THE MISSING LAYER
Deficiencies in Current Rich Client Architectures, and their Remedies

Brendan Lawlor and Jeanne Stynes
Department of Computing
Cork Institute of Technology
Cork, Ireland

Keywords: Rich client, rich internet application, model view controller, software architecture.

Abstract: There is an architectural deficit in most rich client applications currently undertaken: In n-tier applications the presentation layer is represented as a single layer. This fits badly with business layers that are increasingly organized along Service Oriented Architecture lines. In n-tier systems in general, and SOA systems in particular, the client's role is to combine a number of services into a single application. Low-level patterns, mostly based on MVC, can support the design of individual components, each one communicating with a particular back end service. No commonly understood pattern is currently evident that would allow these components to be combined into a loosely coupled application. This paper outlines a rich client architecture that addresses this gap by adding a client application layer.

1 ADDRESSING DIRECT COUPLING IN COMPONENT-BASED ARCHITECTURES

A common problem in the design of a component-based rich client application of any complexity is the absence of any well-defined way of connecting its constituent components together in a predictable and maintainable way without coupling those components directly to each other. The importance of decoupling lies in the reusability it confers on the component. Development projects often result in either (i) relatively decoupled components that are connected to each other by a large, ad hoc and difficult to maintain amount of glue code (also know as the Big Ball of Mud[i]), or (ii) components that cooperate in a more systematic way, but as a consequence come to know a little too much about each other, and become coupled to the extent that they are no longer reusable beyond the context of the application.

Assuming that the Big Ball of Mud is always to be avoided, then the main obstacle to a component-based, rich client architecture is tight coupling between the components. Objects cannot operate as components[ii] unless they can function both independently from and in collaboration with other components. Current rich client architectures do not satisfactorily address this seemingly paradoxical requirement, and permit only a limited decoupling between components that still results in some direct coupling. So there remains an architectural deficit in which developers of rich clients have to work. To address this problem, this paper outlines an approach, called Application Model View Controller (AMVC), that breaks the presentation layer into an application layer and a component layer.

In enterprise systems, the client code is typically represented as a single layer – the presentation layer. (For Internet-based applications a distinction is made between server-side elements and client-side elements, but this is a deployment distinction, rather than an architectural one.) For both desktop and Internet based rich-clients, this single layer has not been enough. Single-layer architectures tend to result in bundles of components, directly coupled to each other as required by the needs of the application. Breaking applications down into components does not in itself reduce the direct coupling between the resulting components. An explicit application layer, specifically designed to prevent direct coupling, is an essential ingredient in any successful rich-client architecture.

In order to ensure the absence of direct coupling which is required to sustain a true component-based architecture, we need to address the features of

Lawlor B. and Stynes J. (2007).
THE MISSING LAYER - Deficiencies in Current Rich Client Architectures, and their Remedies.
In *Proceedings of the Second International Conference on Software and Data Technologies - SE*, pages 351-356
DOI: 10.5220/0001325803510356
Copyright © SciTePress

component-based presentation layer architecture that can lead to direct coupling: *events shared between components*, *event data shared between components, and containment relationships between components*.

The following sections describe the AMVC architecture, which removes each of these causes of coupling between components.

1.1 The AMVC Component Abstraction

The first step in defining a component-based architecture is to define the abstraction of the component itself. The AMVC architecture's notion of a component is heavily influenced by the hMVC[iii] (hierarchical Model View Controller) design (and therefore on PAC of which it is a specialized case[iv]), even to the extent of using the same terminology. But the AMVC component abstraction diverges from hMVC in a number of ways. AMVC can be said to be a modification and an extension of hMVC since the AMVC architecture also incorporates an extra layer of indirection that hMVC does not possess. A summary of the precise nature of the differences between the two architectures is provided towards the end of this paper, after a fuller explanation of AMVC has been offered.

