

RITI-SICOM-WP20021

WHY SICOM?

**Advantages of Building GIS Applications with
RITI's Spatial Information Components**



February 20, 2002
274 Main Street, Suite 302
Reading, MA 01867

Tel./ 781.942.1655 Fax/ 781.942.2161
Website/ www.riti.com E-mail/ riti@riti.com

READING INFORMATION TECHNOLOGY INC.

TABLE OF CONTENTS

WHY SICOM?	1
I. Introduction	3
II. Reusable Components	3
III. Component Design Trade-Offs	4
III.1 Generality vs. Developer Productivity	4
III.2 Access Mode vs. Run-Time Efficiency	5
III.3 Putting It All Together	6
IV. Applying the Trade-Offs to GIS	8
V. SICOM	8
V.1 The SICOM Display Module as an Independent GIS Application	9
V.2 The SICOM Display Module as a Component	10
V.2.1 Pre-Configuring the SICOM Display Module	10
V.2.2 Accessing SICOM Display Module Properties and Methods	11
V.2.3 Application Architectures	13
VI. Review	14

I. Introduction

Spatial Information Components (SICOM) is a suite of components and tools for the rapid development and deployment of GIS applications. This is made possible by reusable components that balance key design trade-offs so as to favor rapid development, while still providing enough flexibility to support a wide variety of applications.

This paper is organized as follows: Section 2 provides a brief overview of reusable components; Section 3 discusses two key trade-offs that played a crucial role in the design of SICOM; Section 4 narrows the discussion of components to GIS, while Section 5 narrows the discussion to SICOM. Finally, Section 6 provides a review of the key points made in the previous sections.

II. Reusable Components

One of the most important ideas of modern software engineering is to minimize development time by maximizing the use of reusable components. A reusable component is a realization in software of an abstract concept, one or more instances of which can be easily integrated into a variety of applications. The phrase "realization in software" implies an object, i.e. a collection of properties and methods, where the properties are the data elements that define the object and the methods are functions to manipulate the properties. For example, the concept of a "button" is realized in a variety of GUI development suites. A button component has at least these essential properties: height, width, a title, and an outlet. The latter associates the button with a response function. A button has one essential method - to invoke the response function when the mouse is clicked on the button.

The properties and methods discussed in the button example are all public, i.e. they can be accessed from outside the object. However, not all of an object's properties and methods are necessarily public. As an object becomes more complex, it may use several private properties and methods to accomplish the objectives of its public methods.

For certain types of objects reusability is enhanced if some of its properties are *a priori* configurable, allowing for a tailored initial appearance and behavior. For example, all of the button properties mentioned above are *a priori* configurable.

Note: from this point on, the terms component and object shall be used interchangeably.

III. Component Design Trade-Offs

The provider of a component suite must decide what form its reusable components should take. In so doing, the provider must examine a number of trade-offs. The final form of SICOM is heavily influenced by consideration of two trade-offs:

- Generality vs. developer productivity
- Access mode vs. Run-time efficiency

Each of these is discussed below.

III.1 Generality vs. Developer Productivity

When designing a component suit, there is always a trade-off between generality, i.e. how broadly applicable the components are, and developer productivity, i.e. how rapidly users of the suit can develop and deploy applications. This trade-off between generality and productivity represents a continuum of possibilities. A specific component lies at a point along this continuum. That point is strongly linked to a characteristic called **granularity**, in a manner illustrated by Figure III-1.1.

