Software Architectures for Geographic and Information Visualization Environments*

Information visualization and EDA strategies rely increasingly on use of two or more software applications to achieve desired functionality. Examples include the linking of ArcView® with XGobi (Cook *et al.* 1997), *Snap-together Visualization* (North *et al.* 2000), and *Orca* (Sutherland *et al.* 2000). To be fully effective, these systems need to support cross-application object selection with linking, and a consistent appearance, as well as linking of statistical attributes (Unwin 1999). This means, for example, that it should be possible to examine data in a map, a parallel coordinate plot, and a spreadsheet, while linking important visualization behaviors across program components, even when these components were developed independently.

*Linking and brushing* (Joining different views on a set of information by visually highlighting the same objects on different views) can be traced back to (FisherKeller *et al.* 1974) and (Newton 1978). Initial applications to geospatial data were proposed by (Carr *et al.* 1987) and (Monmonier 1989) and applied subsequently in many applications, e.g., (Cook *et al.* 1996, Dykes 1997, Haug *et al.* 1997, Andrienko *et al.* 1999b, Andrienko *et al.* 2007). These studies provide evidence that coordinated views for information visualization and query (focused on standard linking and brushing) are effective tools for access and analysis of complex information (Edsall *et al.* 2001).

Strategies for visual coordination across applications (as well as views) beyond linking and brushing have been implemented using a *pipeline* metaphor in several scientific visualization packages (e.g. AVS (Advanced Visual Systems)(Homer *et al.* 1994), IBM Data Explorer(Greg *et al.* 1995)). Figure 2 illustrates the pipeline graphic interfaces provided by AVS and Explorer. Coordination of program components by means of pipelines can be considered a form of visual programming, especially when the resulting program state can be stored independently as in the AVS system.

Figure 2 about here

### **Figure 2:** Interfaces for AVS (left) and IRIS Explorer (right)

The assumption underlying all visual-programming environments for visualization is that supporting interactive linkage of program components via visual programming will enable both the rapid creation of novel visualization approaches, and increase the accessibility of advanced geovisualization for non-programmers. Scientific visualization environments using the pipeline-based visual programming approach have been applied frequently to exploration of geospatial data (Trenish 1995, Wood *et al.* 1997, Gahegan 1998, Masters *et al.* 2000). Within scientific visualization, the focus of work on coordination has emphasized the development of multi-user environments for collaborative work (Brodlied *et al.* 1998, Watson 2001).

A related project, which extends the notion of object-orientation into “actor-oriented” programming, is Ptolemy (Liu *et al.* 2003). Ptolemy allows for visual programming, just as GeoVISTA *Studio* does. Similar to *Studio*, Ptomley uses reflection and supports a wide variety of computing paradigms. However, Ptolemy allows only limited user interaction. In addition, Ptolemy is directed at signal processing, rather than geographic data analysis.

## The “Introspective Observer” Coordination Design of the GeoViz Toolkit

This section describes our approach for extending the potential for coordination among visual and computational components in a geovisualization environment. As noted above, the approach builds on earlier work implemented in the Java-based visual programming environment of GeoVISTA *Studio* (Gahegan *et al.* 2002). We begin by describing the basic features of Introspective Observer Coordination. Next, we describe the coordination mechanism that supports application construction in GeoVISTA *Studio* (the `StudioCoordinator`) as a basis for comparison against our current work to develop an improved coordination mechanism for the GeoViz Toolkit. Next, we describe how the *Introspective Observer Coordinator* was implemented in the GeoViz Toolkit (the `GvtCoordinator`). This is followed by an examination of how these two types of coordinators perform the crucial task of inter-component registration. Finally, we describe the advantages (and potential disadvantages) of the `GvtCoordinator`., compared with the `StudioCoordinator`.

### *Introspective Observer Coordination*

Introspective Observer Coordination connects components using *introspection* to determine at runtime which interfaces and methods each component implements. This information is used to see if the objects could conform to a particular *design pattern* (the *Observer* pattern). Then, *reflective invocation* is used to connect the components appropriately. We explain these terms (introspection, design patterns, the Observer design pattern, and reflective invocation) below, since they may not be familiar to all readers, and since the concept of *Introspective Observer Coordination* is a natural outgrowth of their combination.

*Introspection* is the ability to discover information about objects at run-time. For example, we could ask of any class what its type is, and what the names and types of its properties and methods are. The ability to conduct introspection on all objects is a distinguishing feature of modern object-oriented languages like Java and C#. Introspection has been used in many contexts, such as grid computing (Badauel *et al.* 2007), automatic software testing (Simons 2007), and computer security (Kassab *et al.* 1998).

*Design patterns* describe commonly used combinations of objects, and form a common vocabulary of programming constructs in an object-oriented context. The most popular definition of design patterns is given in the seminal book by Gamma *et. al.*:

A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context (Gamma *et al.* 1995).

