An integration platform for heterogeneous bioinformatics software components

by A. C. Siepel
A. N. Tolopko
A. D. Farmer
P. A. Steadman
F. D. Schilkey
B. D. Perry
W. D. Beavis

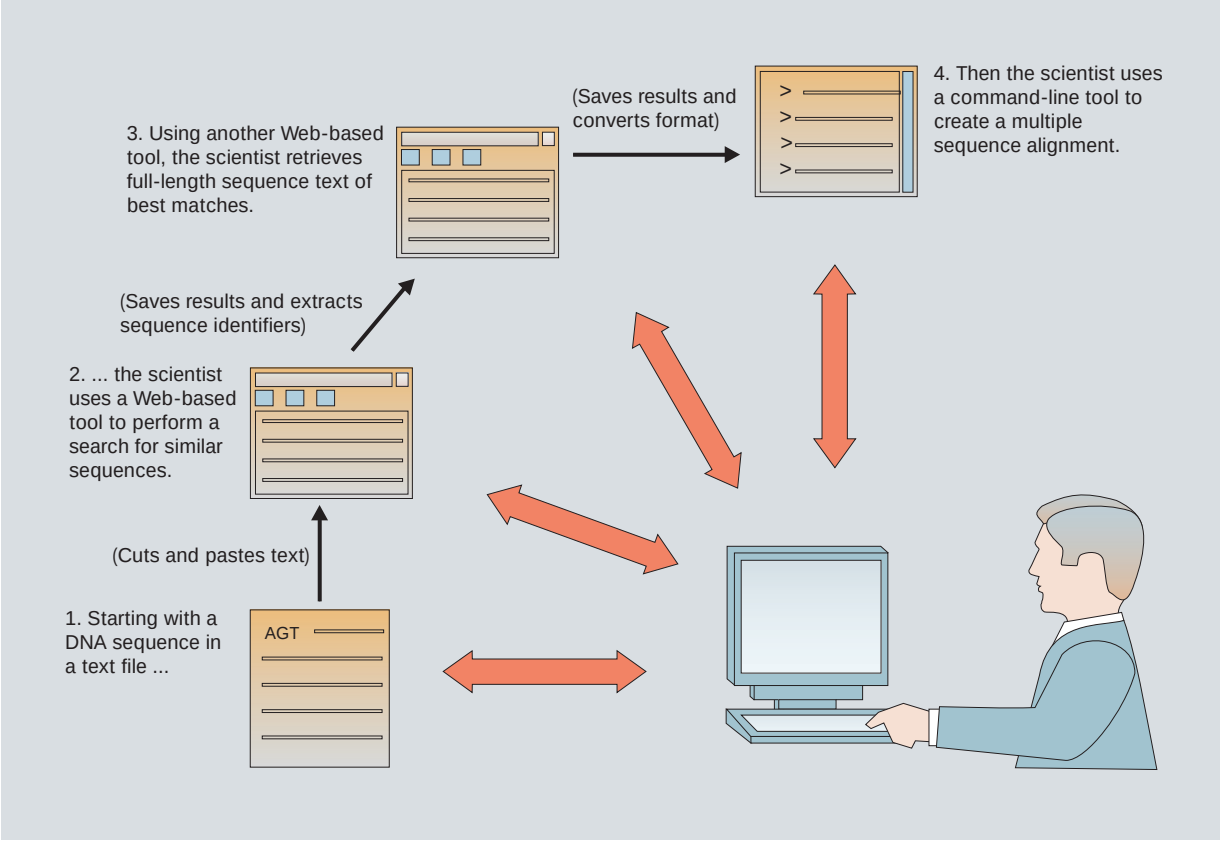
Although computer programs and database resources for bioinformatics applications are becoming more widely available, these resources are unstandardized and frequently incompatible. The problem of integrating heterogeneous software is of immense importance to the field, especially because a rapid pace of change and a general scarcity of development resources discourage re-engineering and compel developers to find ways to use legacy resources. In this paper, we describe an approach to the problem of integration of heterogeneous bioinformatics resources that relies on a generalized software platform, written in the Java™ language, that we call ISYS™. The ISYS platform employs techniques for interoperation among loosely coupled components, such as brokered service exchange and mediated event exchange, that are increasingly common in software engineering but still not used widely in bioinformatics. In addition, it further promotes loose coupling of independent components through a flexible, semistructured data model that supports run-time association of attributes with objects, and allows different components to maintain different "views" of the same object. We describe our general approach, the architecture of the system, the mechanics of event and service exchange, and the implementation of the data model. The platform is not restricted in its utility to bioinformatics, and could be useful for any rapidly changing field in which the integration of heterogeneous legacy components is important.

The field of biology is becoming increasingly dependent on computer software. Molecular and cellular biologists, geneticists, and biochemists rely on analysis programs, modeling tools, databases, and visualization software to accomplish their everyday work. As is true for many scientific fields, however, software development to support biology is more of a craft than a professional engineering discipline.[1] Separately funded groups—public and private—independently create custom tools, often for narrowly conceived, short-term needs. Scientists "moonlighting" as computer programmers write software to support their research, but try to avoid being consumed by engineering concerns tangential to their main interests. In addition, the extremely rapid pace of change of the field—in terms of science, technology, and business—discourages the development of stable standards for interoperation and the formation of commercial companies that can deliver specialized software at reasonable prices.

This disunity and decentralization—which are natural and necessary in any vibrant, developing scientific field—unfortunately result in crippling incompatibilities among software tools and databases. Scientists find it slow, cumbersome, and labor-intensive to establish the connections across information resources that fuel scientific synthesis. At the same time, mounting pressure for "functional genomics" (the identification of the functions of previously characterized genomic structures) makes the need for integration even more acute. For example, a powerful tool for functional inference called "comparative genomics"[2] often requires the integration of data from various specialized databases and can benefit

0018-8670/01/$5.00 © 2001 IBM

Figure 1 Typical sequence of steps a scientist might take for a simple analysis task



enormously if those databases interoperate readily with one another and with assorted visualization and analysis tools.

Figure 1 describes a typical sequence of steps a scientist might take to accomplish a relatively simple analysis task. To examine a DNA (deoxyribonucleic acid) sequence in alignment with similar sequences from a public database (that is, arranged so that homologous portions of the sequences are vertically aligned), the scientist must use three different tools with three different interfaces and convert the output from each one to a format acceptable as input to the next. For more complex analysis, many other resources might be required. One of our colleagues, a quantitative biologist who does comparative genomics using gene expression data, claims to spend more than half of his time on tasks related to the integration of data from incompatible databases and software programs. Moreover, this approach of converting the output of one program into acceptable input for another greatly limits the integrative potential between components that are highly interactive in nature.

Recognizing that integration is essential, but that rapid change and limited development resources prohibit complete re-engineering of legacy components, researchers and commercial companies have tried various "bottom-up" approaches to allow interoperability of formerly incompatible resources.

One common approach, used both by public systems[3–6] and commercial application service providers,[7,8] is to provide a uniform Web interface to various databases and analysis tools. These systems usually use CGI (Common Gateway Interface) scripts or Java** servlets to execute queries against databases, to call analysis programs, or to search file-

based data repositories. Some allow users to send output from one program as input to another without concern for schematic representation or file formats. Many of these systems are ingenious and widely used, but all are ultimately limited by the Web-browser/CGI model in terms of user-interface richness and responsiveness. In addition, they generally permit little customization by the user and are not designed to allow "plug and play" of components.

A second approach, focused on data access rather than analysis or visualization, allows complex declarative queries that span multiple heterogeneous databases.[9–11] This approach has received considerable attention in bioinformatics and has given rise to several interesting systems. However, it either requires problematic "on-the-fly" mappings[12] from representations in source databases to a definitive ontology (roughly, a global schema), or forces users to express queries in the sundry schemas of source databases. Moreover, many cross-database query systems assume a separation of user interface development and data integration. We believe this ignores the need for exploratory query formulation that users experience with inconsistent and unstandardized biological databases.[13] Finally, these systems can be insufficiently flexible (e.g., to the addition or subtraction of source databases or to changes in a global ontology) for such a turbulent field.

A third approach is to package heterogeneous software tools and databases as components adhering to standard, well-defined interfaces, according to which information can be exchanged. This approach encourages components to encapsulate their differences and expose only minimal, abstract attributes and behaviors. Perhaps its strongest advocate is the Object Management Group (OMG), which hosts a "task force" for the life sciences[14] as well as for many other domains. The OMG promotes the development of standard interfaces using its interface definition language (IDL) and implementation of components using CORBA** (Common Object Request Broker Architecture**). The component-based approach has a number of advantages for the problem of integration, in addition to the design, development, and maintenance strengths of component-based design in general.[15] For example, it enables interoperation with minimum homogenization and allows components implementing the same abstract interface to be interchanged. In bioinformatics, several groups have embraced this approach.[16–19] Its main stumbling block is the standardization of interfaces. The OMG defines its standards by committee, in a way that is strenuously fair but slow and arduous. It chooses to define interfaces in relatively specific terms, which encourages exchange of data without loss of information but makes standards harder to agree upon and more constraining for implementers. In addition, the specification of standard interfaces, useful as it is, still does not address how components are to be integrated. System builders are free to integrate them as they see fit, and sometimes write top-level controllers that instantiate and call components directly. In this way, they fail to take full advantage of the great potential for flexibility offered by component-based design.

