# The Myx Architectural Style

The goal of the “Myx” architectural style is to serve as an architectural style that is good for building flexible, high performance tool-integrating environments. A secondary goal is to serve as a foundation for building such environments in Eclipse. It borrows most of its key properties from the C2 architectural style as well as the Weaves architectural style, with some optimizations for this particular application domain.

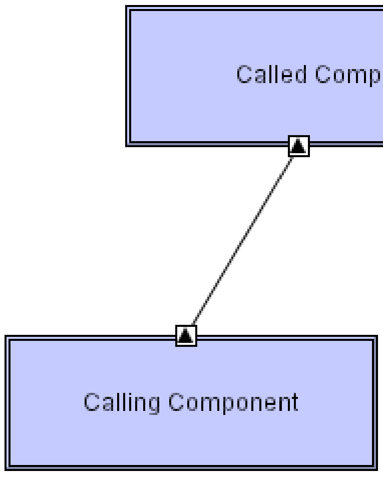
The rules of the style are:

- Components are the loci of computation
- Connectors are the loci of communication
- Components communicate only through well-defined provided and required interfaces
- Components and connectors have two ‘faces’ (also called ‘domains,’ in the C2 literature), ‘top’ and ‘bottom’
- Components interact through three distinct patterns
    - Synchronous bottom-to-top procedure call
    - Asynchronous top-to-bottom (notification) messaging
    - Asynchronous bottom-to-top (request) messaging
- Components may only make assumptions about the services provided above them, and may make no assumptions about the services provided below them.
- Applications have at least one main thread of control. Additional threads may be created by components as necessary. Asynchronous connectors also have their own threads.

Additionally, architects must decide whether or not to allow components to share memory and pass pointers. This significantly increases coupling and limits the ability of Myx architectures to be distributed across hosts. For these reasons, shared memory assumptions should not be made unless absolutely necessary—all parameters passed over a call or asynchronous invocation should be serializable. For some applications, it may be more useful to establish ‘regions of shared memory’ in which groups of components are allowed to pass pointers to one another, but not allowed to pass pointers across the regional boundary. This can be achieved from a modeling/implementation perspective using subarchitectures.

The three primary Myx composition patterns are described here:

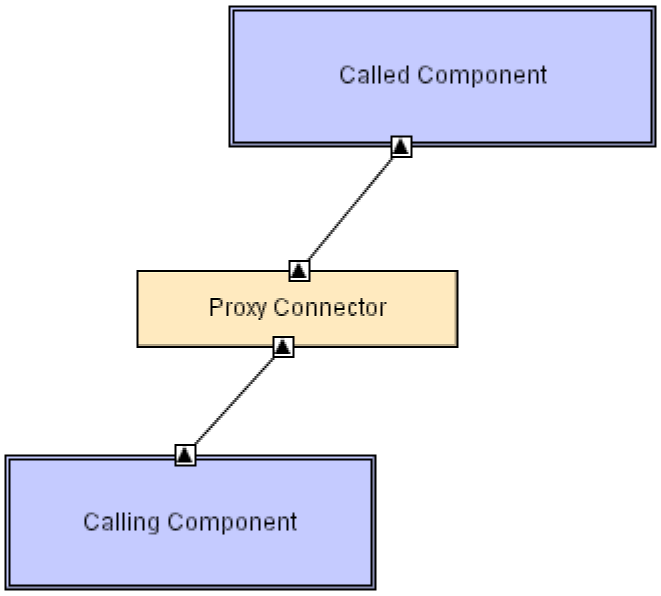
Pattern 1: Synchronous bottom-to-top procedure call:



This pattern represents a synchronous procedure call.  Synchronous calls may only occur from a caller component's top domain to a called component's bottom domain. In the default case, no explicit connector intervenes; the procedure call is an implicit connector. A call transfers the thread of control from the calling component to the called component in the usual manner.

Components may only make synchronous calls through explicit required interfaces to explicit provides interfaces.