Perhaps surprisingly, there are few descriptions in academic literature of using introspection with design patterns (for one exception, see (Neumann *et al.* 2002)).

The *Observer design pattern* defines a one-to-many dependency between objects so that when one object changes state, many objects receive notification and are updated automatically. It relies on *Subject* (the one) and *Observer* (the many) interfaces and concrete classes. The exact relationship and communication mechanism specified by the Observer pattern is described later in this paper.

*Reflective invocation* is an extension of the introspection process described above. It means that automatically discovered fields, methods, and constructors can be programmatically invoked. In the case of the Introspective Observer Coordination architecture, it means that the methods conforming to the Observer pattern which were discovered during introspection are invoked to connect the components.

The Introspective Observer Coordination architecture we have designed and implemented for the GeoViz Toolkit helps support the following user-centered goals for analytical geovisualization:

1. End-users should be able to make their own applications
2. Geovisualization applications should integrate tightly with spatial analysis methods
3. Geovisualization applications should be easily accessible and sharable between users

### *The GeoViz Toolkit Component-Based Coordinator – GvtCoordinator*

The component-based coordinator we have designed for the GeoViz Toolkit coordinates arbitrary types of events between two or more components. It does so by requesting registration (and deregistration) of components based on their class information using *introspection*. In contrast to the object-oriented *Studio* coordinator (described below), the component-based coordinator (the main class for which is the `GvtCoordinator`) works by registering objects with each other. Events are then sent from object to object without the coordinator interceding in any way. The `GvtCoordinator` relies on introspection to discover which components should be coordinated and uses reflective invocation to automatically connect them. It relies on the *observer* software design pattern to know which components are listeners, and which are broadcasters. The Java programming language has the advantage (shared with Microsoft's language C#, which was derived from Java) over C++ and

Visual Basic that there is extensive and automatically derived class metadata available for any object. This enables meta-programming based on class metadata. For example, we can ask any Java object to report which class defines it. For any Java class, we can ask what methods, fields, and constructors that it has. In addition, Java classes can report the interfaces they implement. These automatic reporting mechanisms make powerful meta-programming approaches possible, such as the coordination explained here, the deployment of programs as web services, as well as incorporation of tools in many other programming contexts (Cazzola *et al.* 2002).

Using class metadata queries, whenever an object (in the example in Figure 3, `geoMapOne`) is registered to an instance of `GvtCoordinator`, the coordinator examines `geoMapOne`'s class, `GeoMap`, for methods that can add and remove references to interfaces, for example, the `SelectionListener`. Then, all previously added components are examined to see if they implement that interface. If any previously added components do implement this interface, then `Method.invoke()` is called using the appropriate object identity reference and arguments. In this case, the method is `addSelectionListener(SelectionListener l)` declared by the class `GeoMap`, using `geoMapOne` and the previously added component which implements `SelectionListener` as arguments. The presence of these methods indicates that these components are following the *observer* software design pattern. This registration process is illustrated in Figure 3. After this is done, any selections created and passed on by `geoMapOne` will be passed on to the listening object. Similarly, `GeoMap` will be queried to see if it implements any interfaces for which previously-added components have listeners, and `geoMapOne` will be registered with them if any are found. Therefore, events such as changes in selections or color choices will be received by `geoMapOne` with no user intervention.

### *The Studio Object-Oriented Coordinator – StudioCoordinator*

GeoVISTA *Studio* (Gahegan *et al.* 2002) allows fine-grained control and almost limitless flexibility in how components (implemented as discrete JavaBeans) can be connected, and therefore coordinated. This begs the question as to why there is the need for an improved coordinator for the GeoViz Toolkit. One reason is to reduce the burden on the application designer. If we have 15 components in a design, and each one can coordinate in 6 ways, then we will need  $15 * 6 * 2 = 180$  links for bi-directional association. Considering that in *Studio* one must use a GUI and make connection choices upon making each link, the process is tedious and time consuming. It is desirable to develop a coordinating component that does some of the wiring automatically. In general a coordinator simplifies an application design, even one with only a few components. Since the number of connections expands exponentially with the number of coordinated components, this advantage can be substantial for designs with many components.

The basic design of the object-oriented coordinator used in *Studio* is that of a multi-caster; coordination events are fired by objects being coordinated, intercepted by the coordinator, and then sent by the coordinator to all listening objects. A new component is simply added to a list of components that are being coordinated. One negative consequence of using one master list

for all coordination is that as this list grows, it needs to be traversed an increasing number of times as the number of components and types of coordinated events grows. For example, the sequence that occurs when a user initiates a coordinated interaction between two components, a `GeoMap` and a `Scatterplot`, is as follows. First, the two components are added to the coordinator. Next, the `Scatterplot` receives an interaction from the user. It passes this information to the coordinator, and the coordinator fires an event to the `GeoMap`. Note the “busy” nature of the coordinator during this sequence: all coordination events are sent by the coordinator.

