Component Interaction Patterns

Part of the Component Design Patterns *Project*

Philip Eskelin
Ernst & Young LLP
750 Seventh Avenue
New York, NY 10019
+1 (212) 733-7638
philip.eskelin@acm.org

SDHS photograph 2405e: Operators, Pacific Telephone and Telegraph Co., 1920s. Title Insurance Collection.

ABSTRACT

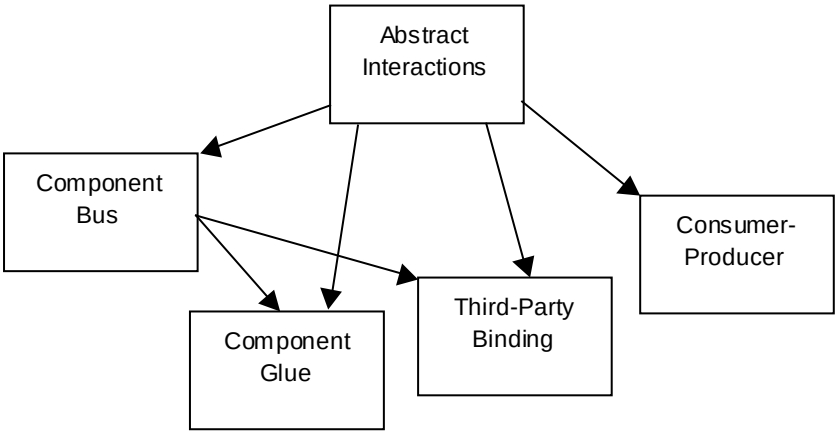
Many projects today use a component-based approach to developing software. Component-Based Development (CBD) stresses language and platform interoperability, and separation of interface from implementation. Existing and newly constructed components are being deployed to clients and servers to build flexible, reusable solutions.

However, assembling a system consisting of custom and pre-built components can be difficult because of hidden dependencies, complex interactions, and obscure design. This pattern language has ABSTRACT INTERACTIONS, COMPONENT BUS, COMPONENT GLUE, THIRD-PARTY BINDING, and CONSUMER-PRODUCER as five patterns that make it easier to assemble components that communicate, collaborate, and coordinate to get a job done..

INTRODUCTION

As with many other technology trends, industry analysts look into their oracles and promise that CBD is the big silver bullet. While it provides many benefits and can facilitate rapid delivery of successful solutions with a high return on investment, it's never a drop in the hat. Lack of solid project management, architecture, and design in CBD can be just as lethal as with any other technology panacea. Developing software in the context of CBD is not easy, but it can lead to highly successful results if done using proven techniques.

How do we make it easier for software developers to do it right? Let's take a step back and explore a historical trend we've seen in software development.



The Component Interaction Patterns Sub-Language

The following are thumbnails that describe each pattern in the language:

ABSTRACT INTERACTIONS

Reduce a component's dependence on its environment by defining interaction protocols between components separately from the components themselves. Specify these interactions in terms of abstract interfaces, and implement components to communicate with each other through them.

COMPONENT BUS

Bind components to an information bus that manages the routing of information between communicating components to remove explicit dependencies from the components themselves. Define interaction protocols that not only specify interfaces required for components to participate, but also the nature of interactions occurring between them.

COMPONENT GLUE

Create "glue" code to act as an adapter for incompatible components, or as a mediator between peers. Only build full-fledged components when glue doesn't meet all of your requirements.

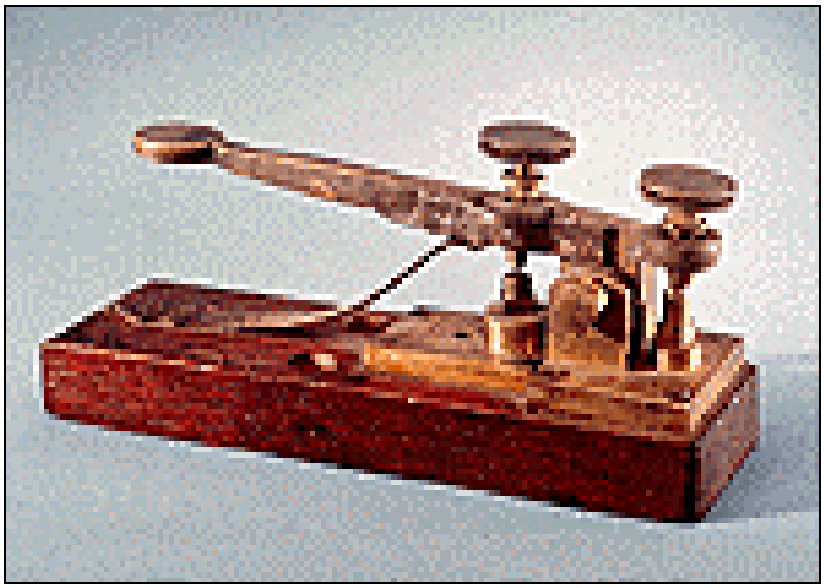
THIRD-PARTY BINDING

Remove connections established in the implementation of a component by having a third component bind two interacting components together.

CONSUMER-PRODUCER

Provide components a unified interface to multiple, seamless connectivity to heterogeneous service providers.

ABSTRACT INTERACTIONS**



Smithsonian Photo by Alfred Harrell

. . . BUY WITH CARE and BUILD FOR THE USER can result in building or reusing components that provide an abstract design for solutions to a family of related problems. Large-scale reuse with a LAYERED COMPONENT FRAMEWORK might be used to reduce development costs. But using these components and frameworks can be difficult. This pattern solves the problem of poorly designed interaction protocols existing between components.

❖ ❖ ❖

While separation of interface from implementation has many benefits, it can also become your worst nightmare. Poor design of interaction protocols between collaborating components can lead to unwanted behavior and internal dependencies that complicate system architecture.

A frustrating thing about acquiring custom off-the-shelf components is that vendors often don't adequately address component behavior and dependencies upon other components. User manuals, help files, or web pages will have an API form of documentation having sections like *method name*, *syntax*, *description*, and *return value*, but they don't truly deliver the essence of how they're used.