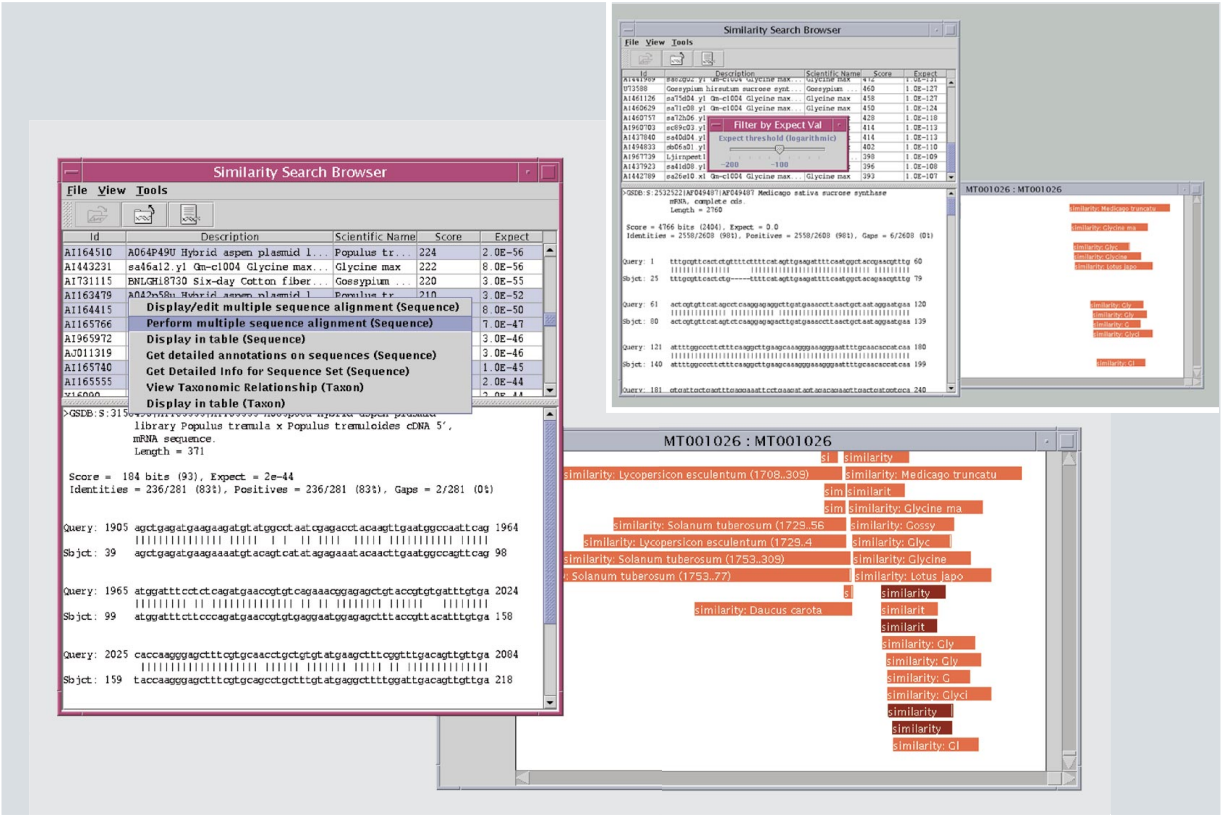
Our method is related to this third approach, but focuses more on component integration and less on component standardization. We have designed and implemented a client-side integration platform (the ISYS** platform) that allows interoperation of components adhering to a minimal set of standards. A decentralized and highly flexible "integrated system" results from the interactions among the components present in a given configuration. The focus is on client-side integration, but proxies to server-side resources allow access to local or remote servers. In general, both legacy software and new development are accommodated relatively easily. We believe the ISYS platform could be useful in any domain in which flexibility is of paramount importance and there is a strong incentive to integrate already-existing heterogeneous components.

The ISYS approach

The ISYS approach is loosely inspired by complex systems such as cellular automata and Boolean networks,[20,21] in which the interaction of simple, independent agents results in the emergence of complex properties. The idea is to define a simple set of ground rules and mechanisms by which disparate components can interoperate in useful and constructive ways (and which discourage distracting and nonintuitive types of interaction), then to allow their interactions dynamically to define the behavior of the system.[22] In this way, we accept the diversity of software in our field, rather than struggle against it.

In ISYS, each component interacts directly only with the platform, which serves as the medium for the exchange of events and services and permits components to remain decoupled. The exchange of events allows components to synchronize their behavior, and the exchange of services allows them to draw on

Figure 2 Screen shots of ISYS prototype in pursuit of goals of Figure 1 (first part). Inset (upper right) shows application of search filter.



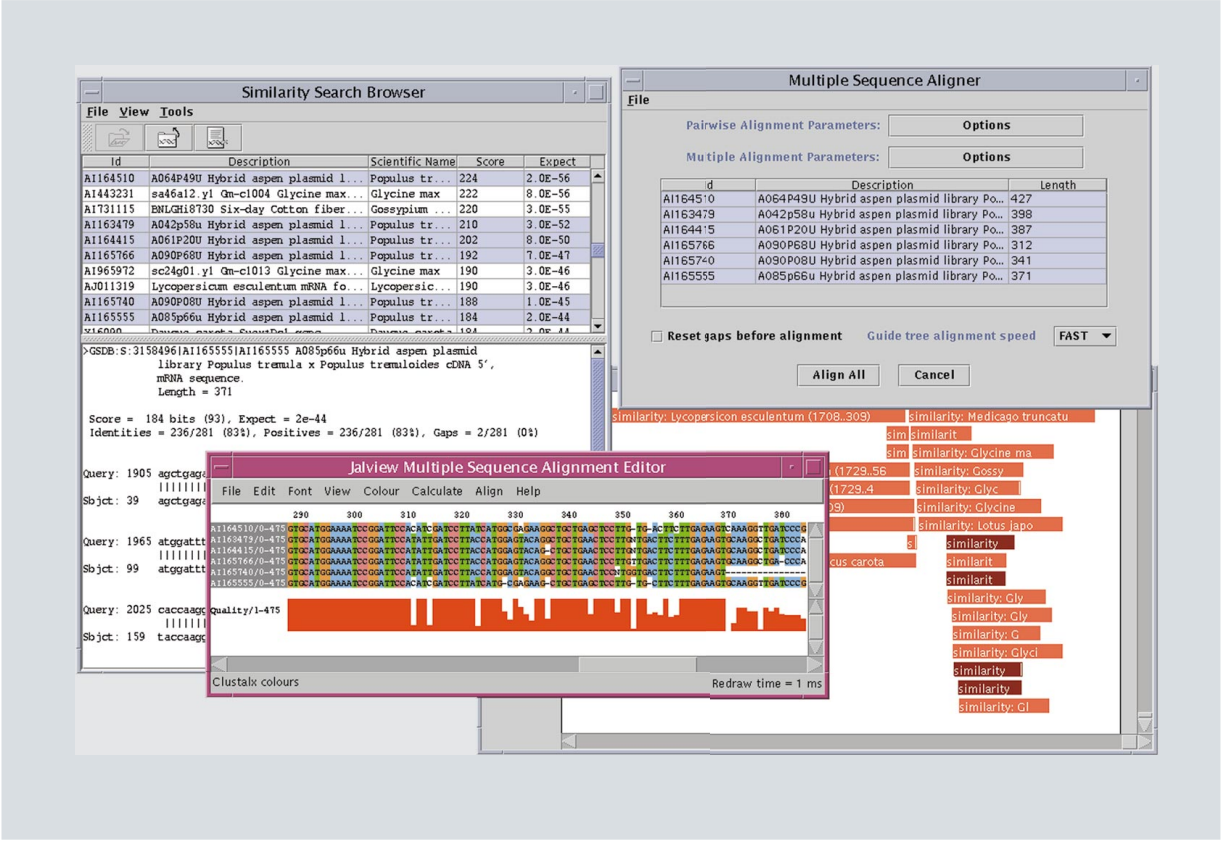
one another's capabilities to retrieve data, perform analyses, or present user interfaces.

Figures 2 and 3 describe the integration of several components using the ISYS platform, as seen by a user of the current prototypical implementation. The user is performing the same analysis steps executed in Figure 1.

Figure 2 depicts two components, the *SequenceViewer* (right) and *SimilaritySearcher* (left). The SequenceViewer component displays colored bars representing a DNA sequence and its annotations (parcels of information about sequence segments based on laboratory experiments or computer analysis). The sequence is represented at the top (off the screen here), stretching from left to right. The annotations are tiled beneath it, each bar color representing a different type (e.g., gene, exon, intron, transcription factor binding site, and region of high similarity to another sequence). The tool lets users scroll and zoom the display and view detailed description of annotations. In Figures 2 and 3 all visible bars represent regions of high similarity to other sequences (i.e., all are of the same type; the ones of darker color are simply selected).

The SimilaritySearcher component launches searches of single query sequences against large databases and displays search results. Figures 2 and 3 show the SimilaritySearcher results browser (i.e., the search has already been launched and the results returned). The top pane of the browser is a summary table of the search results, each row of which represents a single matching sequence from the background database. The bottom pane displays details about the first selected match, including an alignment of the query sequence and background sequence in the regions of high similarity. The search

Figure 3 Screen shots of ISYS prototype in pursuit of goals of Figure 1 (second part)



was executed using a popular program called *BLAST*.[23]

These components were developed separately and have no direct dependencies on one another; nevertheless, they appear to the user to be closely integrated. The SimilaritySearcher component was originally invoked from the SequenceViewer component, which passed it the text of the displayed sequence for use as a query sequence (using ISYS, the user began in the SequenceViewer component, rather than with a text file as in Figure 1). The annotations visible in Figure 2 in the SequenceViewer display were not originally present; they reflect the similarity search hits in the other component and appeared only when the search returned and the browser appeared. Selection and visibility of the search hits are synchronized in the two components. When the user selects items in the table in the top pane of the SimilaritySearcher display, the corresponding annotations are selected in the SequenceViewer display (as seen in Figure 2). Similarly, when the user causes items to disappear in the SimilaritySearcher display, they disappear also from the SequenceViewer display. This can be seen in the inset box of Figure 2, which shows the application of a filtering tool to the similarity search results (the tool filters by BLAST's "expect value," a statistical score indicating the significance of sequence similarity) and the corresponding disappearance of annotations in the SequenceViewer display. All of this synchronization occurs via the broadcast and reception of generic events, which we explore in detail in the next section.