Figure 3 about here

**Figure 3:** UML activity diagram comparing the processes of adding a component to the GeoViz Toolkit coordinator (`GvtCoordinator`) and the *Studio* coordinator(`StudioCoordinator`)

The mechanism outlined above is more complicated than the process implemented in *Studio* where each component is simply registered with the coordinator. There is a payoff for this slightly more difficult process of registration, however. When events are sent, they are sent directly to the receiving component, with no intercession by the coordinator. In fact, the coordinator could be removed after registration has occurred, and the components would function just the same. In a typical analysis session, registration of each component occurs only once, while hundreds or thousands of events may be coordinated between components. It makes sense to have the coordinator do a little extra work on set-up to save effort while the program is running.

#### *Comparing Registration Processes Between the StudioCoordinator and the GvtCoordinator*

Here we compare the sequence of how components are registered with the component-based `GvtCoordinator` to the object-oriented `StudioCoordinator`. For this example, we will assume a single component (the `GeoMap`) has been added to the coordinator. First, the user adds `GeoMap`. Next, the GVT coordinator queries the `GeoMap` to uncover what types of events the map is capable of sending and receiving. The information is gained by examining the `GeoMap` methods and identifying those that match the `EventSet` naming convention in Sun’s JavaBean standards (Microsystems 1997-2009). This identifies the types of interfaces that are implemented, which show the types of events the `GeoMap` may listen for, and also identified the types of events to broadcast, which are found by examining which types of event listener interfaces can be registered/deregistered with the `GeoMap`.

In contrast, the registration process is simpler when a component is registered with the object-oriented `StudioCoordinator`. The `GeoMap` is added by a user, and the coordinator adds the `GeoMap` to its list of coordinated events. This may sound like a component operation that takes place at runtime, rather than an object-oriented design time solution. However, a key aspect of the object-oriented `StudioCoordinator` is that all components that can potentially be registered must implement an interface (`StudioCoordinatorClientListener`) that the `StudioCoordinator` depends on. Therefore, the classes are linked at compile time.

The advantage of the component-based approach is shown in Figure 4 which shows the sequence of a selection change propagated from the `GeoMap` to the `Scatterplot`. The user initiates the change, and the `Scatterplot` receives it, because it was previously registered with the `GeoMap`.

Figure 4 about here

**Figure 4:** UML Sequence Diagram showing event firing during application use

Figure 4 also shows the more elaborate process that occurs when the same sequence happens using the object-oriented `StudioCoordinator`. The selection change is initiated by the user, and the `GeoMap` informs the `StudioCoordinator` that the selection has occurred. The `StudioCoordinator` then rebroadcasts the event to the `Scatterplot`.

We have placed a few restrictions on the component-based coordination process to improve its functionality. First, objects are not registered with themselves. Secondly, interfaces defined in Java *packages* (grouped classes in a common namespace) that primarily concern within-object communication, rather than between-object communication, are excluded. (*Packages* are groups of related classes such as `geovista.geoviz.map` or `javax.swing`.) The packages currently excluded for this reason are `java.awt.event` and `javax.swing.event`. If these packages are not excluded, then generic low-level mouse clicks and window resizing events, for example, would be duplicated across objects, which is (usually) undesirable behavior. Thirdly, the `GvtCoordinator` restricts itself to discovering public methods only. If this restriction is removed, then the `GvtCoordinator` violates Java's security rules for applets (web-delivered applications running inside other programs). With this restriction, `GvtCoordinator` can operate in applets with no security violation.

#### *Advantages of the Component-Based GvtCoordinator over the Object-Oriented StudioCoordinator*

The component-based `GvtCoordinator` has four software architectural advantages over the object-oriented `StudioCoordinator`, listed here in order of importance.

1. Stable Coordination
2. Stable Clients
3. Runtime Control
4. Granular Control

Each of these four advantages is explained with the aid of UML diagrams. Each advantage is explained, and the general relationship to component-based coordination techniques is identified. We also describe how these software architecture advantages enable visual-analytical advances in the GeoViz Toolkit.

#### *Stable Coordination*

The source code of the `GvtCoordinator` does not require modification to be extended to a new type of event. In the object-oriented design of `StudioCoordinator`, if a new type of coordination is desired, the source code for the coordinator needs to be changed, and the class recompiled and redistributed. The `GvtCoordinator`, by contrast, can work with future types of coordination that have not been thought of yet, and the `GvtCoordinator` can work with them without modification or even re-compilation of the coordinator, as long as the future coordination types are presented to the coordinator using the same code-naming patterns. The `GvtCoordinator` can do this via the automatic introspection mechanisms mentioned earlier. We can call this the advantage of *stable coordination*.

