

# ARCHITECTURE- CENTRIC DEVELOPMENT

---

USING THE AENTENDRE FRAMEWORKS

PRINTED: SUNDAY, AUGUST 28, 2005

---

PREPARED BY HILBERT COMPUTING, 2005



---

# ARCHTECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---

## TABLE OF CONTENTS

<b>INTRODUCTION.....</b>	<b>1</b>
<b>CONTROLLER FRAMEWORK.....</b>	<b>3</b>
<b>Controllers and Components.....</b>	<b>3</b>
<b>Containment Model Independence.....</b>	<b>6</b>
<b>Component Internals.....</b>	<b>8</b>
<b>Actions and Contexts.....</b>	<b>10</b>
<b>Controller Processors.....</b>	<b>12</b>
<b>Infrastructure Code.....</b>	<b>13</b>
<b>NEXT STEPS.....</b>	<b>15</b>



---

# ARCHITECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---

## INTRODUCTION

The marketplace is changing faster than ever before. Information Technology departments are responding to more varied demands, not only from internal customers, but as a result of the desire to securely and effectively share some information with other businesses and consumers. Not only does IT have to deal with an increasing pace of change, we have to deal with the increased complexity that comes with inter-networked inter-enterprise systems.

Fortunately, the Information Technology industry has evolved along with the business demands. The most effective way to deal with these contemporary issues is to take an architecture-centric approach to software construction. Traditionally, software has been developed by gathering requirements, creating a design document and developing the software to meet the business requirements. This approach is not well-suited for developing software that can accommodate requirements that are evolving, not well understood by the business community or requirements that the marketplace has not yet realized. This traditional approach also tends to diminish the focus on non-functional requirements such as application security and the operational aspects for diagnosing software problems after production deployment.

Creating an architectural basis for software construction enables software to be more resilient to change. This empowers decision-makers and IT professionals to more fully exploit the opportunities that their software can bring to the marketplace. Software architecture is difficult to explain since, by definition, it must be able to accommodate requirements not yet envisioned. The architecture typically describes the key abstraction points in the software and describes the mechanisms through which the parts of the application software can be decoupled from one another.

The application architecture is embodied in an application framework. The principles and abstractions that make up the logical architecture take shape in the frameworks implementation. The application framework is responsible for managing the lifecycle of the application, which includes reading the external configuration information and populating the object factories. It also serves to instantiate the services that are used for the non-functional requirements. Ideally, applications will have operational logging services, performance metrics services, security services, etc. that are made available to the business objects by virtue of the application frameworks. These services, as with the application architecture in general, are ideally implemented using frameworks that enable them to be replaced with different implementations without requiring change to the applications.

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

## USING THE AENTENDRE FRAMEWORKS

---

A small group of developers created the AEntendre Frameworks with the goal of implementing best practices in application architecture. The approach was straightforward. The group created a collection of frameworks that were largely decoupled but synergistic with one another. That enables companies to use just the frameworks that were appropriate for their environment without requiring a wholesale adoption of all of the frameworks. In addition, the group wanted to create a framework that would not duplicate the efforts in several open-source communities, but instead to leverage that work to the fullest degree possible.

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---

## CONTROLLER FRAMEWORK

Most non-trivial applications being developed today are built around the model-view-controller (MVC<sup>1</sup>) design pattern. This has been effective at decoupling the presentation of the information from the business logic that implements the functionality.

The MVC design pattern separates objects into three categories. The model objects represent an object representation of the real-world business functionality and the behaviors that modify the state of the business information. The views are used to represent the state of the business objects in a different ways, typically to the end-user of the application.

The controller provides the mechanisms for handling the events that change the state of the internal system. These can be user-interface initiated, time-initiated, events from external systems. The controller also provides the place at which much of the operational information can be gathered and managed. This includes the logging of events as they flow through the system, measuring response times, etc. This can also be the place at which permissions for application security are validated. In short, the controller is an anchor for the non-functional operational event management in addition to managing the workflow within the application. Since the controller embodies the most basic structure of the application, the use of the Controller Framework is effectively a wholesale commitment to the AEntendre Frameworks.