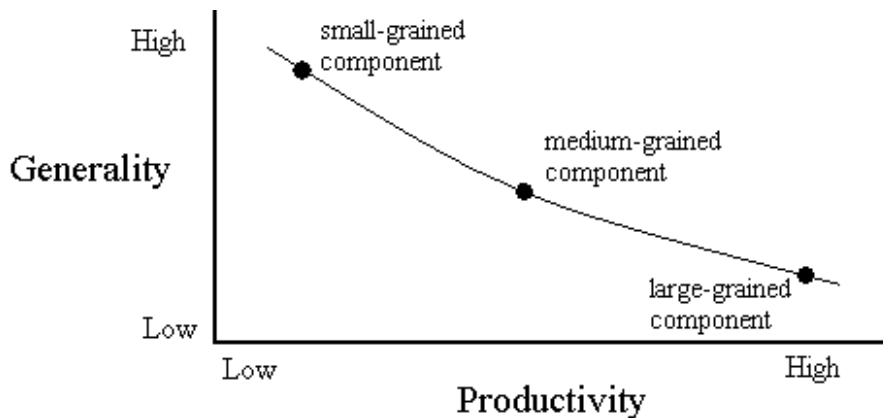


Figure III-1.1: Generality/Productivity Trade-off

Component properties may be simple, atomic data types, such as real numbers or integers, or properties may be other objects. Such objects may, themselves, contain simple properties or other objects. Thus, objects may range in complexity from those that contain only a few simple properties to those that contain a hierarchy of objects. The number and complexity of methods that an object has is typically commensurate with the complexity of its properties. A small-grained object is one that consists of a few simple properties and associated methods. A large-grained object is one that consists of a hierarchy of objects several levels deep. A medium-grained object lies somewhere in between. Note that not all of the complexity of medium- to large-grained objects is necessarily public. Many of the properties

and/or methods of some objects in the hierarchy or even entire objects may be private.

A small-grained component, e.g. the button, typically has broad applicability. As granularity increases, components necessarily represent more and more specific concepts and, therefore, applicability necessarily goes down. For example, while a button component can be used in any application that has a GUI, a component that represents, say, a speedometer can only be used in applications about vehicles. However, for those applications that are about vehicles, such a component would save the developer a considerable amount of work. The larger the granularity, the less original code is required to develop a given application on the one hand, and the more restricted the set of possible applications and their possible architectures on the other hand.

III.2 Access Mode vs. Run-Time Efficiency

In today's computing environment, there is also a trade-off between access modes, i.e. whether or not components are accessible across process (or even platform) boundaries, and run-time efficiency, i.e. the cost in computer resources to achieve access. Access across process (and platform) boundaries is referred to as remote access. We will refer to the remaining alternative as near access. Each type of access has an associated efficiency. Thus, for a given component, the access-mode/efficiency trade-off is essentially binary.

Conceptually, component methods are accessed via function calls. Properties are accessible via function calls as well (either implicitly or explicitly, depending on the language), but the functions are limited to "gets" and "sets", e.g. for a property named X, there might be functions getX and setX. A property may be read-only, in which case there is a "get", but no "set". Near access is achieved via **direct** function calls, while remote access is achieved via **indirect** function calls. In both cases, the developer simply codes a call statement (in the appropriate syntax of the development language). However, depending on whether the call is direct or indirect, it is translated into one of two distinct mechanisms.

The direct call is nearly as old as computing itself. Although somewhat hardware dependent, the mechanism works roughly as follows: the application places the function arguments on the stack and jumps to the function's first instruction; the function gets the arguments from the stack and performs its computations and returns, i.e. execution resumes at the instruction following the function call. If the function returns a value, it is handed back via the stack. There are two restrictions on this mechanism:

- The function must reside in the application's address space, i.e. it must be linked into the executable module at the time the application is built or at run time if it resides in a Dynamic Link Library (DLL).

- The details of how the arguments are handled are language dependent. Thus, the function must, in general, be coded in the same language as the application.

A suite of components that functions in this way is also known as a class library.

The indirect call is predicated on the existence of a server that "exposes" the components. An application that makes use of these components is called a client. Before making indirect calls, the client must establish a connection to the server. Once a connection has been established, the mechanism works roughly as follows:

1. A *direct* call is made to a proxy function that has the same name and list of arguments as the actual function.
2. The proxy function translates the arguments from the format of the client's language to a generic format and hands the arguments off to a transport mechanism, e.g. TCP
3. The transport mechanism organizes the data into packets and delivers and reconstitutes the data at the server-end of the connection.
4. The server translates the arguments from the generic format into the format of its own language, and makes a *direct* call to the actual target function.
5. If the function returns a value, the reverse process occurs.

The specifics of this process depend on the implementation approach or model. Microsoft's approach, the so-called Common Object Model (COM), is in widespread use on WINDOWS platforms. COM supports two types of server: in-process servers and out-of-process servers. The former cannot do anything useful without a client. That is, it functions like a class library that provides language independence. The latter is a separate application and thus can operate stand-alone.