Figure 5 shows the object-oriented `StudioCoordinator` on the left, and the component-based `GvtCoordinator` on the right. In these class diagrams, all class members are shown.

Figure 5 about here

**Figure 5:** UML Class Diagram comparing the structure of `StudioCoordinator` to that of `GvtCoordinator` used in the GeoViz Toolkit

The `StudioCoordinator` has eight event types, and eight corresponding listener lists. If a non-programmer end-user wanted to add an additional event type to this list, she/he would be unable to. A programmer would be able to add an event type by modifying the source code, which makes application extension a non-trivial task. The `GvtCoordinator` has only two member variables, a list of firing components, and a list of listening ones. Any new types of events are automatically discovered and coordinated as long as there are sending and receiving components for that event type – no modification to source code is required.

### *Stable Clients*

The *stable client* advantage is that the coordinated objects do not need to be modified if events are modified or extended. In the object-oriented design, if a new type of event were used, clients would not be able to handle them without their source code being modified. Because events are passed directly from client to client using the component-based `GvtCoordinator`, the events themselves can be extended or otherwise modified without changing old clients or the coordinator. For example we could change the selection event to extend what a selection means from a single subset to multiple subsets with attached authors (i.e., to support multi-user interaction with a display). As another example, we could change the indication event, which is a transient selection of just one observation, to include all items in the class of the indicated observation. In both cases, neither the old clients nor the coordination manager would need to be modified. We call this the advantage of *stable clients*.

This advantage is a function of *stable coordination*, and stems from similar design considerations. Classes that are coordinated by the `StudioCoordinator` have a dependency on `StudioCoordinatorClientListener`, which specifies which events to listen for. Classes that are coordinated by `GvtCoordinator` also have dependencies, however these dependencies are not directly tied to the coordinator. The difference can be seen

in Figure 6. On the left, all the clients of `StudioCoordinator`, in addition to the coordinator itself, depend on the interface `StudioCoordinatorClientListener`. In order to add a new event type to those being coordinated this interface must be extended, and the source code of all clients changed. On the right, all three `GvtCoordinator` clients implement `SelectionListener`, while two implement `IndicationListener`. If we hypothesize that we would like our components to start coordinating conditioning, which enables filtering by a variable, then those clients would need to have their source code modified accordingly. However, the other coordination clients would remain unaffected. In our current toolset, we have sixteen coordinated components. It would negatively impact program stability to change all of them to add a single type of coordination to a single client, which would be required if we were using the `StudioCoordinator`.

Figure 6 about here

**Figure 6:** UML Diagram for object-oriented coordinator's interface

Both advantages of stable clients and that of a stable coordination derive their stability from limiting dependencies. Limiting dependencies helps ensure application stability. Coordination must be performed in a way that does not result in errors that impact effective user interactions. The more classes the coordinator class depends on, the more likely it is that the coordinator will break unexpectedly if these classes change.

Limited dependencies make the `GvtCoordinator` more robust than the object-oriented `StudioCoordinator`. The `GvtCoordinator` has dependencies on only two other classes besides the core Java libraries. Both of these classes are in the same package (namespace) as the `GvtCoordinator`. The object-oriented `StudioCoordinator` design depends on nine custom classes and implements a custom interface.

*Runtime control*

The `GvtCoordinator` enables end users to modify how components coordinate when users interact with an application in the GeoViz Toolkit. The `StudioCoordinator` does not allow the user any such control without rebuilding the application in the design mode of GeoVISTA *Studio*. We can call this the advantage of *runtime control*.

This advantage reflects one of the primary goals we had in developing the component-based coordination design for the GVT. With *Studio*, application *end-users* do not use or see the visual programming environment that *Studio* provides for application *designers* (see Figure 7). But this means that the application end-user has no way of controlling how components interact at runtime, because those mechanisms are only available on the visual programming side of the *Studio* environment. The `GvtCoordinator` overcomes this limitation and allows application end-users to redesign their toolkits on-the-fly.

Figure 7 about here

**Figure 7:** UML use case diagram showing the runtime control that the component-based coordinator enables

### *Granular control*

The `GvtCoordinator` in the GVT allows a fine level of control over how components interoperate. In the `StudioCoordinator`, a component connected to the `StudioCoordinator` would send and receive all the possible coordinated types of event. The `GvtCoordinator` allows individual components to send or not send any combination of potentially coordinated events. We call this the advantage of *granular control*.

A graphical user interface supports *granular control*. This interface allows users to connect and disconnect listeners from each other using the `GvtCoordinatorGUI`. This allows a user to share some types of events, while excluding others. The `StudioCoordinator` does not have this capability, because the linkages between components are defined at compile time (object-oriented), not run-time (component-based).