The pop-up menu obscuring part of the SimilaritySearcher display in Figure 2 appeared when the user clicked the right button of the mouse after selecting several of the rows in the search-results table. The options appearing in this menu were dynam-

ically generated and reflect the set of components installed and registered with the ISYS platform. Using a process we call *interactive discovery*, registered components have been interrogated about the selected data set. Those that respond that they can operate on the data set have been represented in the menu with appropriate descriptions. The interactive discovery process is central to ISYS. This mechanism, which is available from most ISYS components, encourages an exploratory mode of usage in which new paths through the system emerge dynamically, according to selected data and available components. As with graphical synchronization, the exchange of services does not require components to have direct knowledge of one another. We will discuss the mechanics of service exchange also in the next section.

If the user selects the option "Perform multiple sequence alignment" from this pop-up menu, he or she is presented first with a simple interface to a program called *CLUSTAL W*,[24] which computes a multiple alignment of the selected sequences, then with a program called *JalView*,[25] which provides a colorful, editable display of the alignment. Both interfaces are shown in Figure 3. These components illustrate three of the most common ways to integrate existing resources using ISYS. First, CLUSTAL W, which is available as a command-line program written in the C language, has been wrapped with a service provider, a component that registers abstract services with the ISYS platform but hides how they are implemented. Second, JalView, available as a Java application, has been wrapped with a simple Java class that makes it appear to ISYS like any other client. This wrapper handles the broadcast and reception of ISYS events and transmits them to JalView in terms it can understand. Third, the full-length sequences required for the multiple sequence alignment, not available in the SimilaritySearcher component (it knows only of segments of database sequences), were retrieved from a public database of DNA sequences.[26] This occurred transparently to the user by way of another service provider, which acts as a proxy to a resource available on the Internet (behind the scenes, this proxy makes calls to a server for the database using Java Remote Method Invocation). We examine these mechanisms, too, in the next section.

Note that the advantages of ISYS with respect to the scenario in Figure 1 go beyond convenience and efficiency. Even though the user in this discussion followed an anticipated course of action with ISYS, the interactive discovery process suggests paths that one might not have imagined. In addition, through the interactions of separate components, ISYS can provide new capabilities and new perspectives on biological data. For example, when the user filtered the search results and observed the effect in the SimilaritySearcher display (inset, Figure 2), he or she was able to view the physical locations of search results above a certain threshold—something permitted by neither component individually.[27]

System architecture

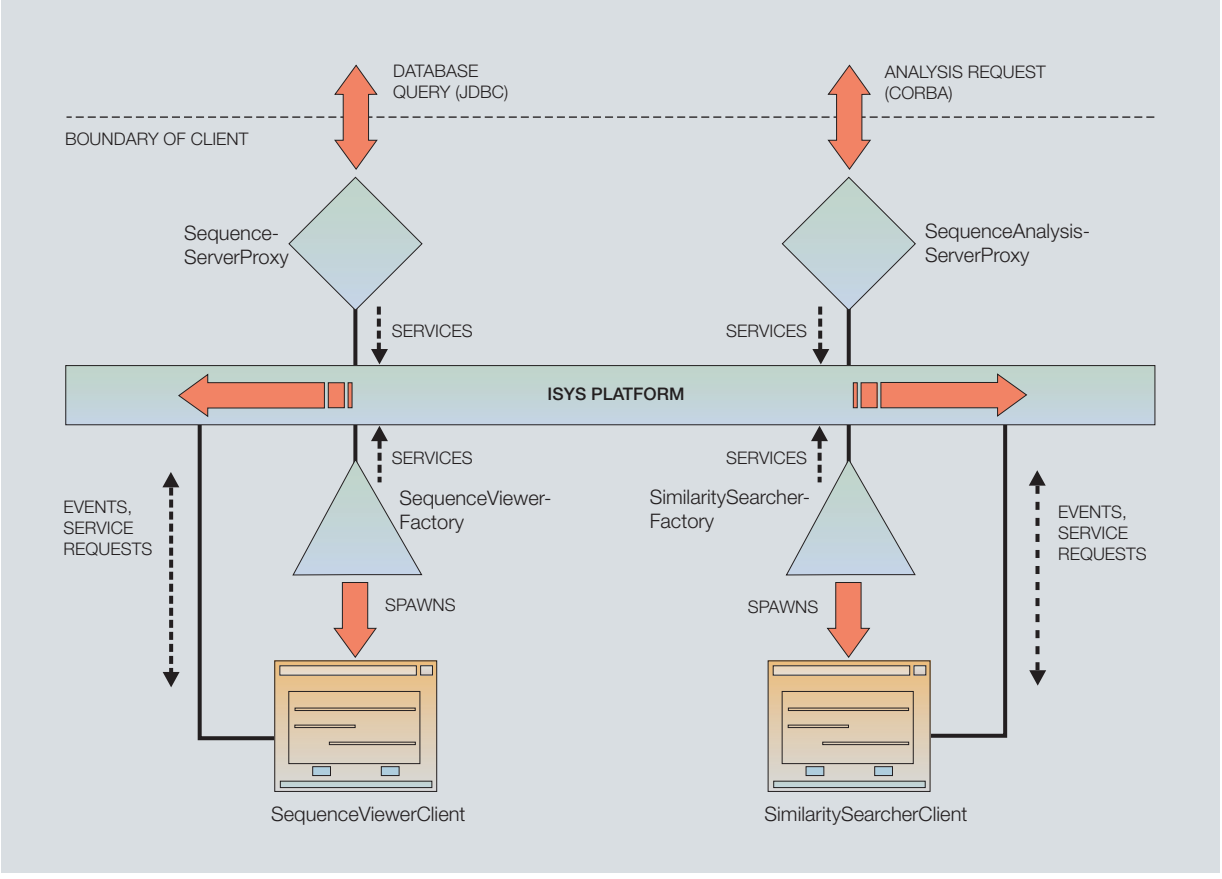
The architecture is based on the principle that no component should depend on direct knowledge of any other, e.g., in terms of specific classes or interfaces. At the level of the platform only abstract events, services, and classes of components are defined. Nearly all specific functionality resides within the components themselves. The system must behave appropriately regardless of what particular components are present. At run time, components register as providers of types of services and as listeners of types of events; they broadcast events and request services. Components must be prepared for the possibilities that services are not available and that no other component is listening to broadcasted events.

Figure 4 shows the components active in the scenario of Figure 2 and their interactions, which are accomplished by the exchange of events and services through the integration platform. We will discuss the integration platform, the exchange of services, the exchange of events, the roles of components, and the data model that unifies components. As we do so, we refer to Figure 4, and to the scenario of Figures 2 and 3.

In our discussion we introduce several terms for different types of ISYS components. Figure 5 shows a taxonomy of these terms to illustrate their relationships. In the figure, labels in italics represent abstract categories for classification purposes only and nonitalicized labels represent concrete categories actually reflected in the code. Of the latter, those having solid, bold outlines are implemented as Java classes, and those having dashed outlines are implemented as Java interfaces. The categories highlighted by the shaded oval represent nonexclusive roles, of which a single component is allowed to play more than one (although multiple roles are rarely used).

The platform. The ISYS platform consists of three components: *EventChannel*, *ServiceBroker*, and *ISYS Client Environment* (ICE). The EventChannel component is responsible for the exchange of events and the

Figure 4 Interactions among components from Figure 2

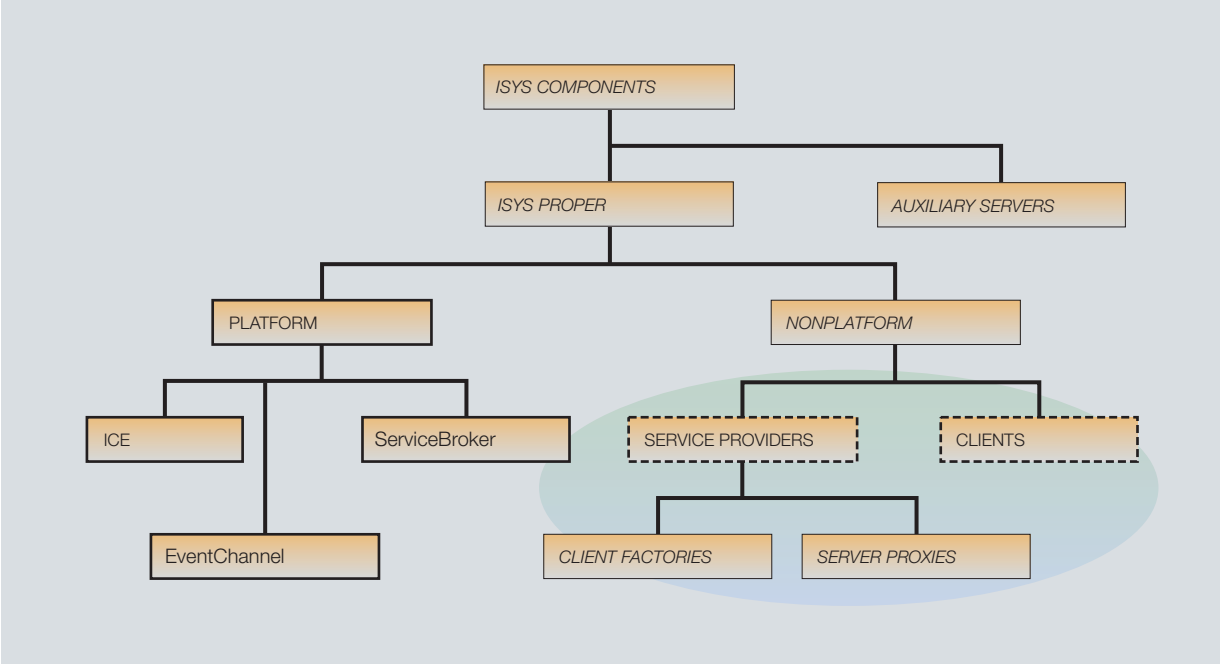


ServiceBroker component for the exchange of services.[28] We will discuss both further in our treatment of events and services. The ICE provides general system-wide user interface capabilities and allows users to configure the system. The EventChannel, ServiceBroker, and ICE components are grouped together behind a single class simply called *Isys* using the Facade pattern.[29] This class also uses the Singleton pattern[29] to restrict itself to one instance per running system. Its public interface supports the following capabilities:

- Registry of a component as a listener for a particular class of events (uses the EventChannel component)
- Broadcast of an event (uses the EventChannel component)
- Registry of a component as a provider of particular services (uses the ServiceBroker component)
- Request of services by type (uses the ServiceBroker component)
- Request of services applicable for a particular object, as demanded by the interactive discovery process (uses the ServiceBroker component)

The ICE serves as a manager for nonplatform components. Upon startup of the system, it discovers and instantiates installed components based on the presence of corresponding class files in a user-defined "class path" (it uses a custom ClassLoader object, which could be enhanced to load components over the Internet). The ICE provides user interface options (buttons and pull-down-menu options) to invoke those components defined as entry points to

Figure 5 Taxonomy of ISYS components



the system. In addition, the ICE can display information on the status of pending asynchronous operations (e.g., the execution of a long search on a remote server). Finally, it serves as a configuration tool for the installed components, allowing the user to select default service implementations (when multiple services of the same type are available).