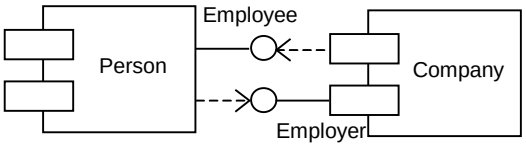
Name
action
Syntax
`boolean action (Event, What)`
Event: An Event object specifying the subject.
What: The Object object from which the event originates.
Description
Called when an action occurs in a component contained by the object.
Return Value
If successful, it returns true, else false.

Many times this is as good as it gets…

While it provides a straightforward format for documenting methods and properties, it doesn't address the bigger picture. It doesn't tell you how to interact with the component in question or what each possible action is, specify underlying behavior, or how it interacts with other components.

A component could call another component that in turn calls it back. Or it could depend on a component that conflicts with other components. Assemblers often don't find out until it hits them in the face. They find out the hard way that specific versions of support files must be installed into the environment or the system fails to behave properly.

An obvious solution would be to mandate that documentation contains more detail. Topics like preconditions, postconditions, and invariants could be provided to tell you more about design constraints for a component. A language like OCL[1] could be used for this purpose.




```
Employee->forAll(p: Person | p.age >= 18 and p.age <= 65)
```

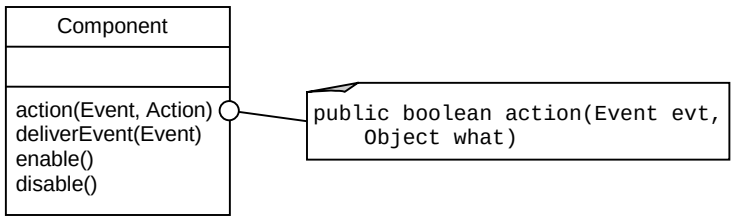
The above example shows that a constraint imposed on the `Company` component is that the age of all employees must be between the ages of 18 and 65.

While OCL is useful as a formal standard for documenting constraints, programmers don't have the desire or ability to follow such methods, and it still doesn't effectively capture interaction protocols or dependencies that exist between the components in question.

Another solution is to distribute source code with components. Source code represents the most detailed specification of component design that programmers use to codify structure and behavior before compiling it into binary form. It's the moment of truth for exactly how a component operates.

Unfortunately, many vendors are unwilling to relinquish such proprietary information. And when they do, it's often poorly documented or difficult to understand. The user could be an assembler without programming experience, or a programmer who isn't familiar with the language used to implement it. And getting familiar with the code to understand how to interact with it can take too long.

So if documentation and formal constraint languages such as OCL and source code aren't enough, what *is*? The answer lies in the heart of the original problem: poor design of interaction protocols. Let's take a look at an early version of the Java `Component` class.



As defined for the *Component* class in JDK 1.02

The intent of the `action()` method is to notify objects containing components when events occur. The following code defines a subclass of `Component` called `ButtonActionTest` that receives events when a user clicks on either of its two buttons.

```
public class ButtonActionTest extends Applet {
  public void init() {
     setBackground(Color.white);
     add(new Button("Red"));
     add(new Button("White"));
  }

  public boolean action(Event evt, Object arg) {
    if (evt.target instanceof Button)
       changeColor((String)arg);
    return true;
  }

  void changeColor(String color) {
     if (color.equals("Red"))
        setBackground(Color.red);
     else if (color.equals("White"))
        setBackground(Color.white);
  }
}
```

When a button is pressed, the `action()` method as implemented tests against its color to determine which button was pressed and sets the component's background to that color. What other events and arguments are possible? What if other components want to find out when a button is pressed?

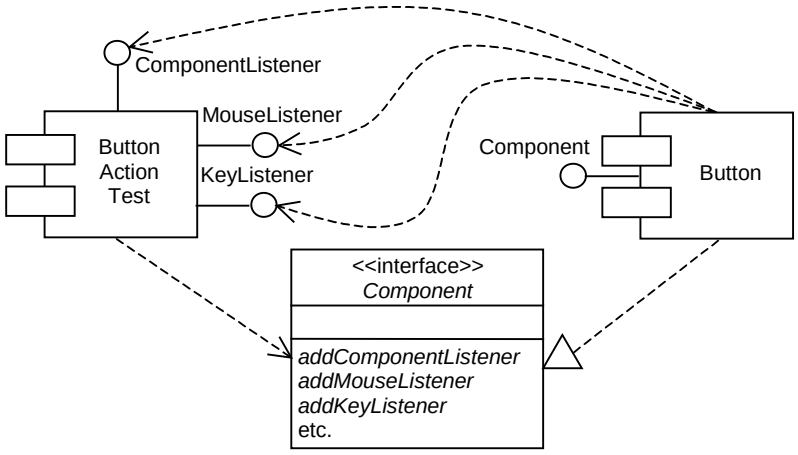
Too much work and too much knowledge of underlying events and arguments is required to make components interact the way you require them to. Even if loads of documentation is provided that describes all of

them, it still lacks clear design that welcomes errors.

The solution is to represent interaction protocols with abstract interfaces that define them. Many interacting components or complex interactions justify spending the time to provide component diagrams showing component relationships and dependencies they have upon their environment.

Therefore:

Reduce a component's dependence on its environment by defining interaction protocols between components separately from the components themselves. Specify these interactions in terms of abstract interfaces, and implement components to communicate with each other through them.



An improved design for *Component* in JDK 1.1

The example above shows listener interfaces that were added to JDK 1.1. While it's true that the `action()` method had a simple definition, it didn't provide any detail regarding the interaction protocols surrounding it. It's virtually impossible to provide a diagram outlining a design like that.

The listener interfaces are far less obscure, and it's easier to illustrate the interaction protocols between `ButtonActionTest` and `Button`. It reduces the burden of communicating interactions in documentation, and implies that every client that uses an object implementing the `Component` interface will need to interact with it through listener interfaces.