Figure 8 illustrates how granular control can be useful. In this design, there are two instances of the `GeoMap`. The two maps are coordinated in their color selection, but not in their data sources. This allows the user to determine whether patterns evident at one scale are evident at another. Here, using a bivariate color scheme, we can see that the general patterns of counties with a higher relative proportion of whites and of blacks are similar at the state and county scales, but not identical (the north-central and northwest U.S. in particular appearing to have a more uniform race distribution at the state level than is apparent at the county level).

Figure 8 about here

**Figure 8:** Coordination between two `GeoMap` instances. The color scheme is coordinated between the two maps, but the data are not coordinated.

### *Limitations of Introspective Observer Coordination*

There are some limitations of the component-based `GvtCoordinator` that deserve attention. Two particularly important limitations are discussed here. First, there is a tendency to pass references to objects as fields of events, which creates shared references to the objects, which can break encapsulation if internal representations are exposed. Second, components may become “over-coordinated” if components are broadcasting events with similar effects.

The first limitation stems from the fact that the easiest way to include information in an event is to add a field that provides access to it. For example, our current `SelectionEvent` functions just this way, having a method `getSelection()` that returns an array of integers, notated `int[]`. The potential problem is that multiple coordinator clients may acquire a reference to the same array. Arrays in Java are mutable objects, thus any of the coordinator clients can change the array, or assign it a null value, potentially breaking the other coordinator clients. A safer approach would be to impose a rule that events have a method that returns an

Iterator, which is an interface that specifies a means for reading the values of an array. Using an Iterator would allow for read-only access to an underlying array. The Iterator approach would be slower than the unsafe shared array approach, but probably faster than another safe alternative, which would be returning a “defensive copy” of the array (returning a copy of an object, to prevent the original from being modified) whenever the field is accessed. Figure 9 illustrates the problem and solution in a UML class diagram. The diagram on the left shows the structure of the default approach, which leads to shared read-write references. The diagram on the right shows the structure of the solution, using an Iterator.

### Figure 9 about here

**Figure 9:** Class diagram showing different patterns for a selection using shared array references (left) and an Iterator (right)

Another potential limitation of the *Introspective Observer Coordination* approach is that it relies on each component sending and receiving the appropriate kinds of events. The default behavior is to coordinate in all possible ways that the components are capable. This can lead to an “over-wiring” situation, with more connections than are desired. Events that have similar meanings are especially problematic, because if a component subscribes to multiple events with similar meanings, later events will undo or change the effects of the first to arrive, without the user being aware of what has happened. For example, if a component is listening for color arrays as well as classes that contain color picking algorithms, one may interfere with the other. A solution to this is less obvious than to the first problem. One option may be to have types of coordination assigned to particular types of task, and have the coordination change to fit the user’s needs (Gahegan 2003).

Although the approach has the limitations outlined here, we believe the *Introspective Observer Coordination* based coordination strategy is a sound one and it has demonstrated to be fairly robust in practice. Next, we elaborate on its advantages are embodied in working software.

### Advances and Advantages of the GeoViz Toolkit

The *Introspective Observer Coordination* strategy described above pays off in the following advantages in the GeoViz Toolkit: End users are able to make their own applications, interactive geovisualizations are tightly integrated with spatial analysis methods, and geovisualizations are sharable between users. In this section, we describe the above advantages of the Introspective Observer Coordination strategy along with some additional advantages of the GeoViz Toolkit that implements it.

The *Introspective Observer Coordination* strategy allows users to assemble a unique collection of visualization and spatial analysis tools in a single, tightly coordinated framework. The GVT advances the state of the art by providing a set of views that can be added to or taken away from the application using menus and mouseclicks. All data and representational coordination happens automatically in real-time, so that users can easily pursue real work. The component-based coordination of the GVT represents a substantial evolution beyond the visual programming environment of GeoVISTA *Studio* because allows application users to work as

application designers without knowledge of programming principles or limitations. Simply put, the software makes reasonable decisions about coordination without requiring user involvement.

The *Introspective Observer Coordination* strategy also allows for the integration of spatial analysis methods. Because the strategy is so general, different software developers can work on separate parts of the GeoViz Toolkit environment, and their separate efforts do not conflict with each other. For example, we have integrated four clustering methods from the Proclude (Conley *et al.* 2005) suite of clustering tools into the GeoViz Toolkit. These tools work well together because they share the *Introspective Observer Coordination* design method.

*Introspective Observer Coordination* also allows end-users to create and work with geovisualizations in an open-source, web-distributable format. By leveraging Java to XML marshalling tools (XStream (Walnes *et al.* 2008) in particular), representations of software components and their relevant states can be turned into XML documents. These documents can then be stored or shared appropriately. Keeping the software pieces as separate components makes what could be an unmanageable problem into a tractable one.