## CONTROLLERS AND COMPONENTS

At a high level, the controller framework looks like:

---

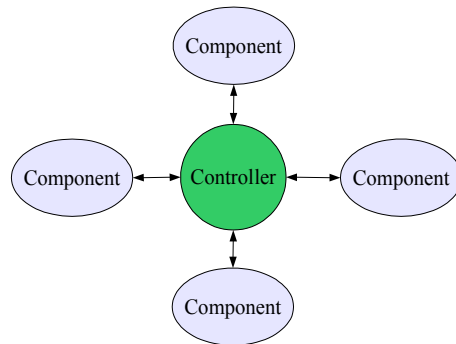
<sup>1</sup> Sometimes also called “Model 2”

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---



The controller is merely a dispatcher that transfers actions from one component to another. A component runs an action object which encapsulates a unit of work within the application. The action object produces an outcome which is transferred back to the controller to be dispatched to another component.

The mapping of the outcomes to actions and components is handled via external configuration. This enables the processing of actions to be decoupled to a very large degree. This enables possibilities for reuse for common functionality and reduces the impact to changes throughout the lifecycle of the application as features are added, removed and modified. This external configuration also means that the application has exceptionally late binding. Instead of binding the parts of the application at the source-code level, or at link-time, this allows the application to be bound at runtime. In essence, this framework allows the application to assemble itself when the application starts. While this has benefits for any enterprise application, this has special value for companies that tailor their software for specific implementations for each customer to which they sell their technology solution.

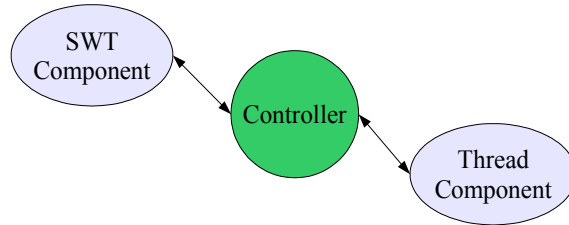
At this level of abstraction, it is difficult to see how this applies to a real-world application, so here is a more concrete example.

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---



In this case, we have two components. The “Thread Component” is designed to run the action objects using asynchronous queuing. In other words, when the controller transfers control of a action to the thread component, the action is queued and the controller is free to continue dispatching actions in parallel with the action running on the thread component. The “SWT Component” implements a framework to render a user interface using the Standard Widget Toolkit (SWT). SWT is a desktop GUI that is similar to Java Swing but uses the native controls on a specific platform.

It is important to note that the action objects, which we will examine in more detail later in this document, are unaware of the component in which they are running. The controller provides the necessary decoupling between components, thus creating a high degree of autonomy in the implementation of the action object and improving the potential for reuse.

Unless the programmer artificially couples the actions that run on the different components, there is a high degree of freedom to make significant changes in the implementation. For example, the SWT Component could be replaced with a Swing Component<sup>2</sup> without inducing any change on the action objects that run in the Thread Component.

This also enables us to change the threading implementation of the application. Note that we stated that the thread component runs actions using asynchronous queuing. That means that those actions run in parallel with actions on other threads, but the actions queued on that thread component will run serially. Assume that we have a few actions that run for an extended period of time, such as report generation. We want to enable the user to continue to interact with the application while the report is being generated. To accomplish this, we create an additional thread component that will asynchronously run

---

2 Although at the time of this writing, there isn't a Swing Component developed

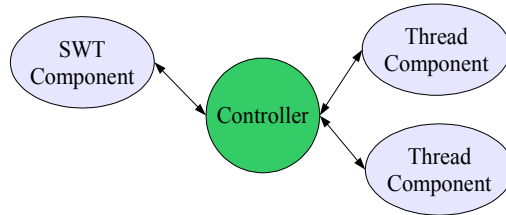
---

# ARCHITECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---

the report generation actions and leave the original thread component for processing the quick actions.