In fact, in the JDK, listeners have been included as a common idiom for allowing listeners of all kinds to be attached and detached to various classes that send events, including JavaBeans. Now the code for ButtonActionTest looks like this:

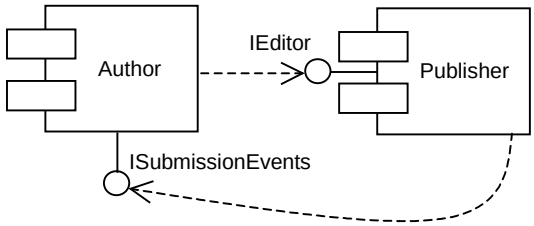
```
public class ButtonActionTest extends Applet
                        implements MouseEvent {
  public void init() {
     setBackground(Color.white);

     Button redbutton = new Button("Red");
     redbutton.addMouseListener(this);
     add(redbutton);

     Button whitebutton = new Button("White");
     whitebutton.addMouseListener(this);
     add(whitebutton);
  }

  private boolean mousePressed(MouseEvent e){
     if (e.target instanceof Button)
         changeColor(e.paramString());
  }

  private void mouseClicked(MouseEvent e){}
  private void mouseReleased(MouseEvent e){}
  private void mouseEntered(MouseEvent e){}
  private void mouseExited(MouseEvent e){}

  void changeColor(String color) {
     if (color.equals("Red"))
        setBackground(Color.red);
     else if (color.equals("White"))
        setBackground(Color.white);
  }
}
```

The behavior of the `ButtonActionTest` component is very straightforward. It's easy for the reader to see that it reacts to a button being pressed by changing its background color to the one equating to the text of the button that was pressed.

We also observe that no behavior has been defined for the other mouse events, or for that matter, any other kind of listener interface supported by a subclass of the `Component` class. Documentation can then be more concise, providing necessary information about constraints without describing detail more easily provided in component diagrams illustrating well-designed interaction protocols.

JavaBeans, implemented and used in a very similar manner to the Java `Component` class, announce events through event listener interfaces. Individual beans have no knowledge of how its listener interfaces are implemented.

A popular variation is to codify interaction protocols in a separate interface definition language. Most commonly used are the Microsoft Interface Definition Language (MIDL) and OMG's CORBA Interface Definition Language (IDL). To demonstrate it, let's look at an example based on Microsoft COM.



Interactions between an author and publisher…

The `author` connects, binds, and submits text to a `publisher` via the `IEditor` interface. Upon acceptance, rejection, or iterative feedback during the work-in-progress phase, the publisher sends events back to the author via the `ISubmissionEvents` interface. The interaction protocol[2] between author and publisher looks like this:

```
interface ISubmissionEvents
{
    void OnAccept();
    void OnReject([in] string reason);
    void OnFeedback([in] string comments);
};

interface IEditor
{
    long Attach([in] ISubmissionEvents* callback);
    void Submit([in] string title, [in] string text);
    void Detach([in] long cookie);
};
```

An IDL compiler is used to generate proxies for components that invoke methods (the client) and stubs for ones that implement them (the server). It operates like a factory that receives interface definitions as input, and generates source code in the desired language as output.

Whether a project uses Microsoft DCOM, OMG CORBA, and DCE/ONC RPC as middleware solutions for distributed component-based systems, the IDL compiler is an integral tool for separating interface from implementation, achieving location transparency, and maintaining language independence.

Other known uses of this pattern are the Darwin[3] and Regis[4] projects. They define component interfaces in terms of provided and required services, where services are typed. Each service type defined an interaction protocol to be used between communication endpoints at the component's interfaces.

❖ ❖ ❖

Certain themes arise regarding how components can collaborate without having dependencies hard-coded into their implementation. A COMPONENT BUS shows how components interact by indirectly communicating through a common routing mechanism. THIRD-PARTY BINDING centralizes responsibility of instantiating and binding components with a third party. And COMPONENT GLUE provides a way

to link interfaces together in order to fill in any gaps from incompatible components . . .

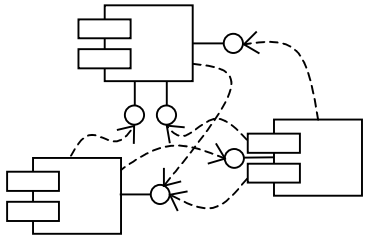
COMPONENT BUS*

The New York Mercantile Exchange.

… ABSTRACT INTERACTIONS describes how to reduce component dependencies by codifying their interaction protocols as abstract interfaces. While this goes a long way toward achieving implementation independence, it doesn't capture binding relationships and runtime behavior issues that occur between interdependent components. This pattern describes a way of binding communicating components without their being explicitly dependent on others.

❖ ❖ ❖

Systems consisting of components with many interdependencies can behave unpredictably or fail to operate altogether if explicit bindings they depend upon aren't established properly or connections are lost.



Imagine if there was a thousand of them…

When multiple components in a system communicate to get a job done, it can be very difficult to coordinate the process of instantiating

and binding them together when there are many explicit dependencies between them. It can quickly become impossible to manage. For example, when collaborating components have an explicit dependency upon one interface in each of the other components, the number of dependencies is N*(N-1), with N being the number of components involved.

Dependencies can thwart the ability to plug and play when it's required to do so. In a distributed system where load balancing is performed in reaction to unexpected usage volumes, you might want to migrate components to other servers and instantiated new ones to handle the extra volume. Hard-coded dependencies on other components can cause performance bottlenecks and failures that are extremely difficult to find.

One solution would be to centralize all dependencies. Build a component that serves as a directory and connection cache for the name and location of other components that communicate. Each time a component wants to interact with the other components, it grabs a connection from the directory.

But what happens when these connections fail without warning? From the point of failure until the connection is re-established and data is updated, data and notifications from the failed connection won't occur and integrity is breached. Reliability can become virtually impossible to achieve without a more flexible approach that maintains consistency.

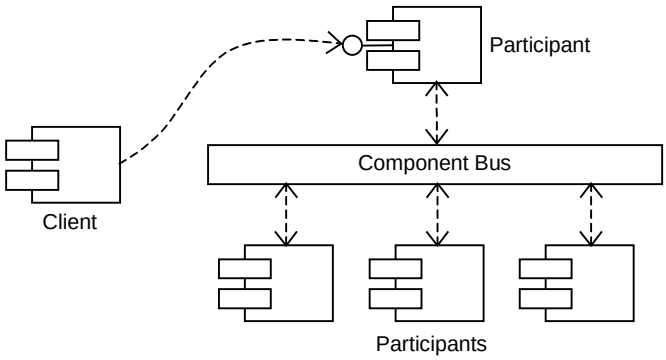
Another solution would be one similar to the Java listener concept. Each component connecting to other components receives notifications when any attributes are changed in other components. It then has the opportunity to react and update its state accordingly. But if a large amount of components exist, and both connections and location are very dynamic because of system and load requirements, then these explicit dependencies can be extremely difficult to manage.