The ICE is not shown in Figures 2, 3, and 4. It appears when the user starts up the system and presents the option that (in this example) allows him or her initially to invoke the SequenceViewer component. It is also active behind the scenes; as the instantiator of all of the service providers, it monitors the asynchronous requests for computation (i.e., the similarity search and the multiple sequence alignment).

The ISYS platform is highly general and essentially independent of the domain of bioinformatics—it could be used to integrate any components adhering to its conventions for exchanging events and services. We encourage other developers to make use of the platform for their own purposes. It is available for download (http://www.ncgr.org/research/isys) with extensive documentation.

Exchange of services. Exchange of services in ISYS depends on the ServiceBroker component, which uses a variation of the Broker pattern.[30] This pattern, well known for its role in CORBA,[31] allows service consumers to access service providers according to properties of the providers. A match between a consumer and a provider registered with the broker is made at run time. The pattern helps requesters' needs to be met even when the set of providers is variable and unpredictable. In Figure 4, four different service providers are shown: the *SequenceServerProxy*, *SequenceAnalysisServerProxy*, *SequenceViewerFactory*, and *SimilaritySearcherFactory*. Each of these registers one or more services upon instantiation. As discussed, these services can be discovered at run time according to various properties.

The ServiceBroker component in ISYS differs from the standard Broker pattern in three main ways. First, services are encapsulated as objects (using the Command pattern[29]) and represent the unit of exchange between consumers and providers. Services are atomic operations, analogous to single method calls. In the standard pattern, service providers adhering to specific interfaces are the unit of exchange, so that related services are bundled together. The focus in

ISYS on the single service reflects a kind of procedural philosophy by which data can be transformed or manipulated through any logical sequence of operations, regardless of which component performs those operations or how they are accomplished (see Discussion section). Such transformations are illustrated in the scenario of Figure 1, where the user proceeds from a DNA sequence, to a set of similar sequences, to a multiple sequence alignment.

Second, the ServiceBroker component in ISYS does not match distributed consumers and providers as in the standard pattern; instead, all concerned components are running locally in the client machine, as can be seen in Figure 4 (access to remote resources is handled via separate client-side proxies to servers, independently of the ServiceBroker component, e.g., the *SequenceServerProxy* and *SequenceAnalysisServerProxy* components). Therefore, we have no need for client- and server-side proxies that handle marshaling and unmarshaling of data and concern themselves with low-level network protocols.

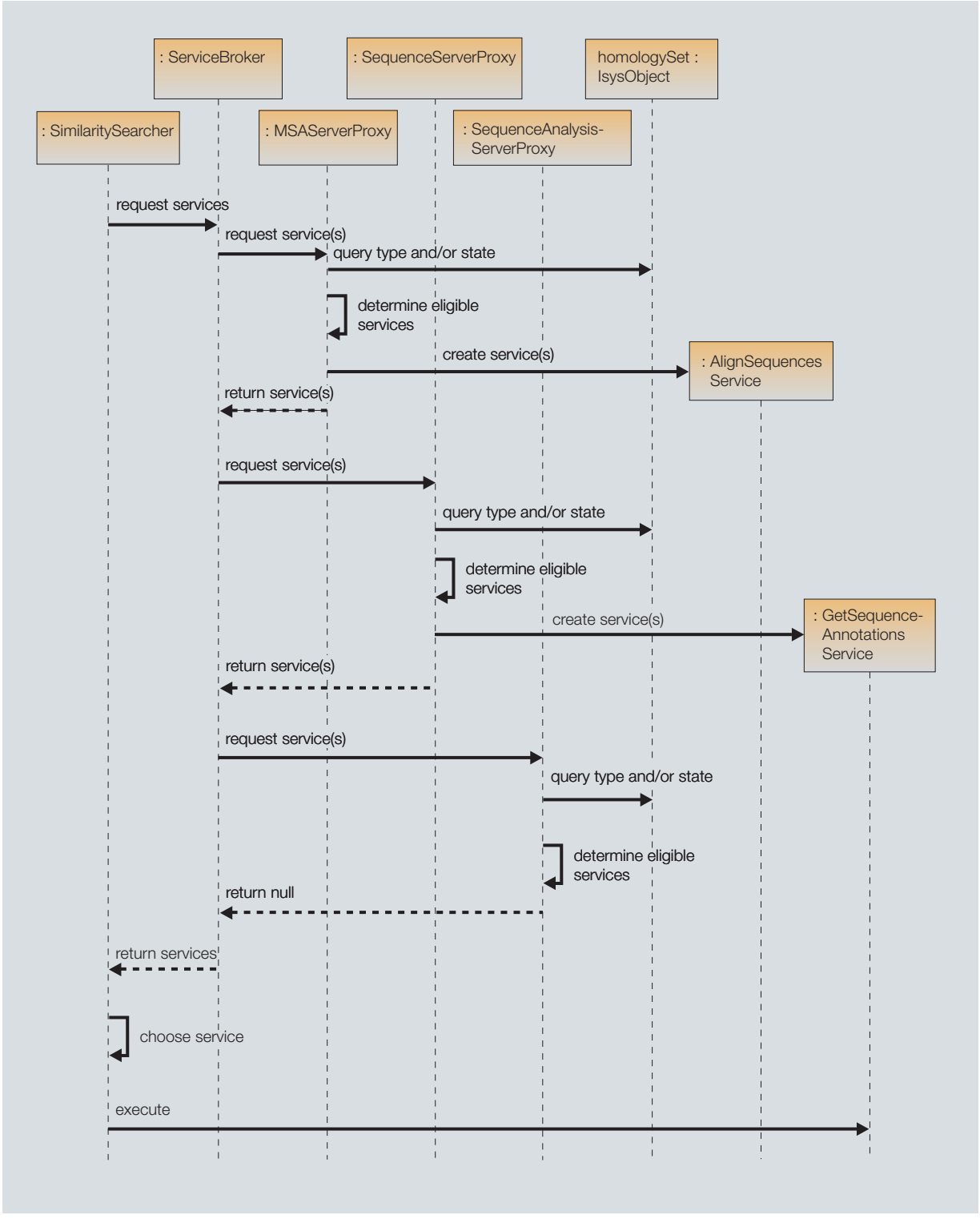
Third, ISYS supports two kinds of service requests: *direct discovery* and *interactive discovery*.[32] Direct discovery request handling is similar to the standard approach, in that requesters specify a desired property and are provided with a single match or informed that none exists. As described earlier, the subjects of requests in ISYS are services (i.e., procedures). In addition, direct discovery accommodates hierarchical relationships among services. Requesters specify a class of service instead of a name or property, and the ServiceBroker component may respond with either a registered service of that specific class or one of any subclass. This hierarchical service structure allows a requester to ask for a general class of service and to take advantage of specific implementations that may be added in the future. The configuration feature of ICE allows the user to define which specific, registered service is to be used for each class of service.

The request for full-length sequences behind the scenes in Figure 3 is an example of direct discovery. The alignment component requested an instance of *GetSequencesByIdService*, from which it knew it could obtain sequence text for specified sequence identifiers. This service, having been registered by the SequenceServerProxy component, was discovered by the ServiceBroker component, which called upon the SequenceServerProxy component to instantiate the service and then passed the instance to the requesting component. Multiple subclasses of *GetSequencesByIdService* could have been registered. For example, suppose one service were registered by an instance of *SequenceServerXProxy*, called *GetSequencesByIdFromXService*, and another registered by an instance of *SequenceServerYProxy*, called *GetSequencesByIdFromYService* (assume *X* and *Y* are different database resources). In this case, the ServiceBroker component would return an instance of the default of these two services (specified via the system configuration menu). As a result of this hierarchical mechanism, a requester can be more or less specific in its request; if less, the system will attempt to behave in a sensible way.

Interactive discovery, illustrated in Figure 2, is a two-step process by which a user is presented with a set of services suitable for a selected data object, then a service is invoked according to the user's selection. Interactive discovery is more complicated than direct discovery, not only because it involves user interaction but also because service providers must evaluate a selected data object before deciding whether they can operate on it.

Figure 6 is a sequence diagram describing interactive discovery in the context of Figure 2. When the user initiates a request, the SimilaritySearcher component requests eligible services from the ServiceBroker component, sending along a reference to the selected data object, in this case an instance of *IsysObject* representing a set of sequence homologs (see the section on the data model), for evaluation by service providers. The ServiceBroker component in turn requests eligible services from all of its registered service providers (three are shown, belonging to classes *MSAServerProxy*, *SequenceServerProxy*, and *SequenceAnalysisServerProxy*). Some of these service providers will find that they have services that can operate on the data object, while others will not. Some will find that the data object is eligible for more than one of their services. In this case, the server proxy that performs multiple sequence alignments, the MSAServerProxy component, recognizes upon inspection of the IsysObject instance that it can apply its *AlignSequences* service to the object, so it creates and returns an instance of this service. Similarly, the SequenceServerProxy component creates and returns an instance of its *GetSequenceAnnotations* service. The SequenceAnalysisServerProxy component, however, does not know how to operate on the IsysObject instance, so it creates no service and returns a null value. The ServiceBroker component collects all returned instances of services and passes them back to the SimilaritySearcher component.