Clearly, an indirect call has significant overhead as compared to a direct call, thus, relatively low efficiency. However, the advantages are that components need not reside in the client's address space, and need not be coded in the client's language. This means that a component can be a fully operational application (with its own GUI if desired), and that developers are free to choose from among a variety of languages when building their applications.

III.3 Putting It All Together

Clearly, when considering whether the same function is called directly or indirectly, efficiency has a binary nature. However, the efficiency of an interoperable component suite, taken as whole, can be seen more like a continuum, when the granularity of the components is taken into account. Imagine two component suites - S1 and S2, identical in every respect, except that S1 provides near access and S2

provides remote access. That is, we obtain S2 from S1 by wrapping S1 in a server that exposes S1's objects. Now, imagine an application implemented in two ways - I1 and I2, exactly the same, but built with suites S1 and S2, respectively. Although the developer of I2 was free to choose from a variety of languages, imagine that I2 was developed with the same language as I1. The source code for I2 looks virtually identical to that of I1, but I1 runs much more efficiently.

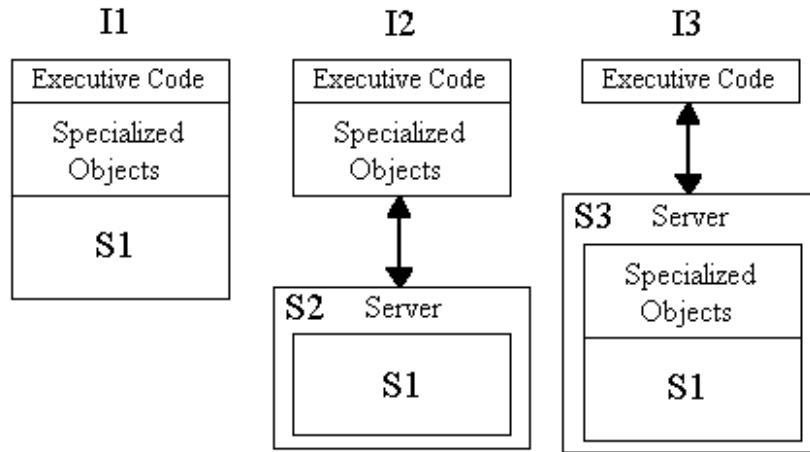


Figure III.3-1: Comparison of Three Implementations

Suppose that this application achieves its goals via a small set, relative to S1, of its own specialized objects, manipulated by a small amount of executive code. Suppose, in addition, that each of the specialized objects requires quite a substantial amount of original code and quite a large number of calls to S1's objects. Now, imagine building a third component suite - S3, consisting precisely of those specialized objects from I1 and the objects of S1. That is, we obtain S3 from I1 by wrapping a server around the specialized objects and S1, but exposing only the specialized objects. It is now possible to create a third implementation - I3, using calls to S3. Instead of making a large number of indirect calls as in I2, I3 achieves the same result with only a few indirect calls. The efficiency of I3 is substantially closer to that of I1.

If the original application is the only one that can use S3, then the creation of S3 is purely an academic exercise. However, if **a wide variety of applications** can make use of S3, then S3 is quite valuable. Not only are these applications almost as efficient as those created with S1, but they can be created much more quickly than with either S1 or S2. First, developers using S3 have a much shorter learning curve, because there are fewer components to learn. Second, there is far less original code to write, because the original code required to appropriately tie together the components of S2 (or S1) is now embedded in S3.

Of course, a certain degree of generality has been sacrificed, i.e. S3 implements a particular set of higher-level objectives in a particular way which may not be suitable for all applications that could be built with S1 or S2. However, for those applications where S3 is sufficient, this is irrelevant. In fact, for those applications

where S3 is sufficient, it would be unwise to use anything other than S3, because developers have less to learn, less to write, have a choice of languages, and the efficiency of the resulting applications is high.