The best solution is to allow components to communicate indirectly through an information "bus". An information bus manages the routing of information between participating components, from those that produce information to those that consume it. Routing of information is managed dynamically, as each participant can attach and detach to and from the bus without effecting the integrity of others.

When components are attached to the bus, they register interest in the information they require. When a component places information onto the bus, it is delivered to those participants that registered interest. Both the connections, and the data passed around as they communicate, are centralized. No COMPONENT GLUE is required to adapt between incompatible interfaces because all participants communicate through the bus.

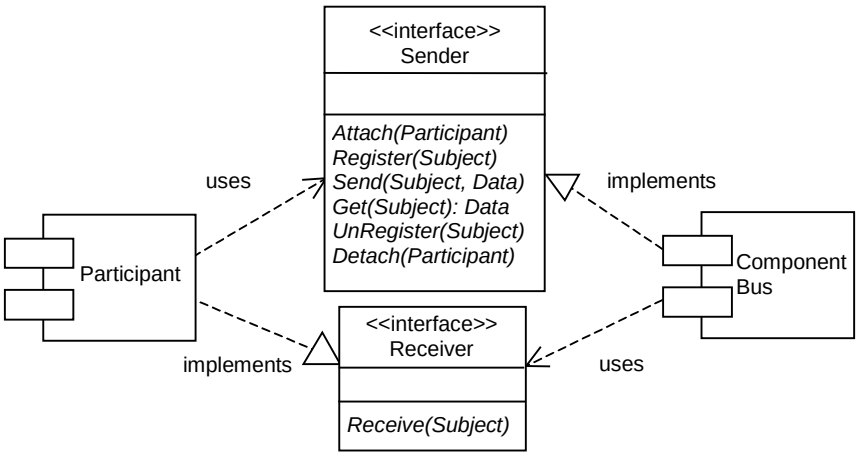
Therefore:

Bind components to an information bus that manages the routing of information between communicating components to remove explicit dependencies from the components themselves. Define interaction protocols that not only specify interfaces required for components to participate, but also the nature of interactions occurring between them.



At runtime, components acting as participants are instantiated and attached to the bus. Each participant communicates indirectly with other participants through the component bus. With the bus being used as a mediator for highly interdependent components, each component only explicitly depends upon the bus component being present at all times. This drastically reduces the number of connections required for communication to occur.

Instead of having N*(N-1) dependencies between communicating components, we now have 2*N dependencies to manage using the bus.

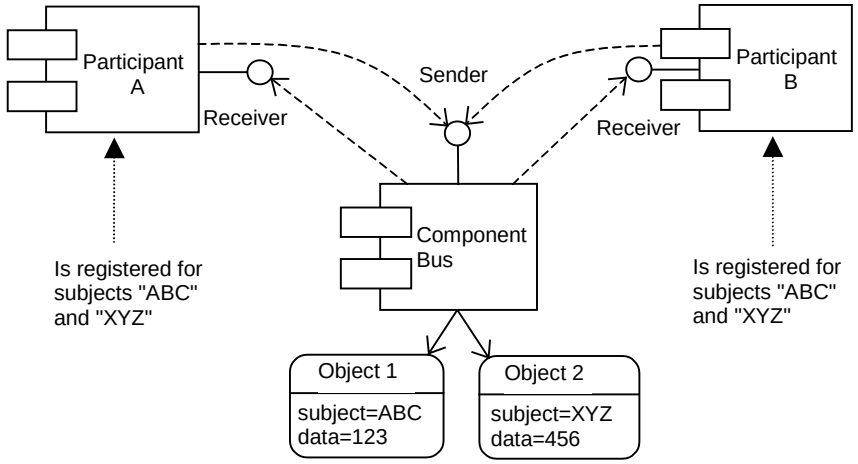


Interaction protocol between a participant and the component bus

With the interaction protocol shown above, participants are required to implement the `Receiver` interface and the bus implements the `Sender` interface. Participants create an instance of the component bus, attach to it, and register for all subjects in which they are interested.

When a participant wishes to update the data corresponding to a subject, it calls the `Send()` method, passing the subject and its updated data. When the bus receives this call, it caches the data for that subject and calls the `Receive()` method for all participants who have registered for it. As each participant receives it, it uses calls the `Get` method in the component bus to retrieve the updated data.

In addition to specifying the protocol between participants and the bus, it is important to specify the interaction that occurs amongst participants. The nature of the interactions in terms of how data flows between each of the participants and how they react to it can cause a circular data storm occurring if multiple participants update a subject in response to an update it received.



Possible circular data storm scenario

In the example above, *Participant A* increments the number corresponding to subject ABC in response to receiving an update for subject XYZ. *Participant B* increments the number corresponding to subject XYZ in response to receiving an update for subject ABC. The first update of either subject causes a circular data storm that can cause the system to crash or bog down the network because of increased traffic.

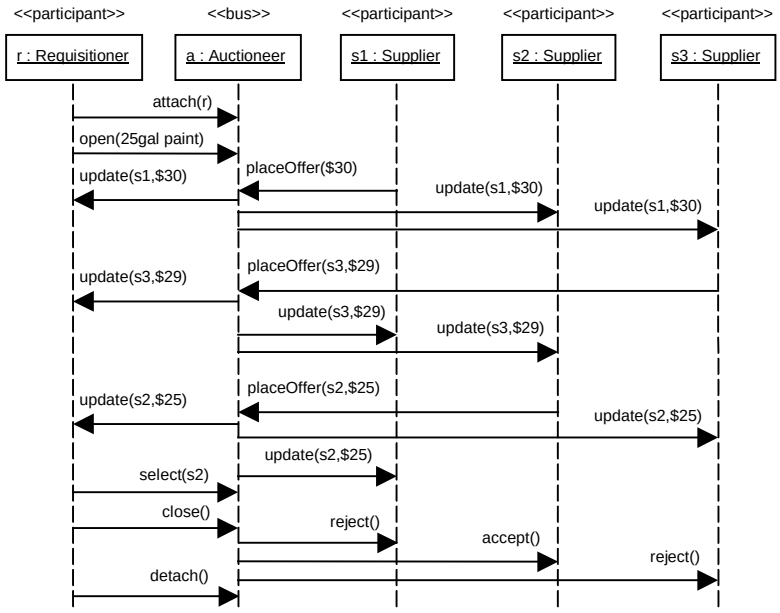
By specifying the interaction protocol that occurs between participants, rules can be established that allow programmers and assemblers to ensure that problems like this don't occur. This is very important to the success of using this pattern. Otherwise, the architecture of the system isn't easily visible from the source code or from its behavior at runtime.