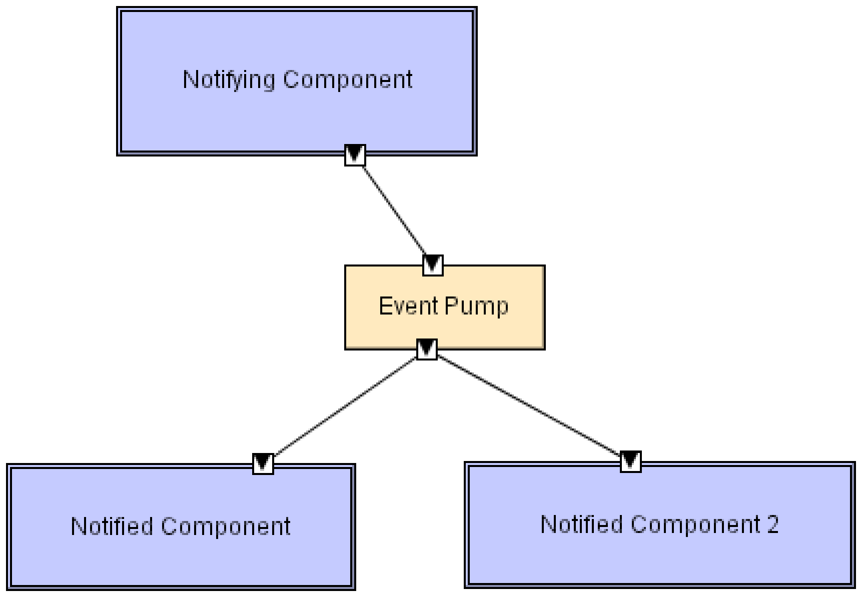
A variant of this pattern is allowed using a proxy.



Here, all interfaces should be of the same type. The proxy connector's main job is to pass the calls onto the called component, but may also provide additional processing on the call, communicating with other components through the same interaction patterns (sync bottom-to-top, async request, async notification) allowed in the style.  An example of this will be shown at the end.

Pattern 2: Asynchronous top-to-bottom messaging (notifications)

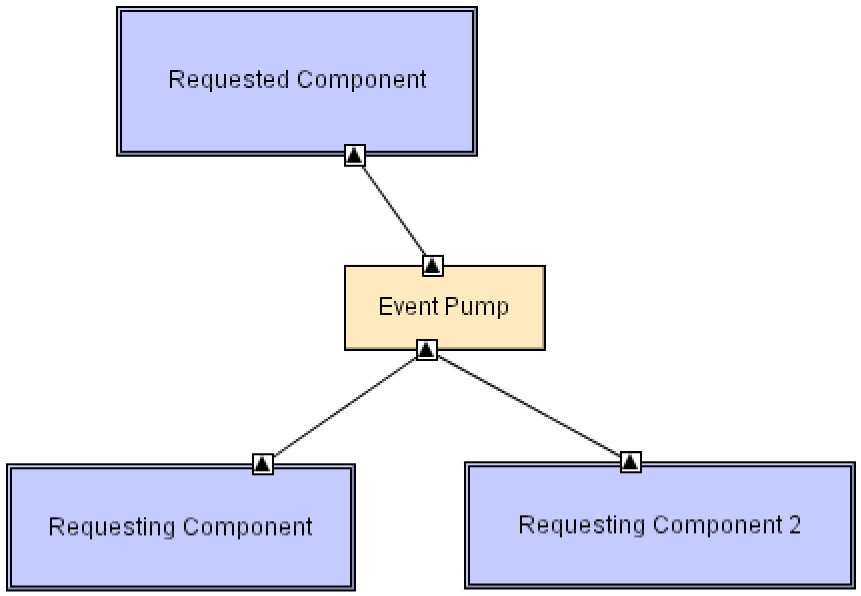
Asynchronous top-to-bottom messaging allows upper components to notify lower components of state changes through asynchronous messaging. The pattern looks like this:



All asynchronous notifications occur through the use of a distinguished connector called an *event pump*.  An event pump is a connector that receives messages and broadcasts them to receivers.  An event pump connector may only have one component on its top domain, but may have many components on its bottom domain. Event pumps have their own thread of control, and may not block the thread of the Notifying Component. This avoids deadlocks when asynchronous notification is implemented synchronously, as is often done in Java applications.