Figure 6 Sequence diagram illustrating interactive discovery



Each service has an attribute called *displayName* containing a value that can be shown to users. The SimilaritySearcher component arranges all such names in a pop-up menu, and when the user selects one, executes the corresponding service on the original IsysObject instance. In this case, the service spawns an instance of a component that serves as a front-end to CLUSTAL W.

Exchange of events. Events are exchanged using the EventChannel component, which implements the Event Channel pattern.[30] The Event Channel pattern is a variant of the standard Observer pattern that allows components to exchange events without registering directly with one another. It is similar to the Broker pattern in that it uses a mediator to avoid direct dependencies among components, but is different in that it allows components to interact by a "push" mechanism rather than a "pull" one—that is, with the Broker pattern the active partner is the requester, whereas with the Event Channel pattern the active partner is the provider.

Event exchange in ISYS follows the Event Channel pattern. First, receiving components register listeners with the EventChannel component. Then when broadcasting components "fire" events, the EventChannel component calls all registered listeners, which prescribe appropriate reactions. Events in ISYS are divided into different types, only one of which is of interest to each listener. Each event is linked to a data object (see the section on the data model), which represents the subject of the action that the event describes. In addition, for event exchange to be active, two components must be explicitly synchronized. This prevents undesired synchronization, which can be confusing to the user and expensive in terms of performance. Components are generally synchronized when one spawns another and passes it a starting data set.

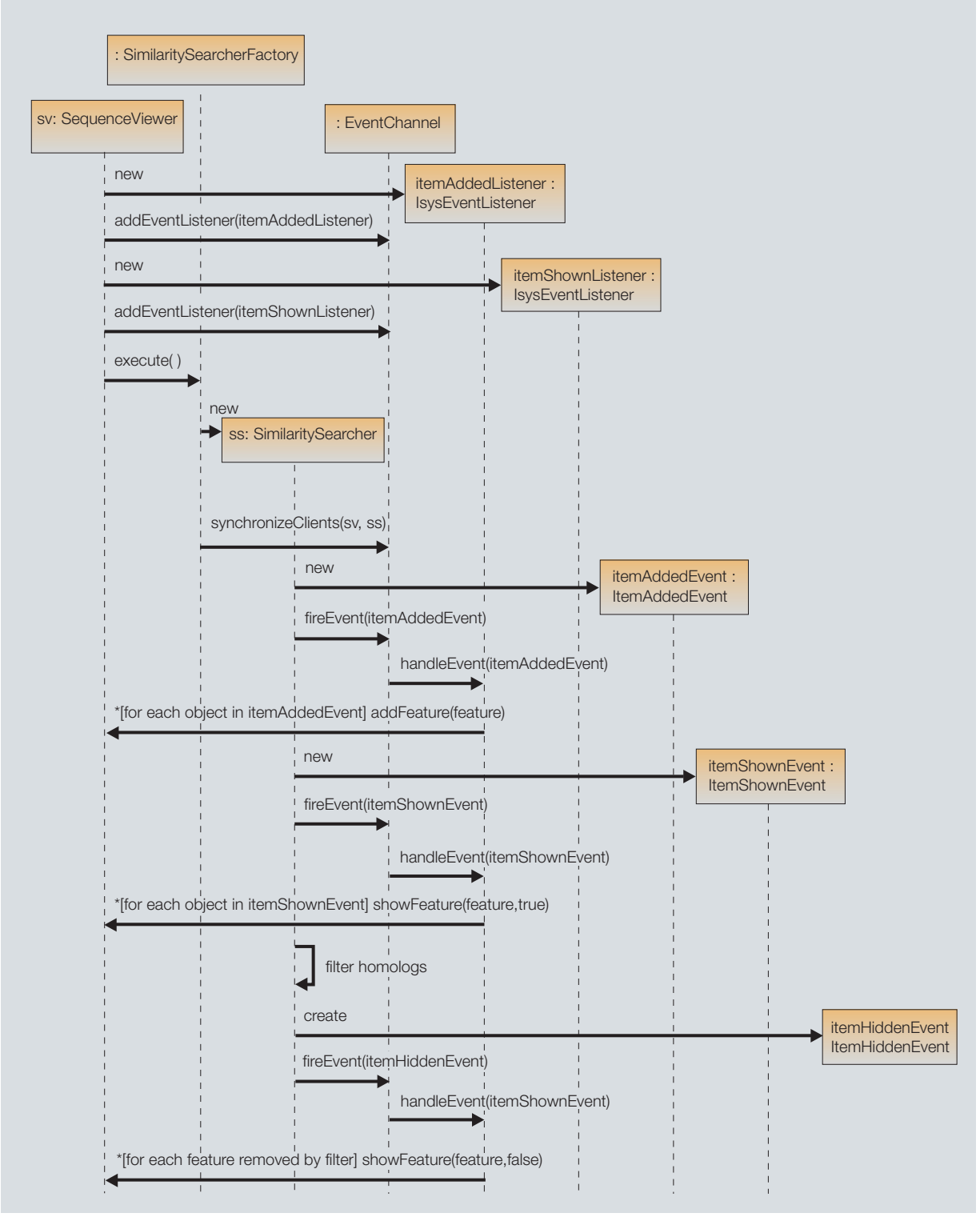
Consider the scenario illustrated in Figure 2, in which the SequenceViewer component and the SimilaritySearcher component exchange events. Even before the SequenceViewer component had spawned the SimilaritySearcher component, it had registered listeners with the EventChannel component. After the SimilaritySearcher component was invoked, the search executed, and the results returned, the SimilaritySearcher component fired events describing the addition and display of the search hits. The EventChannel component called the corresponding listeners on the SequenceViewer component, passing them the events, and the listeners accomplished the addition and display of corresponding annotation bars in the SequenceViewer component. The listeners acquired the information they needed about the hits by examining the data objects linked to the events.

Next, when the user selected search hits in the SimilaritySearcher display, the SimilaritySearcher component fired events describing the selection, the EventChannel component called the appropriate listeners on the SequenceViewer component, and the listeners caused the corresponding graphical bars to become selected. When the user filtered the search hits by expect value (inset of Figure 2), the SimilaritySearcher component fired appropriate events for all of the hits that became invisible, and the SequenceViewer component reacted by hiding the corresponding graphical bars. Note that these components can exchange events in the reverse direction (e.g., to effect selection in the SimilaritySearcher display corresponding to mouse clicks from the SequenceViewer display), but only because the SequenceViewer component knows how to broadcast them and the SimilaritySearcher component knows how to receive them. Event exchange is not automatically bidirectional.

Figure 7 is a sequence diagram describing part of the exchange between the SequenceViewer and SimilaritySearcher components. It shows the SequenceViewer component creating and registering listeners for instances of *ItemAddedEvent* and *ItemShownEvent*, then invoking the SimilaritySearcher component (via a service). Note the explicit call to the method *synchronizeClients*. After the SimilaritySearcher component receives its search results (not shown), it broadcasts events describing the addition of new search hits to its model and to the graphical display. The EventChannel component calls the appropriate listeners on the SequenceViewer component, and the listeners cause corresponding graphical annotations to be created and displayed. Next, the user filters the search results, and the SimilaritySearcher component broadcasts an event describing the disappearance of eliminated hits. Again, the EventChannel component calls the appropriate listener, which this time causes the corresponding bars to become invisible.

Figures 6 and 7 should help to make clear that, despite involving separate components, these scenarios take place within a single process space, and may involve only a single thread of execution. With the exception of auxiliary servers (see next section), all

Figure 7 Sequence diagram illustrating exchange of events



ISYS components currently run within the same Java virtual machine.[33] In the scenario just described, the code for the EventChannel component and the listeners in the SequenceViewer component was executed by the same thread that handled the initial user-interface actions in the SimilaritySearcher component. This is the way the Observer pattern is typically implemented, but it has important implications when multiple components are involved. For example, ISYS components are interdependent in terms of threading and user-interface performance. If one component has a slow, computationally intensive listener, and another component fires events without spawning new threads, the first component can render the second unresponsive. This behavior is contrary to the basic principle of independence of components, and we are working to alleviate it in two ways: by having the platform handle threading itself where appropriate, and by incorporating rules for threading and performance into the standards that each component must meet. On the other hand, sharing a process space means that sending data from one component to another is inexpensive, because it can be accomplished simply by passing references. The EventChannel component in particular cannot become a performance bottleneck (at least with proper threading) because it is not actually a conduit through which data must be passed, but simply an object whose methods are called as needed by interested components.

How components fit into the system. Besides the platform and external servers, all ISYS components implement at least one of two interfaces: *Client* and *ServiceProvider* (see Figure 5). Implementers of Client may broadcast and receive events, and implementers of ServiceProvider may register and provide services. Any component may request a service. Components may implement both interfaces but generally do not.

By convention, ServiceProvider implementations fall into two classes: server proxies and client factories. Server proxies are client-side representatives of server resources that insulate other components from changes to such resources, and that allow equivalent server-side operations to be interchanged (i.e., server proxies break down the capabilities of a server into its constituent services, each of which can be interchanged with equivalent services). For example, one might define server proxy components for two Internet resources that both perform similarity searches of DNA sequences against public databases—call them *SequenceAnalysisServerProxyA* and *SequenceAnalysisServerProxyB*. Each of these server proxies could provide a particular type of service—call it *SimilaritySearch*—and either one (or both) could be installed and could register as a provider of that service. No matter which was installed, the service would be discoverable by the direct discovery process (if both services were registered, the system would provide requesters with the user-defined default), and would perform the same abstract task—although the two servers might differ in terms of performance, search algorithm, or background database. In addition, changes to the interface of either Internet resource would require alteration in ISYS only to the corresponding server proxy.

Client factories register and provide services that instantiate clients. For example, the SequenceViewer component in Figure 2 has a corresponding factory, *SequenceViewerFactory* (see Figure 4). The factory registers as a provider of a service to display sequence annotation. In fulfillment of each request for this service, the factory instantiates a SequenceViewer component.