The development methodology for the GeoViz Toolkit also incorporates a number of best practices from commercial software development. These include unit testing, code coverage, and continuous integration. Unit tests are a simple and effective way to improve the quality and robustness of computer programs (Cheon and Leavens 2001). We used the JUnit unit testing suite, developing tests of particular functions as well as the robustness of the software against different types of data inputs. The completeness of tests can be evaluated with code coverage tools. Code coverage analyzes which code paths are used during an execution of the program. This allows developers to find unused code, or code which was not tested. We used the open source Emma tool for coverage analysis (Roubtsov 2001). One reason Emma was selected was the development of the EclEmma project, which provides integration with the Eclipse development environment (Hoffmann 2007). These techniques can be (and should be) used in concert to improve their effectiveness.

Finally, we have incorporated tools for importing data from SEERStat (Surveillance Research Program 2008), as well as other sources. These sources include popular data formats such as Excel spreadsheets and comma-delimited tables. We have also incorporated data export facilities, in the hopes that users can perform basic data handling tasks without having to use a commercial GIS. In the next section we describe how *Introspective Observer Coordination* helps enable geovisual analysis.

## **Exploring the 2008 U.S. Election With the GeoViz Toolkit**

We have identified four advantages of the `GvtCoordinator` – stable coordination, stable clients, runtime control, and granular control. In the following example, we demonstrate how these advantages relate to a real world application example.

To begin, we compiled a dataset of county results for the 2008 U.S. Presidential Election. A user might wish to explore election results to compare recent results to previous elections and

to evaluate possible relationships to socioeconomic and behavioral covariates. The initial task in this scenario could involve merging together multiple existing datasets – something that the GVT handles easily through having registered data sources with the `GvtCoordinator`. At the interface level, a user simply clicks on the “Load Data” file menu option and can choose datasets in common formats like CSV and XLS to merge with a boundary shapefile. The GVT identifies common keys and merges data sources automatically, using inner-join, left-hand join, and right-hand join semantics from relational databases.

Figure 10 about here

**Figure 10:** A scatterplot, bivariate map, and parallel coordinate plot tool showing 2008 Presidential Election Results in the GVT. The scatterplot and bivariate map show the percentage of votes for Obama and McCain (blue and red axes, respectively). From left to right the parallel coordinate plot shows percentage of Obama, McCain, and Bush (2004) votes, then percentage of Baptist and Catholic Residents, and finally the percentage of Kerry (2004) votes. Jefferson County, Alabama is highlighted in all views. In the parallel coordinate plot one can see that Jefferson County voted for Bush in 2004 but swung to Obama in 2008.

Once the appropriate dataset has been assembled, the GVT lets users add components on the fly to create an application. The advantage of runtime control allows this capability. There is no need to select tools and then launch the application – the process occurs in real time and can involve as many modifications as the user desires. For example, the user may first choose to add a bivariate choropleth map, a scatter plot, and parallel coordinate plot tool (Figure 11). After working with the views for awhile, the user may then decide to remove the parallel coordinate plot and add a table viewer, a histogram tool, and a univariate map (Figure 11). The coordination design of the GVT ensures that there is no interruption to work due to changes in the number or types of components.

Figure 11 about here

**Figure 11:** The GVT has been reconfigured on-the-fly to remove the parallel coordinate plot and scatterplot, and add a table viewer, histogram tool, and univariate map. The table viewer shows the entire raw dataset. The histogram and univariate map are showing different representations of the percent of people currently living under the poverty line. Randolph County, West Virginia is highlighted in each of the views. It has a high percentage of impoverished residents and McCain won by 14 points.

Let us assume the user would like to explore patterns in both the bivariate map and the univariate map. The advantage of granular control allows the user to set up different color and classification schemes for each map, while still maintaining highlight and selection coordination (Figure 12). If at some point the user decides to synchronize color and classification schemes between the two maps, they can link them via a context menu.

Figure 12 about here

**Figure 12:** At top left, a univariate map shows the percentage of Catholics by county. At top right a bivariate map shows the percentage of votes for Obama and McCain. The GVT allows granular control so that users can set up different representations of their data while still allowing coordination between views. At bottom, a selection and highlight has been performed and both maps reflect the change.

In this example, the advantages provided by stable coordination and stable clients are less visible, but remain quite important. These advantages ensure that the application is unlikely to crash or behave erratically while in use. They also allowed for the addition of controls for highlighting behavior, letting the user set the degree of transparency for de-selected items.

## Conclusions and Future Work

The core advance presented here could be extended by using other software design patterns besides the *Observer* pattern. Candidates include the *Strategy* pattern and the *Visitor* pattern. The Strategy pattern would allow for a GUI to be automatically generated to apply alternative algorithms to a common task, such as different clustering methods. The Visitor pattern would allow arbitrary operations to be performed on sets of objects, for example adjusting all color schemes in an application to change saturation.