IV. Applying the Trade-Offs to GIS

General-purpose development suites such as Microsoft Visual C++, Microsoft Visual Basic, and Borland Delphi provide GUI components, including forms, menus, sliders, text boxes, and buttons, to name a few. Many developer suites also provide other useful components, including strings, arrays, lists, etc. However, for applications such as GIS, such components, while helpful, do not come close to representing of the more complex GIS concepts. Therefore, an enormous amount of code must be written by the developer to capture those concepts. Thus, once attention is restricted to GIS applications, it is both appropriate and practical to think in terms of larger-grained components that adequately represent GIS concepts.

What is the appropriate granularity? There are certainly many concepts that can be realized as software components. These include geographic features of various types, ellipsoids, projections, coordinate systems, and maps, to name a few. A coordinate system object may, depending on what type it is, have an ellipsoid and projection as properties. A map will have a coordinate system and any number of geographic features as properties.

Component suites that realize these concepts provide medium granularity. Two examples of such suites are CARIS Spatial Framework and ESRI Map Objects¹. The trade-off between these two suites is exactly like the trade-off between suites S1 and S2, in the discussion of Section 3.3. CARIS Spatial Framework provides near access (C++) components, while ESRI Map Objects provides remote access components. While the latter has an advantage in not being restricted to C++ developers, it also has a serious disadvantage. At the medium level of granularity, a large number of calls to component methods are necessary to achieve moderate objectives. Given the high overhead on such calls, the efficiency of Map Objects must necessarily be low compared to CARIS Spatial Framework. Both suites require significant GIS expertise to be used effectively.

By contrast, SICOM is like suite S3 (in the discussion of Section III.3). That is, SICOM exposes a relatively small set of high-level, specialized objects that are useful in a wide variety of GIS applications; and those specialized objects achieve their objectives efficiently by making direct calls to the CARIS Spatial Framework (which plays the role of S1). Like suite S3, **SICOM balances the key trade-offs, such that developers have less to learn, less to write, and a choice of languages, while the efficiency of the resulting applications is relatively high.**

V. SICOM

The objective of SICOM is to provide developers with the ability to rapidly implement a wide variety of GIS-based applications, without restricting the developer's choice of

¹ They are trade marks of CARIS, Fredericton, NB, Canada and ESRI, Redland, CA, respectively.

language, without requiring significant expertise in GIS or COM, and without compromising efficiency. To achieve this objective, SICOM provides several large-grained, remote access components. The largest and most important of these is the SICOM Display Module (SDM). The SDM is implemented as an out-of-process server, and thus plays the dual roles of being an independent GIS application and being a reusable component. Each of these roles is discussed below.

V.1 The SICOM Display Module as an Independent GIS Application

As an application, the SDM presents an appearance that is typical of simple WINDOWS applications, incorporating the Microsoft Document/View Framework. That is, it presents a main window containing a title bar, menu bar, tool bar, a large client area, and status bar. The client area displays one or more views, each of which displays a document. Naturally, the documents displayed in the SDM's views are maps. Among the items on the menu bar are:

- **File Menu** - supporting opening, closing, printing, and exporting documents
- **Zoom Menu** - supporting zoom in or out, zoom to a user-specified scale, zoom to a rectangle drawn with the mouse, reset to display the entire map
- **Pan Menu** - supporting pan north, south, east, and west, center at mouse click, and drag with the mouse.
- **Help Menu** - providing access to an "About Box" and help for the standard menu items

The tool bar provides rapid access to many of these capabilities. Finally, the status bar shows the display scale of the active view, and provides a continuous readout of geo-location as the mouse moves over a map.

Each map displayed in an SDM view contains a collection of layers. Each layer contains a collection of geographic features, i.e. lines, points, polygons, or images. Layers are drawn into the view according to a predefined ordering, thereby determining what features are drawn on top of what. The number of layers, their drawing order, and the display attributes of their features, i.e. colors, line thickness and style, polygon fill style, choice of symbols to represent points, etc., are determined independently for each map, based on a Data Profile. Data Profiles are easily created using Data Profilers provided with SICOM. Any number of Data Profiles can be created for a single map, allowing the same data to be displayed in different ways for different purposes.

With the capabilities described thus far, SICOM is already able to provide a variety of map viewers and browsers, tailored to specific needs, without writing a single line of code.