Servers are essential for the system but auxiliary (see Figure 5) in the sense that they do not interact directly with the platform—they are always accessed via server proxies. Any resource from which a server proxy can obtain data can function as a server. Such resources may exist locally or remotely and may be called using whatever mechanism is most convenient for the server proxy (e.g., TCP/IP [Transmission Control Protocol/Internet Protocol] sockets, HTTP [HyperText Transfer Protocol], CORBA, Java RMI [Remote Method Invocation], and JDBC** [Java Database Connectivity]). Although they are not servers in the conventional sense, utility libraries or executable programs that are called directly by server proxies may function as servers in the sense of auxiliary data providers.

Existing software tools are typically incorporated into the system as servers or clients. Tools that provide data (e.g., analysis tools or databases) are added as servers, and corresponding server proxies are created. The server proxies must register as providers of services that represent in abstract terms the essential functionality of the servers. Methods of these services responsible for their execution must call data providers appropriately and repackage their output in a form recognizable by other components (see the section on the data model). Tools that provide user interfaces are added as clients. Corresponding client factories are created to register as providers of

services representing the functionality of the user interface tools. These services instantiate the tools when invoked. The tools themselves are generally wrapped with simple, lightweight Java classes that implement the Client interface. These classes handle ISYS-level interaction by delegating calls to the original tools in terms the tools can understand, and intercepting intracomponent user-interface events of interest to other ISYS components, for which they broadcast corresponding ISYS events. In general, separately developed tools can become part of ISYS with relatively little alteration and minimal compromise of autonomy.

Various levels of integration are possible with clients, and there is some tension between integration and autonomy. In the simplest case, a previously developed user-interface tool can be invoked from ISYS (usually with a data set passed from another component) but does not synchronize with other components or perform interactive discovery. The abilities to listen and respond to events, to fire events, and to enable the interactive discovery process can each be added independently. Listening and responding to events requires a wrapper to register as a listener with the EventChannel component and delegate appropriate calls to the original tool.

Often the tool does not provide an interface that allows the wrapper easily to alter the selection or visibility of individual graphical objects (for example), so some invasion of its internal mechanisms is necessary (leading to dependencies between the wrapper and the internals of the tool). If the client is to fire ISYS-level events, the wrapper must intercept in some way the GUI (graphical user interface) events in the existing tool, then create and broadcast corresponding ISYS events. If the code for the tool is available, simple hooks for the wrapper can be added to the appropriate internal listeners. These hooks can easily be written so that the tool can run with or without ISYS. This approach, however, does result in further dependency between the tool and the wrapper, and hooks may need to be added again when new versions of the tool replace older ones. If neither the code nor an adequate programming interface is available, events must be intercepted at a higher level (e.g., that of the windowing environment). This will be more difficult but should be possible (we have not had the need to integrate such a client).

Data model. A common standard for representation of data is necessary to allow components to exchange information. If components are to interoperate, both service providers and event listeners must be able to recognize and interpret the data that other components pass to them. In many integration solutions, this need is addressed with a canonical schema, to which all components must be able to convert their internal representations.[11,34] We have avoided such a schema, however, because it would compromise the flexibility of the system. A shared schema represents an implicit interdependency among components, since changes to the global schema required by one component can necessitate changes to many others; and in bioinformatics, the prospect of a standard, stable schema is unlikely.[35,36] In addition, it is unrealistic to expect every component to represent all data at the same level of detail, since data of primary interest to one component often are peripheral to others. Therefore, even if a global schema is used, it must be possible to "fill out" its data structures to various extents.

Instead of using a canonical schema, we have chosen an approach that allows each component to access only the information essential for its needs. In general, components are interested in only a few attributes of the objects they receive from other components. Additional schematic details are encapsulated within components, so that integration can be achieved with minimal interdependency. We describe this approach in detail, first by describing an early design that we eventually dismissed, and then by introducing an improved solution.

Our first design used a set of Java interfaces to define a high-level data model that represented a kind of common denominator for all components.[37] Each interface characterized a class, with accessor methods defining attributes and links to associated classes. Components implemented these interfaces with classes that could allow considerably richer representations of data. For example, Figure 8 shows the definition of one such interface, *IsysSequence*, and a class that implements that interface, *SequenceViewerSequence*. IsysSequence provides methods signifying the attributes and associations believed to be most essential for integrative purposes. SequenceViewerSequence defines a more detailed representation of a sequence, appropriate for use by the SequenceViewer component. It implements the IsysSequence interface, and fulfills its methods by returning selected aspects of the richer representation. This allows an object of type SequenceViewerSequence to be passed to another component and

Figure 8 Implementation of SequenceViewerSequence interface using old data model

```
public interface IsysSequence
{
  String getICAccession() { ... }
  String getSequenceText() { ... }
  IsysTaxon getTaxon() { ... }
}

public class SequenceViewerSequence implements IsysSequence
{
    // IsysSequence methods
    public String getICAccession() { ... }
    public String getSequenceText() { ... }
    public IsysTaxon getTaxon() { ... }

    // Methods specific to SequenceViewerSequence
    public int getSequenceLength() { ... }
    public Vector getFeatures() { ... }
    public Vector getComments() { ... }
    public Vector getReferences() { ... }
    public Source getSource() { ... }
}
```

interpreted as being of type IsysSequence. The second component can access essential data from the object but does not need to know about unnecessary details of the SequenceViewerSequence extensions. In addition, its code can be dependent only on the relatively stable IsysSequence class, not on SequenceViewerSequence (which may need to be changed as SequenceViewer evolves).

This design was appealing in its simplicity and took advantage of Java's interface mechanism to allow the same object in memory to serve as both the detailed and common-denominator representations of the data object. It had several disadvantages, however. Most importantly, we found that it was difficult to determine a common denominator of attributes for each class. Rather than all components sharing interest in a core set of attributes, different components seemed to require different "views" on the same object that were only partially overlapping or completely disjoint. Furthermore, no matter what common denominator of attributes one chose, some data providers were unable to supply elements of it. Consequently, data consumers could not reliably expect core attributes to be present (in their absence, a null value was returned). In addition, the data model was "hard coded" into the interface definitions, so changes required recompilation and posed problems for already-deployed software.

Finally, the design was clumsy for handling associations of objects. In many cases, an object needs to be interpreted as equivalent to a closely associated object. For example, if a listener receives an event that references an IsysSequence object but the listener is concerned with taxonomy, it should still respond appropriately because an IsysSequence object "has" an IsysTaxon object. In this design, the receiver handled such associations by traversing the network of objects (in this case by retrieving the related IsysTaxon object, then inspecting it), using helper interfaces that defined close associations among classes. This complicated the code of data consumers, forcing them to unravel object relationships that would be better kept transparent.

Our solution was to redesign the implementation scheme for the data model to use generic objects having collections of attributes established at run time. In the new scheme, there are no classes or interfaces representing types of domain objects, such as IsysSequence. There are only generic objects, of a type

Figure 9 Definition of IsysAttribute and IsysObject interfaces and sample attributes

```
// A marker interface
public interface IsysAttribute {}

public interface IsysObject {
    IsysAttribute getAttribute( Class attrClass );
    HashMap getAttributeGroup( Class[] attrClasses );
    Collection getAttributeCollection( Class attrClass );
    Collection getAttributeGroupCollection( Class[] attrClasses );
}

// sample attributes

public interface IsysICAccessionAttribute extends IsysAttribute {
    String getICAccession();
}

public interface IsysSequenceTextAttribute extends IsysAttribute {
    String getSequenceText();
}

public interface IsysTaxonNameAttribute extends IsysAttribute {
    String getTaxonName();
}
```

called *IsysObject*, each of which has a collection of attributes (which are specified by Java interfaces). The IsysObject type supports queries for particular attributes or groups of attributes. The set of attributes belonging to each object is established upon instantiation of the object.[38] In the new scheme, the individual attribute is the atomic unit, rather than the domain object with fixed attributes. Data consumers simply query IsysObject instances to see if they have the attributes needed to perform a task and, if they do, the consumers request the values of those attributes. Because consumers no longer depend on a fixed "contract" (i.e., interface) published by data providers, they do not need to be recompiled when the data model changes (providers do need to be recompiled if they are to provide new attributes). Thus, the dependency of consumers on providers is greatly diminished.

Figure 9 shows the definition of IsysObject, which is accomplished with a Java interface. The figure also shows the IsysAttribute interface (simply a marker) and extensions of it defining a few sample attributes. Note that the IsysObject interface provides several different methods for retrieving attributes: *getAttributeGroup* retrieves a set of closely bound attributes;[39] *getAttributeCollection* and *getAttributeGroupCollection* are "Collection" analogs of the single-item *getAttribute* and *getAttributeGroup* methods. The interfaces defining sample attributes must be implemented by an actual data object (see below).

Figure 10 shows an implementation of *SequenceViewerSequence* using IsysObject. The new version implements the IsysObject interface and the interfaces of several ISYS-level attributes. Attributes expected to be of interest to other components are accessible via the methods of the IsysObject interface. The *getAttribute* method illustrates some of the value of representing attributes as types. For example, this representation, along with the use of Java's

Figure 10 SequenceViewerSequence class implemented using IsysObject interface

```
public class SequenceViewerSequence
    implements IsysObject,
               IsysICAccessionAttribute,
               IsysSequenceTextAttribute,
               IsysTaxonNameAttribute
{
    // attribute methods
    public String getICAccession() { ... }
    public String getSequenceText() { ... }
    public String getTaxonName() { return getTaxon().getName() };

    // methods specific to SequenceViewerSequence
    public int getSequenceLength() { ... }
    public Vector getFeatures() { ... }
    public Taxon getTaxon() { ... }
    public Vector getComments() { ... }
    public Vector getReferences() { ... }
    public Source getSource() { ... }

    // IsysObject methods
    public IsysAttribute getAttribute( Class attrClass ) {
      if ( attrClass.isAssignableFrom( IsysICAccessionAttribute.class ) ||
           attrClass.isAssignableFrom( IsysSequenceTextAttribute.class ) ||
           attrClass.isAssignableFrom( IsysTaxonNameAttribute.class ) ) {
          return this;
      }
    }

    public HashMap getAttributeGroup( Class[] attrClasses )  {
      HashMap result = new HashMap();
      for ( int i = 0; i < attrClasses.length; ++i )  {
          IsysAttribute attr = getAttribute( attrClasses[i] );
          if ( attr == null ) return null;
          result.put( attrClasses[i], attr );
      }
      return result;
    }

    // attribute collection retrieval
    Collection getAttributeCollection( Class attrClass ) { ... }
    Collection getAttributeGroupCollection( Class[] attrClasses ) { ... }

}
```

Class.isAssignableFrom( ) method, allows for hierarchical definitions of attributes. The getAttribute method will respond appropriately not only to requests for any of the ISYS-level attributes assigned to SequenceViewerSequence, but to requests for "parents" of those attributes (in terms of interface extension). Figure 10 does not show the implementations of the attribute collection retrieval methods, for simplicity. They are straightforward generalizations of the implementations that are shown.