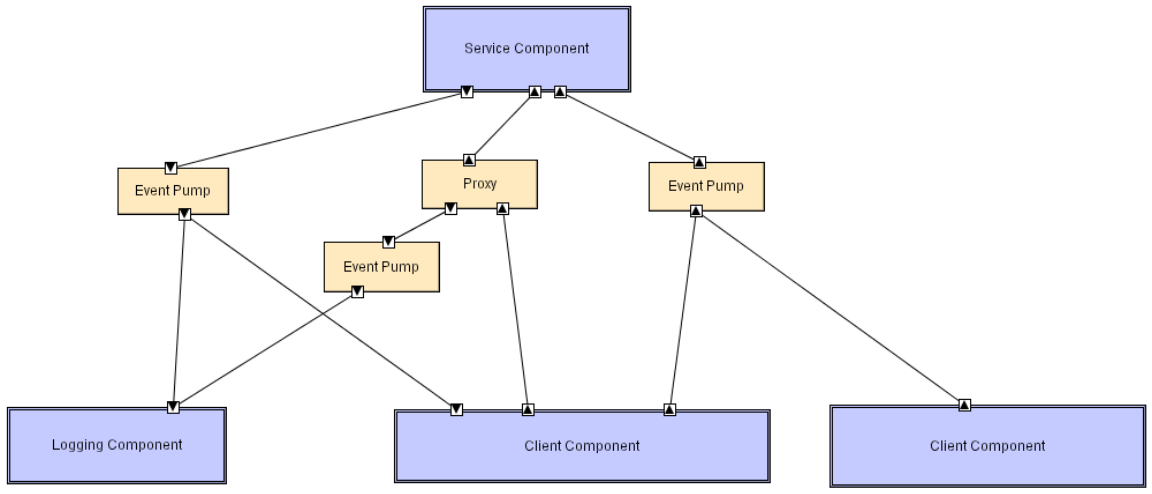
Pattern 3: Asynchronous bottom-to-top messaging (requests)

Requests work just like notifications, only the direction is reversed.  An event-pump connector is also used in this instance, except it acts as a fan-in (many-to-one) rather than a fan-out (one-to-many).  The pattern looks like this:



Here, the event pump still has its own thread of control. Requesting components may not be blocked when sending the message.

Here is an example of an architecture in the Myx style:



## *Implementing Myx Applications with the myx.fw Framework*

An architecture framework is a piece of software that bridges the requirements of a particular architectural style with the services provided by a programming language, operating system, and runtime library. In this sense, it is similar to middleware, except middleware is focused on providing specific capabilities (such as network transparency or logging) without specific regard to a target architectural style.

All versions of ArchStudio through ArchStudio 3 were implemented atop a C2 framework; this framework managed architectural topology, message queuing policies, threading policies, message exchange among components, and so on. Similarly, we constructed an architecture framework for Myx applications called myx.fw that supports the implementation of Myx applications. myx.fw is a Java library that contains classes and APIs for instantiating and managing Myx applications. Myx components and connectors are classes that implement an interface called IMyxBrick. Myx brick (component and connector) classes must minimally implement only two capabilities. First, Myx bricks must provide zero or more ‘lifecycle providers.’ A lifecycle provider is a class (possibly the brick’s main class itself) that implements four lifecycle methods: init(), begin(), end(), and destroy(). These methods are called automatically by the framework as the bricks are created, attached, detached, and destroyed respectively. Second, Myx bricks must provide ‘true objects’ for provided interfaces, given the identifier of the provided interface. Recall that Myx bricks have explicit provided and required interfaces; these interfaces are associated with objects that implement these interfaces. For each provided interface, a Myx brick must (on demand of the framework) produce the object that implements that interface.