V.2 The SICOM Display Module as a Component

As mentioned above, the SDM is implemented as an out-of-process server. Therefore, when used as a component, the SDM can and does provide the same appearance and capabilities described in Section V.1. This is a conscious choice that recognizes an important fact - the overwhelming majority of GIS applications need to display maps in one or more views, and provide a means to zoom and pan such views. The SDM menu bar, tool bar, views, and even the status bar, are things that most GIS applications will require or, at least, desire. Although modern GUI development suites reduce the burden of creating these things, it still requires time to design, code, and test them. The philosophy of SICOM is - why **take the time** to develop this same basic capability over and over again? This is the first of many ways that SICOM saves development time.

That said, the SICOM approach also recognizes that applications need to be tailored to specific needs. Thus, the SDM can be *a priori* configured, and its methods and properties accessed by client applications. Each of these aspects are discussed below.

V.2.1 Pre-Configuring the SICOM Display Module

The initial appearance of the SDM can be *a priori* configured, via (optional) configuration files that can either be placed in a special "Support" directory or at locations specified by environment variables. Among these configuration files are:

- **SDM Profile** - to configure the main window and its view(s)
- **Help List** - to configure the Help Menu and provide access to help documents
- **Tool List** - to add a "Tools" menu to the menu bar and provide access to corresponding components
- **Splash Bitmap** - to be presented as the splash image at start-up, and as the contents of the "About box".

A SDM Profile is created by the developer, using the SDM Profiler program provided with SICOM. The specific configurable items controlled by the SDM Profile are:

- The initial size and position of the main window
- The title appearing in the title bar
- An (optional) initial document to be opened and displayed automatically at start-up, or an (optional) initial location in the file system to position the Open Dialog
- Whether the application uses Microsoft's Single Document Interface (SDI) or Multiple Document Interface (MDI)

- An (optional) overview for each view, i.e. a sub-view within the view that always displays the entire map and indicates, via a rectangle, the portion of the map displayed in its parent view. The position and size of the overview, relative to the parent view, are also configurable.

A Help List is created by the developer using a simple ASCII editor, such as "Note Pad". Each line in the file defines an additional Help Menu item and the corresponding file to be opened when that menu item is selected. The file can be a Microsoft Word document, a PDF, an ASCII text file, or any other format, provided the corresponding application is installed on the system.

When a Tool List is provided, it causes an additional menu (called "Tools") to be placed on the menu bar. The Tool List is also created with an ASCII editor, and defines one menu item per line. Each line associates a menu item with a registered component. SICOM comes with several such components installed. They are:

- **Layer Annotator** - displays textual attributes corresponding to the features in a layer
- **Layer Classifier** - color codes the features in a layer, based on selected attributes
- **Layer Prioritizer** - reorders a map's layers, changes layer visibility, or both
- **Feature Selector** - provides mouse-based feature selection and display of properties
- **Feature Modifier** - allows modification of feature color and/or location.
- **Surveyor** - accurately measures the distance and bearing between two points on a map.

One or more of these components can be seamlessly integrated into an application, simply by editing the Tool List. (The developer can also create his own tools.)

By using easily created configuration files, applications can be developed that go beyond simple viewers and browsers to those providing analytical capabilities. In addition, such applications can have a tailored appearance and identity. Note that this is achieved, still, without writing a single line of code.

V.2.2 Accessing SICOM Display Module Properties and Methods

For fully customized applications, the SDM provides access to its properties and methods. As a large-grained component, the SDM is an object hierarchy. This

object hierarchy is easy to understand because it is structured exactly like the SDM application, as described in Section V.1.

The top of the hierarchy is the **ProgramAPI**, which is accessible through the WINDOWS registry. It provides access to high-level properties and methods of the SDM, including support for further customization of the main window and its menu. It also provides access to the next level in the hierarchy, which consists of two objects - the **MouseModeAPI** and the **ViewMngrAPI**. The former provides methods to control mouse behavior in SDM views and obtain information about user actions with the mouse. The latter creates, destroys, and arranges views, and provides access to the next level in the hierarchy, a collection of ViewAPI objects.