By focusing on specific attributes, which ultimately are what data consumers need, the new approach avoids the problems of standardizing the data model at a global level. Different consumers can have different views of the same object, by requesting different sets of attributes. Data producers can give an object as many or as few attributes as they have at their disposal, without being constrained by an explicit contract. Data consumers are forced to address explicitly the issue that there is no guarantee any attribute will or will not be present in a particular object (rather than being presented with the illusion of a guarantee, as in the old system). In fact, the new approach serves as a method for implementation, but defines no particular schema—even a high-level one like that defined in the previous approach. The schema is implicit in the attributes of and relationships among objects, as defined by component developers.[40]

In many cases, a single IsysObject instance represents a "flattened" view of an entire network of related objects—i.e., the IsysObject instance appears to have all of the attributes of the network. The implementer of IsysObject is responsible for traversing the network as necessary. In this way, data producers, not consumers, bear the burden of establishing associations. If this design is more complicated than the original for data providers, it is simpler for consumers. A default implementation of IsysObject helps to protect developers from some of its complexities.

Observe the handling of the *TaxonName* attribute in Figure 10. This attribute is actually stored in a separate class (Taxon) associated with SequenceViewerSequence. SequenceViewerSequence, however, declares it as an ISYS-level attribute, and retrieves it from the associated Taxon object on demand. In this way, the data consumer is protected from having to navigate a complex network of objects.

Discussion

Although it makes use of standard techniques for loose coupling of components (e.g., service and event exchange), ISYS represents a somewhat unusual approach to the problem of software integration, especially in the field of bioinformatics. Two attributes in particular distinguish the system. First, it avoids the use of a canonical schema and instead relies on a loose, highly flexible, and dynamic model for data representation that further reduces dependencies among components. Second, it depends on no top-level controller to coordinate the integration of components. Instead, ISYS components are relatively independent and autonomous, and the behavior of the system emerges from their interactions. Components are instantiated but not controlled by the ICE, and they use the *ServiceBroker* and *EventChannel* components only as media for interaction. Because each component interacts with the others according to a highly general set of rules (defined by event broadcasters and listeners, requests for and responses to the interactive discovery process, and the definition of the data model by assignment of attributes to IsysObject instances), components can be added or subtracted freely. As the set of active components is altered, the nature of the system changes, while still remaining coherent.

This property of the system to support easy "plug and play" of components makes it well suited for the rapid creation of integrated systems of legacy components. Users can create such systems by installing and activating any combination of properly wrapped components. These systems may not be as perfectly integrated, and will not be as unified, as if they were developed "from scratch." In a fast-moving field with limited development resources, however, the flexibility and speed gained by building systems out of existing tools is worth the price of diminished unity, for many applications.

ISYS can also be useful as a prototyping environment for new components. With a small amount of work, a new application can be fit to plug into the system. The developer can then derive the benefits of access to the other components in the system, for example, as providers of data or tools for visualization and analysis of biological data. This can allow more focus on the specific task the new component is intended to address.

The ISYS platform can be useful in any situation in which developers have a strong incentive to allow client-side integration of heterogeneous software components. The platform is best at allowing graphical synchronization, transparent data retrieval, and probing, exploratory navigation from component to component (using the interactive discovery process). It can be used easily to integrate command-line executables and stand-alone servers, and relatively easily to integrate GUIs with available source code (or an adequate API [application programming interface]). GUIs without source code or an adequate API are more difficult to incorporate, but could (we believe) be accommodated.

The mechanism for representing shared data is independent of the bioinformatics domain, but it may not be appropriate for all applications. It is designed with the assumption that different components will occupy largely orthogonal data spaces, which intersect at only a few attributes. For example, the SimilaritySearcher and SequenceViewer components are mostly concerned with different data—the former with the particulars of similarity scoring and local sequence alignment, the latter with sequence annotations and their physical locations—and they share interest only in the identities of sequences and the locations of search hits. The ISYS platform is not effective for allowing components to share large portions of data models (it discourages such sharing to reduce component interdependency). If it were necessary for multiple components, for example, to share knowledge of many of the particulars of the data representation used by the SequenceViewer component, they should probably interact directly, without the use of the ISYS platform. In such cases, it may make sense to consider multiple components as one with respect to ISYS (i.e., to give them a single point of contact with the platform).

The ISYS prototype can be used to navigate, from a genomic map to sequences on that map, then to their metabolic pathways.

In bioinformatics, we think ISYS is well suited to meet the need for flexible data and software integration in a visually intuitive client-side environment. Although the use case described in this paper was biologically simple, ISYS supports more interesting applications. It can allow sweeping navigation across species, biological data types (e.g., maps, sequences, and pathways), and biological perspectives (e.g., structural, functional, and evolutionary). For example, it is possible using the ISYS prototype to navigate from a genomic map (a high-level structural view of a genome) to sequences placed on that map to metabolic pathways related to these sequences (a high-level functional view).[37] With additional refinement and new components, we think ISYS can become a valuable instrument for comparative and functional genomics. Because it is built from the bottom up, it will continually be able to exploit the work of the independent researchers who drive this exciting scientific field.

We think it is worth reflecting on the design of the system by exploring two questions: (1) *Is our design object-oriented*? (2) *Is it consistent with the Model-View-Controller* (*MVC*) *pattern*? The question of object orientation is interesting because although at first glance ISYS seems purely object-oriented—object-oriented design is used throughout and the platform is implemented with an object-oriented language—at a deeper level it is strongly procedural. The services are essentially coarse-grained procedures, which remain clearly separated from the data on which they operate. Thus, data and behavior—whose combination is arguably a *sine qua non* of object orientation—are kept separate. Like the Command pattern, ISYS uses object-oriented design to accomplish something inherently procedural.

This procedural approach ultimately serves to promote flexibility. The set of behaviors must be able to change as components are plugged into and removed from the system. It makes sense to separate data and behavior because they are not bundled together neatly by the existing tools that we need to integrate. Instead, many different tools support operations for essentially the same data types. For example, there are hundreds of different tools that perform various operations on sequences. The user is typically in the position of performing multiple operations on a single data object or set of related data objects. By separating data objects from operations, we allow the set of operations to change freely, and at run time we can pair data objects with eligible operations (as with the interactive discovery process).

Similarly, the answer to the question of the Model-View-Controller pattern is both yes and no. Individual components can (and often do) use the MVC pattern internally, but multiple components do not share a single model. Instead, components exchange information about specific portions of their models using the common protocol of the data model. The reason, once again, is flexibility. Sharing a single model would require an unacceptable level of component interdependency, in the same way as would adhering to a canonical schema (discussed earlier). By communicating only essential information about their models through the exchange of events and services, components are able to interoperate without having any more interdependency than absolutely necessary.

Currently the ISYS team is focusing on development of several new components, refinement of the platform, and coordination with efforts at NCGR (National Center for Genome Resources) in metabolic pathways, gene expression, and genomic maps. For more information about the project, and to download the latest release of the platform, see http://www.ncgr.org/research/isys.

Acknowledgments

We are indebted to former ISYS team members J. J. Zhuang, David Hanley, and Tom Cartner for their contributions to design and development; to Pedro Mendes and Allan Dickerman for their valuable insight on biological use cases; to Bruno Sobral for helping to make the project possible; and to the National Center for Genome Resources for funding and support.