An AMVC component is a Model-View-Controller grouping (Triad) where the controller has a function beyond the coordination of events between the model and the view: *It also directs events out of the triad to other triads.* Components are organized into a hierarchy as in hMVC and controllers act as the point of contact between the triads. Each component has its own individual functionality, but in the context of the AMVC architecture, the component is abstracted as a Triad that, from an external perspective, is simply capable of receiving events, emitting events, and entering into a parent-child relationship with another triad. Another public aspect of a Triad is the view itself - this will be described later. We now consider the way these components are organized into applications.

1.2 Two Layers – Two Activities

We can identify two separate layers in the construction of a rich client using AMVC. The component layer is composed of independent and functionally specialized Triads. The model elements of these Triads typically communicate with particular services in an SOA[v], if they have any server-side functionality at all. The application layer consists of application-specific configuration information that links instances from the component layer - without introducing coupling between these components. It is this absence of coupling that defines the AMVC architecture and represents its main value proposition.

The work of developing a rich client application can be broken up into two activities that correspond to the two layers above: component development and application assembly. In the absence of direct coupling between the components, applications can be assembled by combining and re-combining components in different ways, re-using the same component types many times within one application, and achieving different results with each combination.

To appreciate how this architecture provides for decoupling between components, it is important to understand the factors that lead to coupling in the first place. But in order to do that, we must first look at the different kinds of events defined by AMVC.

1.2.1 AMVC Event Types

There are two kinds of events defined by AMVC: Component Events and Application Events, corresponding to the Application and Component layers. Within AMVC components, the Model and View elements typically communicate with each other by means of Component Events (implementations allow for views directly calling model for efficiency and simplicity– though not vice-versa). Some of these events are exposed outside the components as emitted events, or received events, or both.

Note that Component Events should be distinguished from what we will here call *widget events*. The latter are well understood concepts of GUI programming and communicate low-level interface-specific events (e.g. 'button A has been pushed') to components. The component layer intercepts, interprets and combines these widget events into Component Events, which represent reusable business functionality for a specific service (e.g. 'List the Orders'). Application Events are described below.

1.2.2 Coupling Due to Event Sharing

Event-passing is a common way for components to communicate. Although this can often be done in such a way that avoids the programmatic coupling of the communicating components, a form of coupling remains: the name of the event itself. If an event is passed from one component to another, the communicating components must have a shared understanding of what that event means. This

constitutes a semantic coupling between the components.

In AMVC, to avoid the coupling that comes with sharing events between components, Component Events are never passed from one component to another. Instead, components communicate with each other through Application Events. These Application Events are specific to the way in which the Components have been recombined and represent user experience or even steps of a use case (e.g. List All Orders for the Selected Product).

Figure 1 shows how components communicate with each other through Application Events. Component Events are 'translated' into Application Events at the interface of the emitting components, and these Application Events are further translated into another Component Event at the interface of the receiving component.

This event indirection removes the first source of inter-component coupling, namely shared events between components. It is not just the fact that the translation is done but where it is done that counts: It is done in the Application Layer, and therefore during application assembly. Thus, the components themselves (that is to say their source code and their runtime states) are not affected by this translation. They remain completely independent of what happens to their events once those events leave the components' scope. The information about the components' collaboration is created as part of the entirely separate application layer.

1.2.3 Coupling Due to Event Data

Events not only have names, they often also carry data, or payload. Complex types being passed from one component to another, albeit through translated events as described in the previous section, leads to coupling. The event payload data types must be understood by both the sending and the receiving components, and thus constitute coupling through shared code. The two components cannot be said to be independent and reusable.

The AMVC architecture facilitates a translation between event payloads. To avoid coupling one component to another through a shared data type, each component event specifies its own payload type, and the application layer provides for the declarative mapping of one into the other as part of application assembly.

1.2.4 Coupling due to the Containment Relationship

The third and final coupling force at work in a rich client application is the containment relationships that exist: Dialogs have parent Frames, Buttons sit in Panels, and so on. Normally this relationship is expressed through code.

Through the *triad* abstraction in AMVC, this containment relationship is hidden from the Component code and expressed only at *application assembly* time. Both parent and child triads are aware only that they may be connected to other implementations of the triad abstraction, but they do not know *which* ones.