Each **ViewAPI** object provides access to the properties and methods of one view. Through these, the client can

- 1) change the state of the view window, i.e. minimize, maximize, restore, resize, reposition the window,
- 2) effect the relationship between the view and its map, i.e. center at a location, center at a feature, zoom in our out by a any amount, zoom to a rectangular extent, and
- 3) output the contents of the view, i.e. print, copy to clipboard, or output an image file.

Each ViewAPI also provides access to the next level in the hierarchy - its own **OverviewAPI** (if overviews are defined in the SDM Profile) and its own **MapAPI**. The former, if available, provides access to properties of the view's overview window, allowing the client to minimize, restore, resize, and reposition it. The latter provides access to the properties and methods of the view's document, i.e. its map. These include

- 1) access to properties of the map, such as its name, location on the file system, resolution, and number of layers,
- 2) geometric methods to compute distances and bearings, to perform coordinate transformations, and to query relationships among features and
- 3) methods to determine the characteristics of selected features.

Each MapAPI also provides access to the next level in the hierarchy - its own set of **LayerAPI** objects, one for each layer in the map, and its own (transient) **FeatureAPI**. Each LayerAPI object provides access to the properties and methods of one of the map's layers. These include access to the properties of the layer, such as its name, type, number of features, etc. and access to methods to perform layer annotation and classification. The FeatureAPI manages the color, visibility, and disposal of transient features (features added to temporary map layers by the client), and provides access to type-specific transient feature components: **PointAPI**, **LineAPI**, **ShapeAPI**, **TextAPI**, and **ImageAPI**, which allow the client to add and modify features of the each type, respectively.

Because of this simple hierarchy that essentially reflects the visible structure of the SDM, the developer can learn to use it relatively quickly, yet build a wide variety of highly customized GIS applications. Furthermore, one indirect call to an SDM method (that manipulates the GIS data or performs GIS calculations) does a lot of work, via direct calls to the medium-grained components of the CARIS Spatial Framework, knitted together by specialized code. Thus, the amount of custom code is significantly reduced, while run-time efficiency remains high.

V.2.3 Application Architectures

When thinking for the first time about a client making indirect calls to the SDM, it is natural to think in terms of the simple client/server architectural model shown in Figure V.2.3-1.a. In this model, the end-user starts the client from the desk-top or the file system. The client then connects to the SDM, thereby starting it. Thereafter, the client makes indirect calls to the SDM, as described in Section V.2.2. However, the SDM has the capability to make a small, prescribed set of indirect calls back to the client. This allows for additional architectural models, the most basic of which are shown in Figure V.2.3-1.b and V.2.3-1.c.

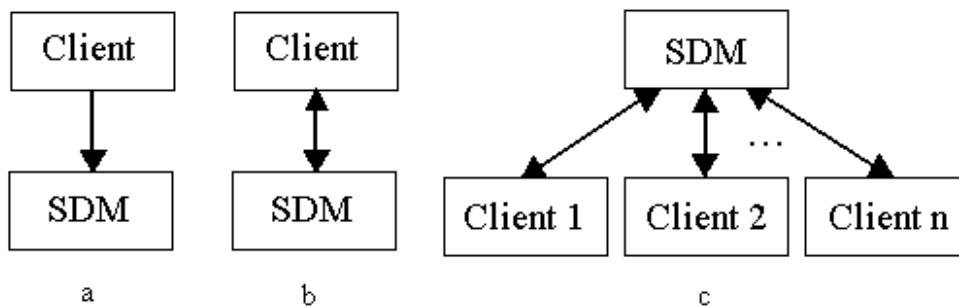


Figure V.2.3-1: Architectural Models

Model-b (Figure V.2.3-1.b) is started the same way as model-a (Figure V.2.3-1.a), but in model-b the SDM also makes indirect calls back to the client. It is the MouseModeAPI that provides this reverse capability, allowing tighter coordination between the client and the SDM in responding to mouse operations. In particular, model-b makes it easy to sustain a mouse mode for a series of end-user operations. Both model-a and model-b have the virtue of simplicity, and also have the opportunity to further configure the SDM before it actually becomes visible.