Cited references and notes

1. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, NJ (1996).
2. "Comparative genomics" is a broad term used here in reference to various techniques for learning about one organism's genome through comparison to a better-studied relative.
3. K. Sirotkin, "NCBI: Integrated Data for Molecular Biology Research," *Bioinformatics: Databases and Systems*, S. Letovsky, Editor, Kluwer Academic Publishers, Norwell, MA (1999), pp. 11–19.
4. R. Unwin, J. Fenton, M. Whitsitt, C. Jamison, M. Stupar, E. Jakobsson, and S. Subramaniam, "Biology Workbench: A Computing and Analysis Environment for the Biological Sciences," *Bioinformatics: Databases and Systems*, S. Letovsky, Editor, Kluwer Academic Publishers, Norwell, MA (1999), pp. 233–244.
5. T. Etzold, A. Ulyanov, and P. Argos, "SRS: Information Retrieval System for Molecular Biology Data Banks," *Methods in Enzymology* **266**, 114–128 (1996).
6. R. F. Smith, B. A. Wiese, M. K. Wojzynski, D. B. Davison, and K. C. Worley, "BCM Search Launcher—An Integrated Interface to Molecular Biology Data Base Search and Analysis Services," *Genome Research* **6**, No. 5, 454–462 (1996).
7. Entigen Corporation (formerly eBioinformatics, Inc., and Empatheon, Inc.), http://www.ebionavigator.com.
8. DoubleTwist, Inc., http://www.doubletwist.com.
9. V. M. Markowitz, I. M. A. Chen, A. Kosky, and E. Szeto, "OPM: Object-Protocol Model Data Management Tools '97," *Bioinformatics: Databases and Systems*, S. Letovsky, Editor, Kluwer Academic Publishers, Norwell, MA (1999), pp. 187–199.
10. S. B. Davidson, O. P. Buneman, J. Crabtree, V. Tannen, G. C. Overton, and L. Wong, "BioKleisli: Integrating Biomedical Data and Analysis Packages," *Bioinformatics: Databases and Systems*, S. Letovsky, Editor, Kluwer Academic Publishers, Norwell, MA (1999), pp. 201–211.
11. P. G. Baker, A. Brass, S. Bechhofer, C. Goble, N. Paton, and R. Stevens, "TAMBIS: Transparent Access to Multiple Bioinformatics Information Sources," *Proceedings, Sixth International Conference on Intelligent Systems for Molecular Biology*, Montreal, Canada (June 28–July 1, 1998), pp. 25–34.
12. Mapping one database schema to another, even with the aid of human-defined meta-data, is an extremely difficult problem, unless the semantics and syntactic conventions of both schemas are formally defined and rigidly obeyed (which is generally not true of bioinformatics databases). This is one of the main motivations for the "looser" data model we have used.
13. We elaborate on this claim in: A. Siepel, A. Farmer, A. Tolopko, M. Zhuang, P. Mendes, W. Beavis, and B. Sobral, "ISYS: A Decentralized, Component-Based Approach to the Integration of Heterogeneous Bioinformatics Resources," *Bioinformatics* **17**, No. 1, 83–94 (2001). Note, however, that some query systems for heterogeneous databases have created advanced graphical user interfaces that do support exploratory querying to a degree. Examples are GeneLogic, Inc.'s Object-Protocol Model (see Reference 9) and IBM's PESTO (M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams, "PESTO: An Integrated Query/Browser for Object Databases," *Proceedings 22nd International Conference on Very Large Databases* (*VLDB*), Bombay, India (September 3–6, 1996), pp. 203–214.
14. Life Sciences Research Task Force of the Object Management Group, http://www.omg.org/homepages/lsr.
15. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley Longman Limited, Essex, England (1998).
16. K. Jungfer, G. Cameron, and T. Flores, "EBI: CORBA and the EBI Databases," *Bioinformatics: Databases and Systems*, S. Letovsky, Editor, Kluwer Academic Publishers, Norwell, MA (1999), pp. 245–254.
17. NetGenics, Inc., http://www.netgenics.com.
18. Synomics, Ltd., http://www.synomics.com.
19. J. Hu, C. Mungall, D. Nicholson, and A. L. Archibald, "Design and Implementation of a CORBA-Based Genome Mapping System Prototype," *Bioinformatics* **14**, No. 2, 112–120 (1998).
20. H. R. Pagels, *Dreams of Reason: The Computer and the Rise of the Sciences of Complexity*, Simon and Schuster, New York (1988).
21. S. Levy, *Artificial Life: A Report from the Frontier Where Computers Meet Biology*, Random House, New York (1992).
22. In contrast to the elements of cellular automata and Boolean networks, the components in ISYS are not necessarily simple; but they encapsulate their complexity and interact with one another in simple ways.
23. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. L. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology* **215**, No. 3, 403–410 (1990).
24. J. D. Thompson, D. G. Higgins, and T. J. Gibson, "CLUSTAL W: Improving the Sensitivity of Progressive Multiple Sequence Alignment through Sequence Weighting, Position-Specific Gap Penalties and Weight Matrix Choice," *Nucleic Acids Research* **22**, 4673–4680 (1994).
25. JalView, http://www2.ebi.ac.uk/~michele/jalview/.
26. M. P. Skupski, M. Booker, A. Farmer, M. Harpold, W. Huang, J. Inman, D. Kiphart, C. Kodira, S. Root, F. Schilkey, J. Schwertfeger, A. Siepel, D. Stamper, N. Thayer, R. Thompson, J. Wortman, M. Zhuang, and C. Harger, "The Genome

Sequence DataBase: Toward an Integrated Functional Genomics Resource," *Nucleic Acids Research* **27**, 35–38 (1999).
27. This capability is perhaps more powerful than the example here suggests. A slightly better illustration appears in Reference 13.
28. The EventChannel and ServiceBroker components represent a division into two parts of the component called the *ClientBus* in Reference 13, Siepel et al. (2000).
29. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, MA (1995).
30. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A System of Patterns: Pattern-Oriented Software Architecture*, John Wiley & Sons Ltd., Chichester, England (1996).
31. *CORBA: Common Request Broker Architecture and Specification*, available from the Object Management Group (1996). See http://www.omg.org/ for contact information.
32. The latter is called *dynamic discovery* in Reference 13.
33. Clement Szyperski defines components as "XXX." We have taken the liberty of allowing that while they may be independently *executable*, they need not be independently *executing*. In fact, ISYS components are JAR files that may have, but are not required to have, classes with *main* methods that allow execution with or without the ISYS platform.
34. P. Karp, "A Strategy for Database Interoperation," *Journal of Computational Biology* **2**, No. 4, 573–586 (1995).
35. V. M. Markowitz and O. Ritter, "Characterizing Heterogeneous Molecular Biology Database Systems," *Journal of Computational Biology* **2**, No. 4, 547–556 (1995).
36. S. B. Davidson, C. Overton, and P. Buneman, "Challenges in Integrating Biological Data Sources," *Journal of Computational Biology* **2**, No. 4, 557–572 (1995).
37. See Reference 13, Siepel et al.
38. Although the IsysObject interface does not explicitly support this capability, it is possible to implement an IsysObject such that new attributes can be added later in its life. In this way, an object can gain attributes as new data are produced by the system.
39. Space does not allow a full discussion of attribute "groups." Essentially, they are necessary to maintain a sense of which attributes describe the same entities when networks of objects are "flattened" (as discussed later in the text). See the ISYS Web site for additional information.
40. A full treatment of the issue of the data model is not possible here. As they create their components, developers will be able to contribute attributes to the data model, if it does not meet their needs. Thus, the model for object attributes and relationships will evolve according to the needs of individual component developers.

Accepted for publication January 8, 2001.

Adam C. Siepel *National Center for Genome Resources, 2935 Rodeo Park Drive East, Santa Fe, New Mexico 87505 (electronic mail: acs@ncgr.org).* Mr. Siepel received a B.S. degree in agricultural and biological engineering from Cornell University in 1994. He has worked in bioinformatics since 1995, first with the HIV database at Los Alamos National Laboratory and starting in 1996 at the National Center for Genome Resources (NCGR), where he began as a software developer. He recently moved from a role as a group leader for software development to become coprogram leader for NCGR's new integration program.

Andrew N. Tolopko *National Center for Genome Resources, 2935 Rodeo Park Drive East, Santa Fe, New Mexico 87505 (electronic mail: ant@ncgr.org).* Mr. Tolopko studied computer science at North Carolina State University. In 1994, he went on to serve as the Chief Technology Officer at LibraSoft, Inc., in Santa Fe, New Mexico, where he led the development of vertical-market application software. Mr. Tolopko joined the National Center for Genome Resources in 1999. He currently acts as a senior software developer in the Integration program, and one of the chief architects of the ISYS project.

Andrew D. Farmer *National Center for Genome Resources, 2935 Rodeo Park Drive East, Santa Fe, New Mexico 87505 (electronic mail: adf@ncgr.org).* Mr. Farmer received a B.A. degree in philosophy and mathematics from St. John's College, Santa Fe, in 1993. He went on to begin a career in bioinformatics as a research assistant with the HIV database at Los Alamos National Laboratory. Following a position at the Santa Fe Institute where he applied hidden Markov models to genetic sequence alignments, he took his current position at NCGR, where he develops databases and software systems in bioinformatics.

Peter A. Steadman *National Center for Genome Resources, 2935 Rodeo Park Drive East, Santa Fe, New Mexico 87505 (electronic mail: ps@ncgr.org).* Mr. Steadman received a B.S. degree in philosophy and mathematics from St. John's College, Santa Fe, in 1992, then studied physics at the University of Colorado at Boulder. Mr. Steadman developed novel radiation detectors at Los Alamos National Laboratory before coming to NCGR in 1997. He is currently a software developer for the Integration and Pathways programs.

Faye D. Schilkey *National Center for Genome Resources, 2935 Rodeo Park Drive East, Santa Fe, New Mexico 87505 (electronic mail: fds@ncgr.org).* Ms. Schilkey received her B.S. degree in computer engineering in 1986 from Oakland University in Rochester, Michigan. Before coming to NCGR, she worked for nine years in the areas of automotive engineering, real-time embedded software systems for military applications, and general system administration. She joined NCGR in 1996 as a systems administrator and later became a database administrator. Thereafter, she became a program leader for the Infrastructure program while maintaining her database administrator role. Her other technical interests include software development.

B. Dawn Perry *National Center for Genome Resources, 2935 Rodeo Park Drive East, Santa Fe, New Mexico 87505 (electronic mail: bdp@ncgr.org).* Ms. Perry received her M.A. degree in librarianship and information management from the University of Denver in 1983. She spent several years working in the computer-based training industry as a writer, instructor, tester, designer, configuration manager, librarian, and manager of groups of software engineers and training developers. She joined NCGR in 1997, where she is a knowledge management specialist focusing efforts on technical writing, intranet development, and maintaining a research library.

William D. Beavis *National Center for Genome Resources, 2935 Rodeo Park Drive East, Santa Fe, New Mexico 87505 (electronic mail: wdb@ncgr.org).* Prior to joining NCGR in 1998, Dr. Beavis spent 12 years developing both information systems and statistical methods for Pioneer Hi-Bred International. Since joining NCGR he has been promoted to Director of Science Programs and has directed the development, budgets, and staffing of pro-

grams in sequence processing, comparative mapping, gene expression, metabolic pathways, and computational biology. He also acts as coprogram leader for NCGR's Integration program. In addition to his administrative duties, he has maintained his interest in development of statistical methods for genetic research. His current activities include development of inferential analysis methods for m-RNA expression arrays (he is coauthor with Munneke et al. of "A Penalized Dissimilarity Measure and Permutation Confidence Measures for Gene Expression Cluster Analysis," in progress) and QTL (quantitative trait loci) identification (he is coauthor with R. C. Jansen of *Mapping Phenotypic Traits in Plant Breeding Populations on Basis of Haplotypes for Multiple Genetic Markers*, in press).