That kind of change, which involved adding another thread, segregating the long and short actions (which changes the end-user response time experience) can be accomplished with exactly zero lines of Java code! This change is accomplished by changing the configuration document for the controller to instantiate the additional component and dispatch selected actions to the new component.

## CONTAINMENT MODEL INDEPENDENCE

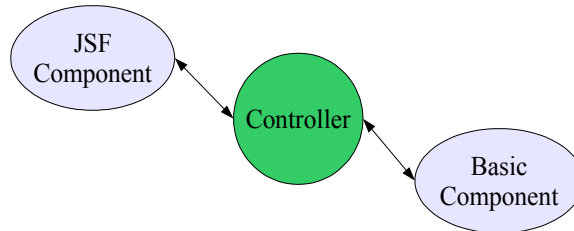
The above example is a Java application as opposed to a Java servlet. The presentation layer is a desktop graphical user interface (SWT). The action objects that run in the thread components should have no awareness of the component in which they are running. That means that the action objects (and I promise I will explain these soon) are also unaware of their containment environment. That means the actions that are not user-interface specific, which is often the bulk of the application, could run in a servlet environment as well as an application environment. That implementation would look like the following:

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---



In this implementation, the threading model is changed to conform to the requirements of the servlet model. The “Basic Component” will execute actions synchronously (meaning the controller that calls the component is unable to dispatch actions on its thread until the component finishes the action). We also changed the user interface from SWT to a web-based interface using JavaServer Faces (JSF).

This is a substantially more radical change than switching from SWT to Swing or changing the threading model of an application. It would no doubt result in some Java code changes, but probably not as much as you would expect. All of the business-logic actions will continue to run unmodified (unless the programmer errantly created references to SWT components in those actions). The information being passed from the Thread/Basic Component(s) to the component for rendering the user interface (SWT/JSF) should have been encapsulated in a view object, so there is considerable reuse opportunities for those encapsulated views. Only the rendering of the display changes.

Admittedly, it is fairly rare for an application to change its containment model, but this level of decoupling and abstraction still has value. We have seen the need for existing servlet applications to be friendly in a portal environment. While portal implementations can render the view of a servlet within a portlet, it can be advantageous for an application to gain access directly to portal services for security or profiling. This application architecture would enable the replacement of the JSF Component with a Portlet Component<sup>3</sup> and significantly ease the cost of that change.

---

3 At the time of this writing, there is no Portlet Component, but one could certainly be written without much difficulty

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---

There is also a need at times to reproduce some of the processing in the interactive systems in a batch processing environment. This would enable reuse of the binding layers between online systems such as web and desktop applications with those in batch jobs.

I have also seen where some long-running functionality in servlets should be moved to an external processing facility. That is, a long running part of a servlet – such as report generation – should not block the web browser from being able to continue with other parts of the application. Since this framework provides container independence, that functionality could be shipped to an external process instead of running within the servlet JVM with very little effort.

Perhaps the most compelling reason for this level of decoupling is the one most difficult to articulate. A well-designed architecture can accommodate requirements as yet unenvisioned. While the servlet container seems to have the mindshare within the Java community, there is no telling what may come on the scene five years from now. Having decoupled architectures with runtime binding allows us to exploit those new innovations much sooner than a tightly-coupled, monolithic architecture.

## COMPONENT INTERNALS

We have talked about the controller and the components at a high level of abstraction. The examples start to give some insight into the architectural richness that it brings to the application. As we look into the internals of the component implementation, we will start to see some additional value that this architecture brings as well as demystifying how some of this “magic” occurs.

The role of the controller is fairly simple. All it needs to know how to do is accept outcomes from actions that have run on a component and use that outcome to determine which component, if any, receives the next action.

A stripped-down version of the controller interface follows:

```
package org.aentendre.framework.controller;

public interface Controller {

    void transfer(String componentName,
                 Outcome outcome,
                 ControllerContext context);
```

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---

```
void setComponentRegistry(ComponentRegistry registry);
ComponentRegistry getComponentRegistry();
ControllerConfiguration getConfiguration();
void initialize(ControllerConfiguration configuration)
    throws Exception;
void shutdown();
}
```

The `initialize` and `shutdown` methods handle the lifecycle management of the controller (and most likely the application itself). The `ComponentRegistry` is a collection of named components.