A key future means of extending the GeoViz Toolkit will be through the G-EX Portal (Rinner 2007). The GVT will leverage the G-EX project in two primary ways: first, GeoViz Toolkit projects will be exported to the G-EX Portal, allowing them to be retrieved and worked on later. Second, analysis artifacts such as screen captures will be published to the G-EX Portal. Conversely, comments created in the G-EX Portal will be integrated into the projects that the GeoViz Toolkit uses.

The GeoViz Toolkit features advantages for application designers and end-users of geovisualizations. For designers, the GVT embodies a robust yet flexible software architecture for coordination that allows novel types of coordination to be added at run-time. For visualization users, that flexibility and robustness translates into a software package that contains leading edge geographic visualizations and analysis tools, yet remains usable without programming expertise.

## Acknowledgements

This research is supported by the National Visualization and Analytics Center, a U.S. Department of Homeland Security program operated by the Pacific Northwest National Laboratory (PNNL). PNNL is a U.S. Department of Energy Office of Science laboratory.

This material is based upon work supported by the National Institutes of Health under Grant # R01 CA95949-01

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

## References

- Adger, W.N., 2000, Institutional adaptation to environmental risk under the transition in Vietnam. *Annals of the Association of American Geographers*, 90, pp. 738-758.
- Andrienko, G. and Andrienko, N., 1999a, Interactive maps for visual data exploration. *International Journal of Geographical Information Science*, 13, pp. 355-374.
- Andrienko, G. and Andrienko, N., 1999b, Knowledge-based visualization to support spatial data mining. In *Advances in Intelligent Data Analysis, Proceedings* (Berlin), pp. 149-160.
- Andrienko, G. and Andrienko, N., 2002, Intelligent support of visual data analysis in Descartes. In *Proceedings of the 2nd international symposium on Smart graphics* (Hawthorne, New York), pp. 21-26.
- Andrienko, G., Andrienko, N., Jankowski, P., Keim, D., Kraak, M.J., Maceachren, A. and Wrobel, S., 2007, Geovisual analytics for spatial decision support: Setting the research agenda. *International Journal of Geographical Information Science*, 21, pp. 839-857.
- Azevedo, J., Beires, N., Charpentier, F., Farrell, M., Johnston, D., LeFlour, E., Micca, G., Militello, S. and Schroeder, K., 2000, Multilinguality in voice activated information services: The P502 EURESCOM project. *Speech Communication*, 31, pp. 369-379.
- Baduel, L., Baude, F. and Caromel, D., 2007, Asynchronous typed object groups for grid programming. *International Journal of Parallel Programming*, 35, pp. 573-614.
- Brodlie, K.W., Duce, D.A., Gallop, J.R. and Wood, J.D., 1998, Distributed Cooperative Visualization. *State of the Art Reports at Eurographics98*, pp. 27-50.
- Carr, D.B., Littlefield, R.J., Nicholson, W.L. and Littlefield, J.S., 1987, Scatterplot matrix techniques for large n. *Journal of the American Statistical Association*, 82, pp. 424--436.
- Cazzola, W., Ghoneim, A. and Saake, G., 2002, Reflective analysis and design for adapting object run-time behavior. In *Object-Oriented Information Systems, Proceedings*, pp. 242-254 (Berlin: SPRINGER-VERLAG BERLIN).
- Conley, J., Gahegan, M. and Macgill, J., 2005, A genetic approach to detecting clusters in point data sets. *Geographical Analysis*, 37, pp. 286-314.
- Cook, D., Majure, J.J., Symanzik, J. and Cressie, N., 1996, Dynamic graphics in a GIS: Exploring and analyzing multivariate spatial data using linked software. *Computational Statistics*, 11, pp. 467-480.
- Cook, D., Symanzik, J., Majure, J.J. and Cressie, N., 1997, Dynamic graphics in a GIS: More examples using linked software. *Computers & Geosciences*, 23, pp. 371-385.
- Dykes, J.A., 1997, Exploring spatial data representation with dynamic graphics. *Computers & Geosciences*, 23, pp. 345-370.
- Edsall, R.M., MacEachren, A.M. and Pickle, L., 2001, Case study: design and assessment of an enhanced geographic information system for exploration of multivariate health statistics. In *Information Visualization, 2001. INFOVIS 2001. IEEE Symposium on*, pp. 159-162.
- Fekete, J.D., 2004, The InfoVis Toolkit. In *IEEE Symposium on Information Visualization* (Austin, TX), pp. 167-174.
- Fisher, K., Friedman, J. and Tukey, J., 1974, Prim-9: An Interactive Multidimensional Data Display And Analysis System. (Stanford, California: Stanford Linear Accelerator Center).