1.3 Application Assembly

1.3.1 Design Time

We saw in the previous sections how a component can be entirely decoupled from any other while still allowing them to collaborate. This section shows how that collaboration can be established in a rich client application *purely through declarative configuration*. The consequent potential for increase in productivity and component reuse demonstrates the value in achieving this total decoupling.

Application assembly consists of (i) arranging component instances into a hierarchical organization, (ii) connecting output events of some components to input events of others, through their translation into application events, (iii) translating event data (payload), where necessary, from that produced by the emitting component to that expected by the receiving component.

The hierarchy serves to establish the parent-child relationships between the v*iews* of any two given components. Note that a triad's view can be decomposed into a number of named areas. For example the view can contain a number of panels each capable of containing a child view. In this case, the area names are exposed as part of the public interface of the triad. When a child relationship is established with another triad, the relationship includes the name of the view and this creates a widget containment relationship between the parent's named View area and the child's view. The details of the way in which the views combine depends on the *view* types, and can be deduced automatically by the AMVC implementation, or can be prompted by the AMVC declarative description

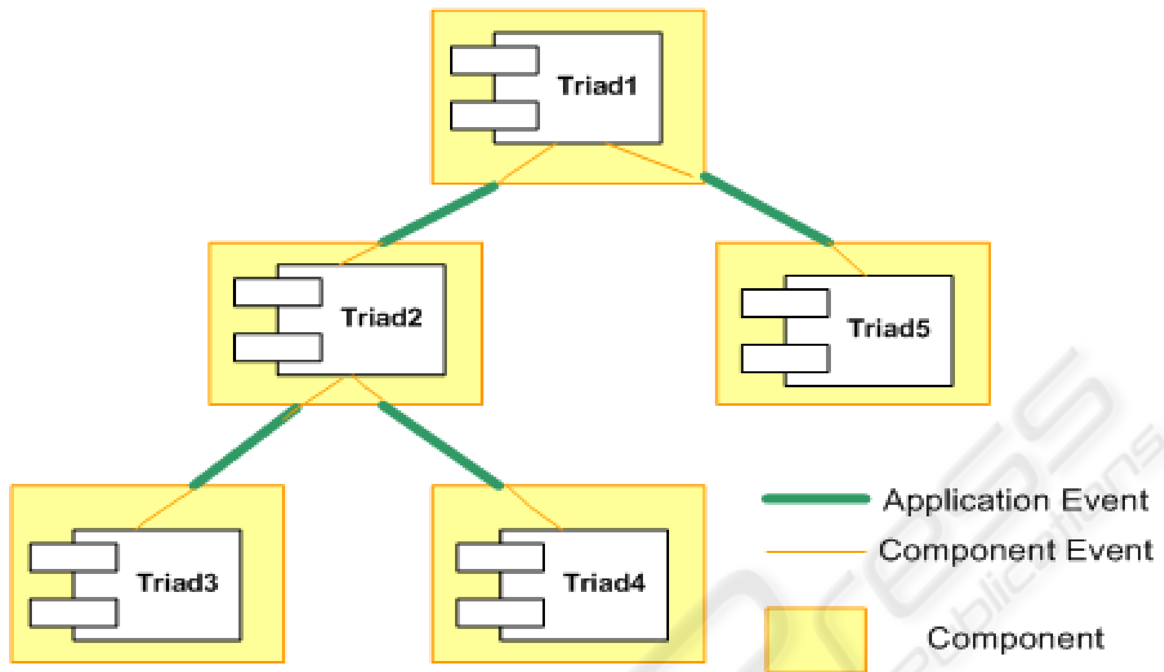

Figure 1: Translation of Component Events into Application Events.

of the inter-component relationship (e.g. the relationship can specify a dialog child).

The hierarchy also establishes lines of communication between components. *Application Events* can move between components across these parent-child connections. There is no impediment to routing Application Event between two components directly (as opposed to using the hierarchy), and this might make sense in a lot of cases, but the hierarchy provides a good first option.