The bulk of the code is in the implementation of the `transfer` method. Each component has a reference to the controller that manages it. When a component has completed running an action, it will transfer control to the controller with a reference to its name, the outcome of the action and the context of the action. The controller then determines, based on the `Outcome` object, which action is to be run and the component on which that action is to be run.

Each component is required to implement the `Component` interface, a stripped down version of which follows:

```
package org.aentendre.framework.controller;

import java.util.Map;

public interface Component {
    Map getComponentAttributes();
    void receive(String dispatchId, ControllerContext context);
    void initialize(Controller controller) throws Exception;
    void shutdown();
}
```

Each component is responsible for receiving the name of an action to be dispatched and the context of that action. In essence, each component is a micro-container environment that is responsible for executing application requests.

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

## USING THE AENTENDRE FRAMEWORKS

---

### ACTIONS AND CONTEXTS

There has been a lot of talk about these action objects. As promised, I am going to finally explain what they look like. The code that runs in a component must implement the `ControllerAction` interface. That interface is:

```
package org.aentendre.framework.controller;

public interface ControllerAction {
    String process(ControllerContext context);
}
```

That interface is exceptionally simple. It will perform processing of some sort and return the name of an outcome, which the controller uses to determine the next action. The controller context defines the programming model for these actions. So what is the controller context.

The controller context was modelled on an understanding of the lifecycles within an application and provides a collection of named objects<sup>4</sup> that correspond to those lifecycles. In order to understand how to use the controller framework, you must understand the following lifecycle scopes:

<i>Scope</i>	<i>Use and Meaning</i>
Application	Objects in application scope exist for the duration of the application.
Component	Component scope variables are available only to request running in that component. For example, the SWT component places all named user-interface widgets in component scope so they are accessible only to requests running in that component (and preventing the programmer from passing a reference to a widget to another component and creating an inappropriate coupling).

---

<sup>4</sup> This is implemented as a Map. In some languages, typically untyped or loosely type languages such as REXX, this is called a variable pool.

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---

<i>Scope</i>	<i>Use and Meaning</i>
Session	<p>The definition of what constitutes a session is up to the implementation of each component. A session will often model the lifecycle of an authenticated user or a network connection. In the case of components that run in a servlet container, the controller session maps directly to the <code>HttpSession</code> object.</p> <p>In the case of the <code>DaemonComponent</code>, that implements an <code>inetd</code> type network server, the session maps to the lifecycle of the network connection to a specific client.</p>
Request	<p>Request-scope variables exist for the duration of an atomic unit of work. Again, this the definition is up to the implementation of the component. In the case of those components that run within a servlet container, the request-scope variables map directly to the <code>HttpRequest</code> object.</p> <p>In non-servlet components, the request lifecycle continues until the outcome returned to the controller is a null reference or does not map to any configured outcome in the controller configuration.</p> <p>The implementation of the component determines how a new request scope is created. In the SWT component, there is a new request scope created for each user-interface action such as a button press. In the <code>DaemonComponent</code> that implements a TCP/IP-based network server, a new request scope is created whenever a new record is read from the network.</p>
Client	<p>Client-scope variables are used with there is a two-process implementation where the variables are contained on the remote client.</p> <p>In the case of components that run in a servlet container, the client-scope variables map directly to the cookies stored in the browser</p>

This describes a programming model that should be familiar to servlet developers. Much of servlet development involves accessing information from the request scope and updating the objects contained in session and application scope. This provides a similar programming model, with the addition of the component-scope variables, but without the coupling to the servlet container programming interfaces.

It is important to note that an action object should be used to bind controller events to business objects. The actual business logic and business object model that is derived

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

## USING THE AENTENDRE FRAMEWORKS

---

from requirements gathering, use cases, etc. should not be coded in the `ControllerAction` objects. The actions should be used to move information from the controller context, transform that information as appropriate and call methods on the business objects to change the state of the application.