In model-c (Figure V.2.3-1.c), the end-user starts the SDM directly. In this model, the SDM is pre-configured to display custom menu items. When the end-user selects such a menu item, the SDM starts a corresponding client. Such a client immediately connects back to the (already running) SDM, so that it can make any of the standard indirect calls, just like the clients in model-a or -b. (This is, in fact, how the SDM interacts with the tools

described in Section V.2.1.) The GUI presented by such a client appears to the end-user as an integral part of the application. The end-user neither knows nor cares that a separate process is responsible for presenting and managing the GUI.

Model-c is supported not only by the Tool List configuration file, but also a Manager List configuration file. This looks and operates like the Tool List, but the name of the new menu that appears on the SDM menu bar is defined in the configuration file. Clients that are defined in the Manager List are collectively call managers. The distinction between managers and tools is largely one of taste. However, the general guideline is that tools are more independent, and applicable to a broader spectrum of applications than managers, while managers work together to achieve the objectives for one application or a class of similar applications.

Hybrid models are also possible. For example, one client can have the special status of starting and configuring the SDM as in model-a or -b, but that in no way precludes other clients being involved as in model-c. In addition, clients in model-c can utilize the reverse capability of the MouseModeAPI, as in model-b. It is also possible to have several clients connect to the SDM in the style of model-a or -b.

Through these architectural models, the SDM supports and fosters modular design. A complex application can be broken down into several independent client processes that work together to achieve desired objectives. Such a collection of clients can be developed, tested, and maintained independently of one another, to a large degree. This provides a natural breakdown of responsibility for a team of developers, so that they can work together without getting in each other's way.

VI. Review

We have seen that reusable components provide a powerful mechanism for shortening the software development process. However, component suites can differ greatly, due to decisions that have been made regarding important design trade-offs. One of these important trade-offs is **generality vs. developer productivity** that, in software terms, translates into small-grained vs. large-grained components. Another important trade-off is **near or remote access vs. high or low run-time efficiency**, respectively. Although the latter implies lower run-time efficiency, it also implies language-independent access to components through a server that may be either embedded in the client application or a stand-alone application.

The design philosophy of SICOM is based on three important facts:

1. There exists a broad spectrum of GIS applications that have so much in common that it becomes possible to boost developer productivity through the use of large-grained components, without sacrificing a great deal of generality.

2. Through the careful use of large-grained components, it becomes possible to have the language independence of remote access components, while mitigating to a large degree the associated impact on run-time efficiency.
3. Once it has been concluded that large-grained, remote access components are the best choice, and given that the commonalities of GIS applications extend to GUI elements, servers should be stand-alone applications that provide those elements.

Thus, SICOM provides a collection of large-grained, remote-access servers, each of which is an application with its own GUI. One of these, the SICOM Display Module (SDM) is the central GIS engine. The remaining components are tools that are invoked via the SDM menu and extend its functionality. Many aspects of the appearance and behavior the SDM, as well as maps displayed by the SDM, can be *a priori* configured easily, using Profiler programs supplied with SICOM. With these capabilities alone, a significant variety of applications can be created without writing a single line of code.

For those applications that require further customization, the SDM exposes a robust set of properties and methods that support access to and control of GUI elements, views, maps, and map layers and features. The SDM also has the capability to "talk back" to clients, via a small, prescribed set of calls. With these, the developer can create various types of client applications, including clients that invoke and further configure the SDM before it appears, additional tools, and managers - another special class of client that seamlessly extends the functionality of the SDM. In this way, the SDM supports a variety of modular architectures.

Because of its large-grained design, SICOM is not appropriate for every conceivable GIS application. However, for the same reason, it represents the best choice for the vast majority of applications. For these applications, the developer either writes no code at all, or far less code than that required by medium-grained component suites. Furthermore, when it becomes necessary to write code, the learning curve is shorter, because all of the complexity of GIS is hidden inside the large-grained components. In addition, because SICOM supports remote access, the developer is free to choose any development environment that supports Microsoft's Common Object Model and, because of the large-grained design, the overall run-time efficiency is not far below that of near-access component suites.

Thus, for the developer using SICOM, productivity is enhanced in three ways:

- 1) less code to write,
- 2) shorter learning curve, and
- 3) freedom to use a familiar language and development environment.

This significant productivity enhancement is achieved without an equivalent sacrifice in either the variety of applications that can be built or the run-time efficiency of these applications.