Let us further illustrate the issue in a fictitious scenario involving a supply chain process between an automobile manufacturer and its suppliers. A procurement system using a real-time reverse auction process to allow each supplier to outbid other suppliers is used on purchase orders for parts used on factory assembly lines. The business benefit is great cost reduction and a more dynamic partner relationships.

Requisitioners enter bidding sessions with multiple suppliers using a COMPONENT BUS to communicate offers back and forth until a deal is accepted. An invoice is sent from the supplier to the requisitioner, accounts payable sends electronic payment, and a selected logistics provider is used to ship the parts in an overnight package to the factory requesting them.

During the auction, XML documents using a standard markup language for auctioning could be exchanged through the bus, which acts as a mediator for communication between suppliers and requisitioners. When a deal has been accepted, a commerce markup language like cXML[5] for the order payment and fulfillment process.

Interaction between each participant and the bus is simple, providing a single point of entry for sending and receiving data. But designers must additionally specify all possible interactions occurring through the bus via XML documents to ensure that the system meets requirements.



The above sequence diagram shows the interactions that occur in terms of the bus protocol, and stereotypes specifying each object's role in the COMPONENT BUS. The requisitioner and three suppliers act as participants, with the auctioneer acting as the bus. This diagram tells us a lot more about the interactions that occur through the interfaces defined by the bus.

A popular variation is one that uses a COMPONENT BUS that's not a component itself. TIBCO Rendezvous[6] is a popular alternative for bus architectures. A large investment bank uses a component framework providing publishers and subscribers used from Microsoft Excel spreadsheets and custom-developed applications for real-time data solutions.

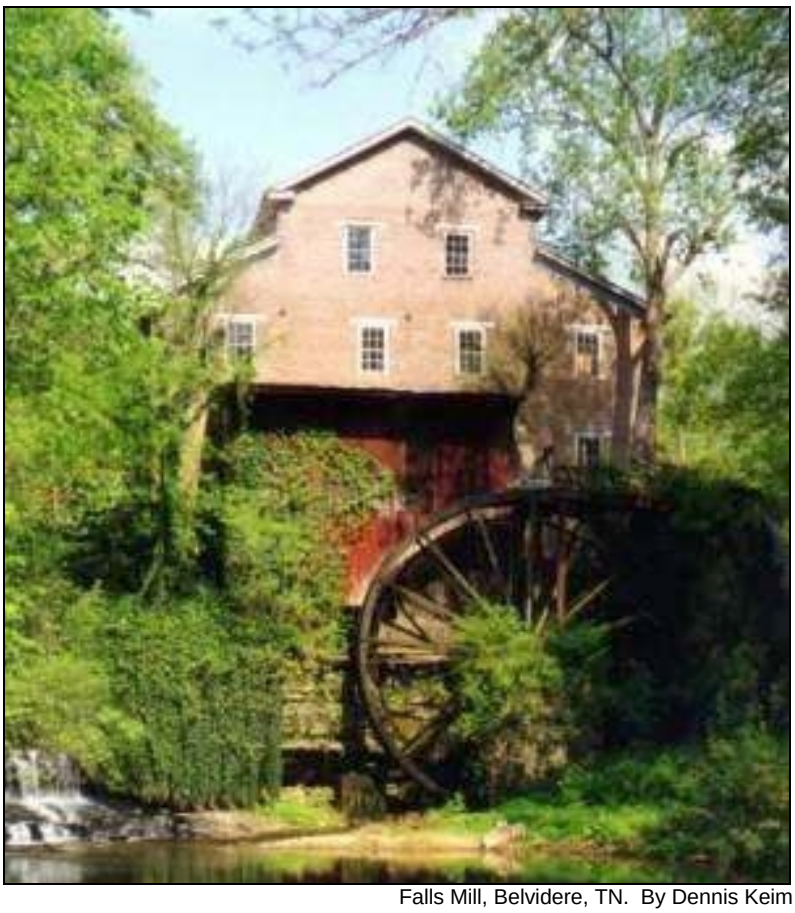
Each component links to a static library that interacts with a local Rendezvous daemon that communicates to central multicasting services. These services use IP Multicast at the lowest level, which provides a huge performance advantage over point-to-point or broadcast-based alternatives, when communicating messages between publishers and subscribers.

Other known uses include the following: The Java InfoBus[7] architecture implements a COMPONENT BUS for JavaBeans that exist in the same address space. Messaging middleware such as iBus[8], and tuple spaces such as Linda[9] or IBM TSpaces[10] and JavaSpaces can be used as a distributed COMPONENT BUS.

Other known names for this pattern are: Software Bus, Coordination Model of Distribution, and Tuple Space.

THIRD-PARTY BINDING can be used to instantiate all participants and bind them to the component bus. When incompatible components need to interact with the bus, COMPONENT GLUE can be used as an intermediary. CONNECTION SINGLETON can be used if many participants exist in one local address space to interact with the component bus. If connections are remote, CONNECTION OBSERVER can be used to allow each participant to observe changes in the state of its connection to the bus.

COMPONENT GLUE*



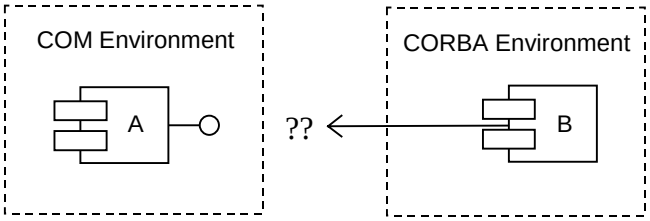
Falls Mill, Belvidere, TN. By Dennis Keim

… ABSTRACT INTERACTIONS describes how to reduce component dependencies by codifying their interaction protocols as abstract interfaces. COMPONENT BUS provides a solution for allowing components to communicate indirectly. Both patterns assume the assembler has full control over interaction protocols and implementations provided by all components. This pattern shows how to make incompatible components work together without sacrificing independence.

❖ ❖ ❖

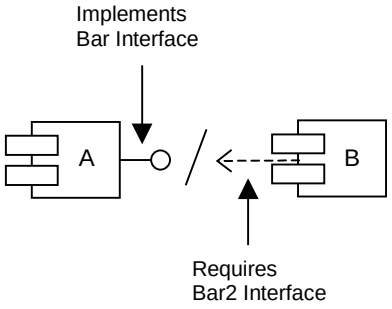
Assembling components that are incompatible in the way they communicate or interact can be time consuming and inevitably complicates system architecture.