- Gahegan, M., 1998, Scatterplots and Scenes: Visualization Techniques for Exploratory Spatial Analysis. *Comput. Environ. and Urban Systems*, 22, pp. 43-56.
- Gahegan, M., 2003, Personal Communication.
- Gahegan, M. and Hardisty, F., 2008, *GeoVista Studio*. In *Open Source Approaches in Spatial Data Handling*, G. Brent (Ed.)Springer).
- Gahegan, M., Takatsuka, M., Wheeler, M. and Hardisty, F., 2002, Introducing GeoVISTA Studio: an integrated suite of visualization and computational methods for exploration and knowledge construction in geography. *Computers, Environment and Urban Systems*, 26, pp. 267-292.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1995, *Design patterns: elements of reusable object-oriented software*.
- Greg, A. and Lloyd, T., 1995, An Extended Data-Flow Architecture for Data Analysis and Visualization. In *Proceedings of the 6th conference on Visualization '95*(IEEE Computer Society).
- Haug, D., MacEachren, A.M., Boscoe, F.P., Barnes, D., Mararra, M., Polsky, C. and Beedasy, J., 1997, Implementing exploratory spatial data analysis methods for multivariate health statistics. In *GIS/LIS* (Cincinnati, OH), pp. 205-213.
- Heer, J., Card, S.K. and Landay, J.A., 2005, prefuse: a toolkit for interactive information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (Portland, Oregon, USA: ACM).
- Homer, P.T. and Schlichting, R.D., 1994, A Software Platform for Constructing Scientific Applications from Heterogeneous Resources. *Journal of Parallel and Distributed Computing*, 21, pp. 301-315.
- Kassab, L.L. and Greenwald, S.J., 1998, Towards formalizing the Java security architecture of JDK 1.2. *Computer Security - Esorics 98*, 1485, pp. 191-207.
- Liu, J., Eker, J., Janneck, J.W., X., L. and Lee, E.A., 2003, Actor-Oriented Control System Design: A Responsible Framework Perspective. *IEEE Transactions on Control System Technology*.
- Masters, R. and Edsall, R., 2000, Interaction Tools to Support Knowledge Discovery: A Case Study Using Data Explorer and Tcl/Tk. In *Visualization Development Environments* (Princeton, New Jersey).
- Microsystems, S., 1997-2009, JavaBeans Available online (accessed January 14 2009).
- Monmonier, M., 1989, Geographic Brushing - Enhancing Exploratory Analysis of the Scatterplot Matrix. *Geographical Analysis*, 21, pp. 81-84.
- Neumann, G. and Zdun, U., 2002, Pattern-based design and implementation of an XML and RDF parser and interpreter: A case study. In *Ecoop 2002 - Object-Oriented Programming*, B. Magnusson (Ed.), pp. 392-414.
- Newton, C., 1978, Graphics: from alpha to omega in data analysis. In *Proc. Symposium on Graphical Representation of Multivariate Data*, Wang (Ed.), pp. 59-92(Academic Press).
- North, C. and Shneiderman, B., 2000, Snap-together visualization: can users construct and operate coordinated visualizations? *International Journal of Human-Computer Studies*, 53, pp. 715-739.
- Rey, S.J. and Janikas, M.V., 2006, STARS: Space-time analysis of regional systems. *Geographical Analysis*, 38, pp. 67-86.
- Rinner, C., 2007, A geographic visualization approach to multi-criteria evaluation of urban quality of life. *International Journal of Geographical Information Science*, 21, pp. 907-919.

Robinson, A., 2007, Synthesizing Geovisual Analytic Results. In *IEEE Visual Analytics, Science and Technology Conference Doctoral Colloquium* (Sacramento, CA).

Simons, A.J.H., 2007, JWalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Automated Software Engineering*, 14, pp. 369-418.

Surveillance Research Program, N.C.I., 2008, SEER\*Stat software (seer.cancer.gov/seerstat) 6.4.4

Sutherland, P., Rossini, A., Lumley, T., Lewin-Koh, N., Dickerson, J., Cox, Z. and Cook, D., 2000, Orca: A visualization toolkit for high-dimensional data. *Journal of Computational and Graphical Statistics*, 9, pp. 509-529.

Trenish, L.A., 1995, Visualization of scattered meteorological data. *IEEE Computer Graphics & Applications*, 15.

Unwin, A., 1999, Requirements for interactive graphics software for exploratory data analysis. *Computational Statistics*, 14, pp. 7-22.

Walnes, J. and Schaible, J., 2008, XStream. Available online at: <http://xstream.codehaus.org/>.

Watson, V.R., 2001, Supporting Scientific Analysis within Collaborative Problem Solving Environments. In *HICSS-34 Minitrack on Collaborative Problem Solving Environments* (Maui, Hawaii).

Weaver, C., 2004, Building Highly-Coordinated Visualizations in Improvise. In *Information Visualization, IEEE Symposium on*.

Wood, J., Wright, H. and Brodlie, K., 1997, Collaborative Visualization. In *IEEE Visualization '97* (Phoenix, AZ, USA).