Application assembly can be done mechanically and indeed visually if supported with the right tools.

1.3.2 Runtime

Initialisation of an AMVC application can be organized in any way that the application developer sees fit. That said, it makes sense that the process be started by the propagation of a *startup* Application Event to all Triads, through the hierarchy. The implementation described in the next section assumes that all triads are instantiated before that point, but another implementation may allow for a lazy-loading approach.

This event can be wired to Triad events as part of the normal application assembly. Each Triad type can offer its own incoming initialisation event and prepare itself for operation in whatever way makes sense for that Triad.

It is worth mentioning that no noticeable performance penalty should be paid by this architectural approach, and indeed none has been noticed in the implementation.

2 AN IMPLEMENTATION OF AMVC

This section outlines some of the salient points of the AMVC implementation.

The Proxy design pattern[vi] was employed in the design of the Controller's event handling mechanism. At configuration time, a controller has no guarantee that it will have been connected to a View or Model before having its events configured (in fact these View and Model elements can be considered optional, so the controller may never be connected to either). The same goes for the parent and child Triads that a Controller may eventually be connected to. To deal with this problem, Proxy objects take the place of the destination Model, View or external parent or child Triad Controller at configuration time. For example, a Model Proxy 'remembers' that its event must be directed at the Model. At runtime, when events actually arrive in the Controller, all connections to the Model, View and external Triad Controllers have been made, and the Proxy objects are used to complete the routing.

Unnecessary programming configuration is avoided by using coding conventions[vii]. In both

Model and View, the methods use naming conventions to ensure that component events are forwarded to those methods. Java Reflection is used extensively to handle events. Models or Views that wish to handle a particular event need only implement a method whose name indicates the Component Event name. A layer of GUI components that use the Triad code but provide general rich client functionality is included as part of this AMVC implementation.

Dependency Injection[viii] is another important design pattern used in this implementation of AMVC. The power and importance of this pattern in providing for decoupling has been documented in many other places, and has led to the development of a number of frameworks specifically to support its use. We have used the Spring Framework[ix] in this particular implementation for a number of reasons, principal amongst which is its new custom namespace feature. A key feature of AMVC is the declarative nature of the application assembly. Whereas components are composed of both code and declarative descriptors, the application layer is entirely declarative. The customisable nature of the Spring Framework's XML configuration made it the ideal way to combine components without coupling them, while providing a terse but readable declarative configuration format.

An example of this custom Spring-based format is provided in Listings 1 and 2 below to give an idea of the ease with which applications can be assembled within an AMVC architecture.

```
<amvc:appEvent id="appLogIn"
               name="LogIn"/>
…
<amvc:triad id="appLoginTriad"
            type="loginTriad">
   <amvc:terminate
       appEventRef="appLogIn"
       compEventRef="compLogin"/>

   <amvc:emitToParent
    appEventRef="appLogInSuccess"
    compEventRef="loggedInLogin"/>

   <amvc:emitToParent
      appEventRef="appExit"
      compEventRef="cancelledLogin"/>
</amvc:triad>
```

Listing 1: Declaration of event and login triad.

The above is an example of the declaration of an application event of id appLogIn followed by the instanciation of a component of type *loginTriad*. This triad instance (a component which provides basic username and password login functionality) is declared to capture the appLogIn application event, translate it into its own internal *compLogin* event, and direct it inwards to be processed as a *compLogin* event.

Similarly, two of the internal *loginTriad* events are emitted as application events – in both cases being directed upwards to the parent triad. The declaration of that parent triad, which happens to be the root or main triad of the application, looks like this:

```
<amvc:mainTriad type="mainTriad">
   <amvc:terminate
      appEventRef="appLogInSuccess"
      compEventRef="logInSuccessful">
      <amvc:payloadMapper
         targetClass="LoginResult">
         <amvc:map
            sourceField="user"
            targetField="username"/>
         <amvc:map
            sourceField="pw"
            targetField="password"/>
      </amvc:payloadMapper>
   </amvc:terminate>
   …
   <amvc:dialogChild
      ref="appLoginTriad"/>
</amvc:mainTriad>
```

Listing 2: Declaration of main triad.