Components using services provided by other components sometimes exist in different operating environments. Other times, they play incompatible roles, are compatible but reference slightly different type signatures, or were decoupled from other components by design.



Each component resides in a different world…

With clients and servers running different operating systems, and with components from outside the organization being acquired and reused, a common problem in distributed systems is that components operating in one environment need to communicate with components operating in another.



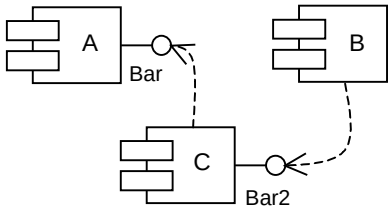
How do you make them communicate?

Even when in the same component environment, often incompatibilities come down to type incompatibilities. Above, component B expects interface `Bar2` from A, which implements `Bar`. They play compatible roles, but A implements an interface that is not compatible with B.

Furthermore, while loose coupling between components is a positive design trait, a side effect is that components are designed to be autonomous and don't interact with their peers. Components used to build GUIs (e.g., buttons, text fields, and lists) send events to their containers and receive messages from them but don't interact with each other.

Other times, data needs to be joined together from multiple sources and sent to a component for processing. Data retrieved by executing a query against a database, legacy data, or data from other components are often combined and sent to a component because it wasn't implemented to do so itself.

How do you connect incompatible components in situations like these? One solution is to build a third component acting as an ADAPTER which implements the interface required by the client, and performs a translation to the interface the server implements.



A third comes in and saves the day…

Above, C intercepts B's binding to A, implements B's required interface, and translates to A. This adds additional interactions to a system's architecture, and could result in performance or reliability problems.

The problem is that developing, integrating, and testing new components can be too time consuming to be considered a viable solution. For example, building a component that acts as a container for components which is itself embedded into the original container can overcomplicate design or be impossible in some cases. Let's look at a web page containing a form as an example.

```
<HTML>
<BODY>
   <FORM action="/scripts/sendaddr.dll">
      Name: <INPUT name="name" value="" size=40><BR>
      Address: <INPUT name="address" value="" size=80><BR>
      City: <INPUT name="city" value"" size=40>
      State: <INPUT name="city" value"" size=5>
      Zip: <INPUT name="zip" value"" size=10><P>
      <INPUT type="submit" value="Send">
   </FORM>
</BODY>
</HTML>
```

The HTML script above displays five text fields that allow the user to enter his or her name and address information. Below that, a "Send" button allows the user to submit their information to the web server for processing.

The "Send" button has no knowledge of the existence of name and address fields. Buttons don't have any knowledge of the existence of the text fields, and vice versa. The browser doesn't perform any validation check on the contents of the text fields. What if we don't want the user to submit any information until they've entered data in all text fields?

Obviously, building a third component that contains the text fields and button would be too complicated. It would need to interact with the web browser to send and receive messages and notifications to allow or disallow sending the request to the web server for processing. It would be difficult and completely coupled to the browser in question.

A popular Internet solution is for web servers to return an error page highlighting the problem once it receives the processing request. However, this is not always viable. Performance issues due to the

amount of data or the complexity of server-side data validation can be too expensive.

In addition, problems with incompatible components don't always involve the web or web pages, which further complicates the issue when component reuse is taken into account.

Another way of solving component incompatibilities is by using type coercions[11]. While they seem like a clever way to create components that are more flexible, they hinder the ability for the architectural view of the system to emphasize the behavior of this kind of interface adaption.

The Arian 5 disaster[12] is an example of a bug caused by type coercion. The code in question was perfectly valid in the software for the Arian 4 but was made invalid when other parts of the system were modified for the Arian 5.

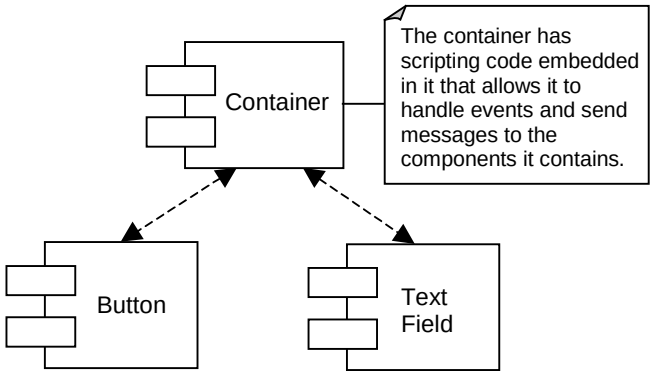
In all of these cases, it's easier to utilize an available and appropriate scripting or programming language to create "glue" code that mediates or adapts between disparate components. Glue code ties them together without increasing coupling, and prevents designers from having to introduce protocols riddled with adapters into the architecture.

Since the glue code isn't implemented as components, they don't plug into the framework themselves; instead, they *facilitate* the plugging of existing components into the framework.

In certain cases, however, replacing glue code with components is appropriate. For example, when you find yourself writing the duplicate glue code in many parts of the system, it is important to consider the value of refactoring the design to implement a component as a substitute.

Therefore:

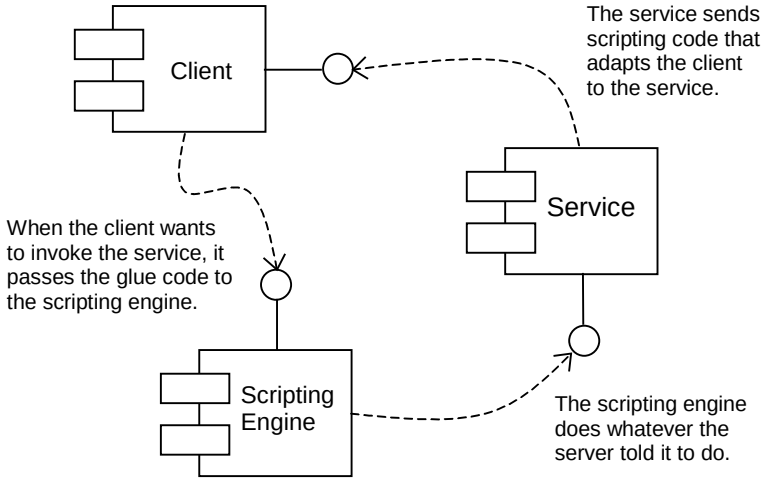
Create "glue" code to act as an adapter for incompatible components, or as a mediator between peers. Only build full-fledged components when glue doesn't meet all of your requirements.