### CONTROLLER PROCESSORS

The components are responsible for servicing the request. As mentioned, each is essentially a micro-container environment within the application. The implementation of each component involves constructing the resources specific to that component<sup>5</sup> and establishing an implementation of a `ControllerProcessor` that manages the factory construction and execution of the actions within that component.

The implementation of the controller processor implements the interceptor design pattern to enable the declaratively assembly of processing for an action. The controller processor also makes any security authorization checks that were configured. The pseudo-code for the processor follows:

```
try {
    make security authorization checks
    run preprocessing interceptors
    construct and run the controller action
    run the postprocessing interceptors
}
catch(...) {
    run exception interceptors
}
finally {
    run final interceptors
}
```

The interceptors can be declaratively attached to an action before or after it runs, only where there is an exception or after processing is complete, regardless of any uncaught exceptions that were thrown.

The implementation of the interceptor design pattern – similar to aspect-oriented programming - can resolve cross cutting issues within the object model. For example, there are interceptors within the framework that will dump the contents of the controller con-

---

5 For example, a network daemon component would set up a TCP/IP listening socket. The SWT component would construct the user-interface widgets that were configured.

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---

text variables to the logging facility. These can be attached to an action to aid in diagnostics and then removed without changing the implementation of the action class itself.

Interceptors could also be used to measure the response times of an action and report on the success of performance tuning in a production environment and then removed when the necessary diagnostics are gathered.

The controller processor is completely independent of the application binding code that is implemented in the controller actions. It is also invoked for each of the very granular actions that are processed within the application. This makes this ideal for the implementation of infrastructure services.

## INFRASTRUCTURE CODE

Most applications are developed with the business functionality as the only focus. This makes perfect sense since the development of software is there to support the business. However, there is another customer of the software that is often overlooked: the operational support staff. The better software development life cycles make an accommodation of these non-functional requirements, but in my experience, these are always short-changed, always to the detriment of the enterprise as a whole.

Once the software is placed into production, there are operational considerations that need to be managed. The most important is the availability of the application services. When there is dysfunction within the application, the operational support staff have to make a determination if the problem is related to the environment, such as a database engine being down or a network segment unavailable, or whether the dysfunction is a result of an application defect. In order to make that determination, the operational staff needs to have insight into the application's view of the dysfunction. In Java, that typically results in exceptions being thrown or application “hangs” of some sort for which there is no processing within the application.

The controller processor is the ideal place to attach non-functional requirements that are needed for all applications, but would otherwise not be included in applications due to delivery schedules or budgetary constraints. A basic set of infrastructure services for logging, performance metrics<sup>6</sup> and other diagnostic information can be included in the

---

<sup>6</sup> At the time of this writing, performance metrics have not been integrated into the controller processor. However, when the metrics framework is complete, all framework applications will get the functionality “for free” with no code modifications

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---

controller processor – and enhanced over time – so that all applications within the enterprise can benefit without having to slow the development of the business functionality to create these non-functional requirements each time. Because this functionality is pushed into the frameworks, infrastructure enhancements can be enjoyed by all applications without requiring a change to the application code.

---

# ARCHITECTURE-CENTRIC DEVELOPMENT

USING THE AENTENDRE FRAMEWORKS

---

## NEXT STEPS

This is a draft of this document. The intent is to document the motivation and overview of all of the frameworks in the collection of AEntendre Frameworks. The choice was made to start with the controller framework since it involves the greatest amount of commitment to the frameworks with presumably the greatest amount of benefit.

### Alphabetical Index

aspect-oriented programming 12  
authorization 12  
Component interface 9  
controller interface 8  
ControllerAction 10  
interceptor design pattern 12  
JavaServer Faces 7  
JSF 7  
lifecycle scopes 10  
MVC 3  
runtime binding 4, 8  
security 12  
Standard Widget Toolkit 5  
Swing 5  
SWT 5  
SWT Component 5