The above section of the application assembly declaration demonstrates a number of important points.

Firstly, from the terminate element, we can see that the *appLoginSuccess* application event (emitted from the *appLoginTriad* from the previous listing) is consumed by the main triad, having first been translated into the main triad's own *loginSuccessful* event. In this case, the termination and translation of the event requires a mapping of the event payload. The event emitted by the login triad includes a payload instance made up of two fields called *user* and *pw*. The main application triad which terminates the event expects a payload with two fields called *username* and *password*. AMVC allows for the declarative mapping of the source payload object into the target payload object, as outlined in section 1.2.3 above.

A second point is the *dialogChild* element of the main triad's declaration. It is in this way that the parent child relationship between the main triad and the login triad is established. The View elements of the two triads are combined without using code. Moreover, though the child triad's View element has been added here as a dialog, AMVC could just as

easily have added as a panel bounded by the main triad's View element.This detail is important in promoting real component reuse.

Note that these examples have demonstrated how a declarative Application Layer can weave together reusable elements of a Component Layer in such a way as to completely avoid the coupling that comes from Event Sharing, Event Data and the Containment Relationship. This approach provides a template for a *divide-and-conquer* approach to rich client application development.

3 COMPARISON WITH HMVC/PAC

While the Triad of AMVC is based on hMVC, the way in which AMVC Triads communicate with each other is new. hMVC/PAC allows event names and event data generated in one Triad to travel to any other Triad in the application. AMVC uses the extra Application Layer to capture and convert both event names and event data, as part of routing those events from one Triad to another. Applications following the AMVC architecture consist therefore of a set of truly decoupled Triads in one layer, declaratively bound through event routing and mapping by the Application Layer.

Put another way, the architectural description of hMVC stops at the Triad. Because the hMVC Triad is responsible for its own communication with other Triads, and because its event names and data must thus be shared with other Triads, the hMVC Triad loses it re-usability and component nature. AMVC architecture describes not only the Triads, but the Application Layer than connects them without coupling them.

4 CONCLUSION

An AMVC component encapsulates the entire View and Model of a business process, and its interface is specified in term of business events. The approach taken by AMVC eliminates even indirect coupling between components and its extra layer allows components to be easily recombined and reused. AMVC can be applied to any component-based, event-driven presentation layer, and so can be used for desktop clients and Rich Internet Applications alike.

REFERENCES

Cai, Kapila and Pal (7-21-2000) HMVC: The layered pattern for developing strong client tiers. *Javaworld* (http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc_p.html)

Coutaz, Joëlle (1987). "PAC: an Implementation Model for Dialog Design". H-J. Bullinger, B. Shackel (ed.) *Proceedings of the Interact'87 conference, September 1-4, 1987, Stuttgart, Germany*: pp. 431-436, North-Holland.

Fowler (2001) Reducing coupling. In *Software IEEE, Volume: 18, Issue: 4.*

---

[i] http://www.laputan.org/mud/

[ii] One idea of what defines a software component can be got by reading Szyperski and Messerschmitt's list of their characteristics: Multiple use; Non-context-specific; Composable with other components; Encapsulated; and a unit of independent deployment and versioning. To the last point, one could add the word 'purchase' – the component can also be seen as an economic unit of software.

[iii] Jason Cai, Ranjit Kapila and Gaurav Pal, JavaWorld.com, 07/21/00. HMVC: The layered pattern for developing strong client tiers.

[iv] http://en.wikipedia.org/wiki/Presentation_Abstraction_Control

[v] Service Oriented Architecture: http://en.wikipedia.org/wiki/Service-oriented_architecture

[vi] http://en.wikipedia.org/wiki/Proxy_pattern

[vii] http://blog.decaresystems.ie/index.php/2006/01/13/convention-over-configuration/

[viii] http://www.martinfowler.com/articles/injection.html

[ix] http://www.springframework.org