An extended version of the IMyxBrick interface is available for brick implementations that want to provide support for runtime dynamism. This interface, IMyxDynamicBrick, requires that the brick implement additional callbacks so the framework can notify the brick when a link is being connected to or disconnected from one of its interfaces. This allows specific bricks in the application to support runtime dynamism where it is needed, without complicating bricks that do not need it. Dynamism can also be added to connections without recoding bricks through the use of dynamism-aware proxies and event-pump connectors.

myx.fw also provides a management API called IMyxRuntime that allows callers to manipulate a Myx application. This API has methods for adding and removing bricks, adding and removing interfaces from bricks, composing architectures hierarchically (i.e., bricks that have Myx substructure themselves), and connecting and disconnecting bricks via their interfaces. A default implementation of this management API can be created by an outside caller (e.g., a main() method), but it is also provided as part of a Myx runtime component, supporting multi-level construction (i.e., applications where a Myx application can create, control, and manage other Myx applications).

The myx.fw framework, like the Myx style, is tooled for performance. Specifically, aside from management tasks, the framework is entirely uninvolved in a Myx application’s operation. Unlike previous C2 frameworks where communication among components was entirely mediated by the framework, communication between components in Myx is direct. This means that the framework imposes effectively no runtime overhead on its applications.

## *Implementing Eclipse Plug-ins using Myx*

The Myx style and framework were developed with supporting Eclipse in mind, but are not in any specific way tied to the Eclipse platform—Myx applications can be specified, implemented, and run as stand-alone applications. Eclipse itself imposes a number of architectural constraints and assumptions upon its plug-ins, and these are not necessarily compatible with a style like Myx. We investigated the compatibility of Eclipse and Myx, and came to a number of conclusions.

First, it is (in general) possible to implement Eclipse plug-ins as Myx applications. In a pilot study, we were able to use the plug-in initializer class to instantiate a Myx application using the Myx runtime interface, and have this run in the context of Eclipse. We were also able to wrap the myx.fw framework itself as an Eclipse plug-in, suitable for use by other plug-ins. We did have to make some small changes to the runtime implementation in myx.fw to support Eclipse's dependency-based classloading model, in which a downstream plug-in cannot dynamically instantiate classes provided by an upstream plug-in.

Second, there is an architectural mismatch between Myx components and Eclipse extensions, namely Editors, Views, and Preference Pages. These extensions participate in the Eclipse UI—Editors are the document editors for various file types, views are the tabbed information providers that surround the editor, and preference pages contribute to the Eclipse preferences dialog. Ideally, Eclipse would be able to query a distinguished Myx component to identify all the various editors, views, and preference pages provided by an application. Then, this Myx component would query other components in the Myx application to return this information. If Eclipse wanted to start up or instantiate one of these elements, it would do so by invoking the Myx component.

Unfortunately, using Eclipse's current plug-in model, this behavior is simply not possible. Eclipse reads basic metadata about editors, views, and preference pages from specially formatted XML files provided along with the plug-in, and cannot be coerced to do otherwise. The reason for this is that Eclipse has a policy of "lazy instantiation"—it wants to present appropriate menu options as if all plug-ins are loaded, but only wants to actually load the plug-in classes if the user invokes one of those options. For Eclipse to dynamically query an application for this information would require loading the plug-in, and defeat lazy instantiation. Also, Eclipse loads and instantiates editor, view, and preference page classes directly, and there is no way to intercept this behavior or redirect it to, e.g., a distinguished Myx component. Effectively, this means that Eclipse can "reach into" the domain of a particular Myx component and instantiate its classes without asking first. To make matters worse, Eclipse assumes that all editors and views are running on the same host, in the same process, and therefore share memory, so it freely passes pointers to and from these elements, again violating a constraint of the Myx style.

To work around these limitations, we simply compromised the Myx style somewhat. We created a shared registry, called the EclipseRegistry, in which Myx components can register themselves upon plug-in startup. When Eclipse instantiates a view, editor, or preference page, the object implementing that element looks up its Myx component counterpart in the EclipseRegistry. From this point on, all communication between the editor/view/preference page and other parts of the application is done via the Myx component. While this strategy compromises the ability to distribute editors/views/preference pages across multiple hosts, Eclipse does not allow this anyway, so nothing is really lost. Within the confines of the Myx application, data and control passes among these elements in ways permitted by the Myx style.