Web pages requiring validation checks prior to submitting a request

to the web server for processing can use client-side scripting to react to the click event. Each text field can be checked for data, and an error message could be displayed instructing the user to enter data and the request cancelled if any one of them is missing data.

In addition to acting as an adapter or mediator between components, glue code can be used as a strategy to parameterize components with application-specific actions. Therefore, instead of glue code connecting disparate components being embedded in a third party, it is embedded directly in the components themselves.



Anything's possible…

In the above example, a service passes scripting code to a client component, which is executed in place of calling the service directly. The location of parameterization via glue code depends on the requirements of the application in question.

This is similar to how Jini clients and services interact. A Jini service discovers and joins a network by uploading its services as a proxy onto the Lookup Services. Clients who then come along and join the network through discover/join protocols and use the Lookup service to invoke the Jini service download glue code to allow for direct communication with the service via the proxy[13].

Tcl/Tk components use Tcl scripts as glue to tie the components into an application. Components evaluate scripts in the Tcl intepreter to announce events. The interpreter therefore acts as a mediator between components, and scripts act as application-specific strategies used to parameterize components.

JavaBeans development tools generate anonymous inner classes to act as glue between components, routing events announced by one component to the method(s) of another component that can handle the events.

❖ ❖ ❖

If components support INTERFACE DISCOVERY, glue code can be generated automatically from the interface definitions of the components it sits between. If this pattern is implemented using scripting, the script engine acts as a MEDIATOR between components and the scripts used to react to a components events can be thought of as STRATEGIES modifying the behavior of that component.

THIRD-PARTY BINDING*

Source: The U.S. Embassy and Information Service in Israel

… ABSTRACT INTERACTIONS describes how to reduce component dependencies and improve design by codifying their interaction protocols as abstract interfaces. COMPONENT BUS describes how to bind them together without being directly connected to each other. This pattern provides a solution for both cases where components are attached and detached by a separate component.

❖ ❖ ❖

Changing bindings in components that have internal dependencies to other components inside their implementation can make system maintenance a nightmare and reuse virtually impossible.

When a client component binds to a server component, often there is an explicit binding programmed into the implementation of the component that can introduce problems, bottlenecks, or a single point of failure once the system is a living, breathing animal.

If maintainers or programmers who are reusing components containing these bindings wish to replace it with a binding to a different component, the brittleness of the system can make it impossible to make changes while maintaining reliability.

One possible solution is to design the code around the connection to be generic by building an object, perhaps an adapter, which encapsulates the connection to the server component. The problem with this is that the solution is still centered inside the component's implementation, and

the architecture must include a coupled relationship between the client and server components.

A better way to solve this problem is to allow a third component (or outside party such as a runtime application initializer) to instantiate each component in the correct order and bind them to the appropriate services. Maintenance involving modification of bindings between components becomes much easier.

Therefore:

Remove connections established in the implementation of a component by having a third component bind two interacting components together.

The dependencies between components are visible at a higher level, making it easier to specify component interactions in the architecture, and easier to maintain the system. It becomes easier to reuse components, because they don't have implicit dependencies on other components in their environment.

Because interactions between components are made explicit, it is easier to model and reason about the behavior of the system in terms of the behavior of those components.

It is impossible to remove all implicit dependencies. Components will always be dependent on one or more component frameworks that define the interactions between them and other components. For example, a third-party can be used to bind and attach components to a COMPONENT BUS.

Microsoft's DirectShow media API implements stream filters as COM objects that expose one or more "pins" through which media frames flow. Frames are passed into a filter through input pins and out of a filter through output pins. Filter graphs are created to perform stream processing and rendering by instantiating components and connecting the output pins of up-stream filters to the input pins of down-stream filters.

A real-time market data component and distributed systems management component are bound together by a third-party in a project developed in-house at a large global investment bank. That third party can be Microsoft Excel or any other user of the components building COM-compliant systems.

Many architecture description languages describe system architectures in terms of only instantiated components and provide/require/use relationships between those components.

❖ ❖ ❖

In each of the components that are bound to others via this pattern, using the NULL OBJECT can be useful in defining an implementation-centric object that serves as a placeholder "null" connection to the server

component until the third-party attaches it.

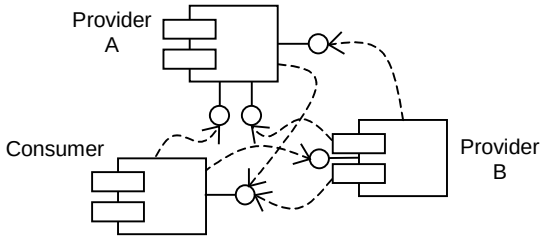
CONSUMER-PRODUCER*

[[[[insert picture here]]]]

… ABSTRACT INTERACTIONS describes how to reduce component dependencies by codifying their interaction protocols as abstract interfaces. While this goes a long way toward achieving implementation independence, it doesn't capture how to loosely coupled components to service providers providing similar services. This pattern describes a way of using service providers without being explicitly dependent on them.

❖ ❖ ❖

Forcing programmers and assemblers to explicitly depend upon components providing system services can complicate maintenance, limit reuse and acclerate the process of systems being classified as "legacy" systems.



Dependency management only worsens as new systems emerge…

Core system services are utilized in virtually every application in one way or another. Heterogeneous databases, naming and directory interfaces, and other services are used throughout systems in various components.

Major problems with this is that there is no consistent way to interface with these services, programmers and assemblers must re-learn new APIs each time new services and/or platforms are accessed, and as those services change or become obsolete, the value, reusability maintainability decreases rapidly.

One way to solve the problem is to build an abstract factory that centralizes the creation of objects that represent system services. The problem with this is that factories have the tendency to be embedded in every single component requiring these services, and reuse is difficult. In addition, the problem of different interfaces for different programmers and services still exists.

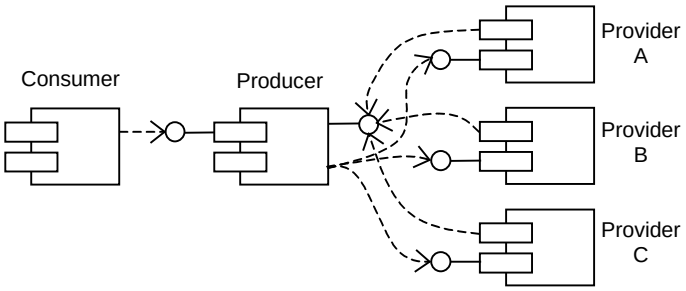
Another idea is to build a separate component for each system that encapsulates direct access to all system services used by an application, providing interfaces for each system accessed. The problem with this is

that the component needs to be developed and maintained separately, resulting in different levels of quality, and again each programmer is free to roll their own custom interfaces for each system, making consistency impossible.

The ultimate solution to this problem is to build a producer component that sits in between consumers and providers. It provides a single, generic interface that programmers can reuse for all services. By building back-end providers for each service, reuse on both sides can be accomplished in a far more consistent manner.

Therefore:

Provide components a unified interface to multiple, seamless connectivity to heterogeneous service providers.



Consumer utilizes unified interface for access to services…

The design of this solution includes a component (see Producer above) that fills the pivotal role of service manager. It manages connections to providers and provides a single point of access for components that use it (Consumer above).

From the consumer's point of view, an application programming interface (API) is defined that allows programmers to access a unified interface for all service access. On the provider's side, a service provider interface (SPI) is defined that allows programmers building service providers to conform to a proven producer and immediately allow existing consumers access to it.

In the middle, sitting between the API and SPI, is a manager that manages connections, instantiates and binds to services, caches data, etc.

There are many known uses for this pattern. Java Naming & Directory Services (JNDI) provides a common interface for obtaining metadata information about where objects and services can be found[14].

Microsoft's Universal Data Access[15] initiative includes a component framework called Microsoft Data Access Components (MDAC) that implements this pattern. Its producer component is called ActiveX Data Objects (ADO), and each service provider is implemented as an OLEDB provider.

❖ ❖ ❖

This pattern benefits from the use of CONNECTION SINGLETON and CONNECTION OBSERVER, since it requirements can depend heavily on providing connection management services to its consumers.

CONCLUSION

While many middleware platforms and component object models provide a nice way for clients and servers to communicate in a langauge- and implementation-independent manner, they don't focus on recurring themes and best practices that are present on real projects.

The *Component Interaction Patterns* focus on present themes that are seen on component-based projects where programmers implement interaction protocols between components based on abstract interfaces and connections that provide just the right amount of coupling between components and cohesion to meet the needs of its users.

ACKNOWLEDGEMENTS

Nat Pryce initially documented many of the Component Interaction Patterns and actively participated in follow-on discussion.

Joshua Kerievsky provided valuable input and was instrumental in convincing us that it was best to use the Alexander form to document them. He was the pattern shepherd for PLoP'99. Ken Auer was the Program Committeeperson who also provided comments and introduced it to the Triangle Pattern Language User's Group for review.

And thanks to Dafydd Rees, Brad Appleton, Mike Krajnak, and others who contributed to the Wiki Wiki Web discussion and evolution of the patterns in this language.

REFERENCES

[Alexander+77] C. Alexander, S. Ishikawa, M. Silverstein with M. Jacobson, I. Fksdahl-King, and Shlomo Angel. *A Patterns Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[Booch+99] G. Booch, I. Jacobson, J. Rumbaugh. *The Unified Modeling Language User Guide*, Addison-Wesley Longman, Reading, MA. 1997.

[Gamma+95] E. Gamma, R. Helm, R. Johson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Longman, Reading, MA, 1995.

[Szyperski97] C. Szyperski. *Component Software*, ACM Press/Addison-Wesley Longman, 1997.

[1] Object Constraint Language (OCL) is a formal specification for expressing object constraints. It is an integral part of UML and was used to formally denote its metamodel semantics. For more information, see http://www.rational.com/uml/resources/documentation/ocl.

[2] The IDL format is used for semantic simplicity. With MIDL, different parameter and return value types, metadata, the concept of a "coclass", GUIDs, and dispatch IDs are included that clutter up examples.

[3] *Darwin* is an architecture description language that was used as a configuration language for the *Regis* distributed programming environment. See http://groucho.doc.ic.ac.uk/research/darwin/darwin.html for more details.
[4] *Regis* is a distributed programming environment distributed as a set of class libraries, frameworks, and tools for the development of component-based concurrent and distributed systems. It was used on the *Management of Multiservice Networks* project at Imperial College in the United Kingdom. See http://groucho.doc.ic.ac.uk/~np2/regent/regent.html for more information.
[5] The Commerce Extensible Mark-up Language (cXML) is an open, standard application of XML to the procurement and order resource management process. See http://www.cxml.org for more information.
[6] Rendezvous is a cross-platform, scalable, high-availability data exchange platform that can be used over LANs or across the Internet. It includes the option of using IP Multicast as a means for efficiently notifying listening subscribers of data updates. See http://www.rv.tibco.com for more information.
[7] InfoBus enables the dynamic exchange of data between JavaBeans by defining a small number of interfaces between cooperating Beans and by specifying the protocol for use of those interfaces based on the notion of an information bus. See http://java.sun.com/beans/infobus for more information.
[8] iBus is a scalable, flexible set of enterprise products that deliver messages and business events among application through various supported network protocols in near real-time. See http://www.softwired.ch/products/ibus for more information.
[9] Linda is a coordination language consisting of a half-dozen operations and a tuple space (an ordered collection of typed data objects) that can be accessed from a Linda program. See http://www.sca.com/lfaq.html for more information.
[10] IBM TSpaces is a network communication buffer with database capabilities. See http://www.almaden.ibm.com/cs/TSpaces for more information.
[11] The VARIANT type commonly used with COM components to allow for variable type coercion.
[12] See http://www.cnn.com/TECH/9710/30/ariane.launch for coverage of this event. Also, see the Ariane home page at http://www.arianespace.com.
[13] S. Ilango Kumaran, *A Quick Look At Jini*, Java Report. Vol. 4, No. 8, August 1999.
[14] J.P. Morgenthal. *Understanding Enterprise Java APIs*. Component Strategies. Vol. 2, No. 2, August 1999. Also see http://java.sun.com/products/jndi for more information on JNDI.
[15] See http://www.microsoft.com/data for more information on UDI and MDAC.