6.1 Composite Frontend (Portal)

When we try to think about service consumers, the obvious candidates are, of course, other services. Nevertheless there are other software components that interact with services e.g. legacy systems, Non-SOA external systems or reporting databases. The Composite Frontend pattern deals with yet another type of service consumer – the User interface.

First let just verify that User interfaces aren't in fact services. One reason user interfaces are not services is that they converge several business areas e.g. if you want to enter an order you'd probably also want to lookup information about the customer, maybe you'd also want to browse the product catalog, look at open invoices etc. In addition to convergence, user interfaces deliver data rather than process it. User interfaces are data producers (actually there's one exception to that – where the UI is the front of a "human service" see orchestrated choreography pattern (in chapter 7) for more details.

Ok, so UIs aren't services, does it matter? Well, it does and the problem is not that UIs aren't services per se. The main challenge caused by user interfaces comes from their main difference i.e. the aggregation or convergence of several services into a cohesive and useful UI.

6.1.1 The Problem

To better understand the challenges caused by user interface working with multiple services, let's consider an example with just a single point of friction.

In a project I worked on we've designed the C4ISR (Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance) system for an Unmanned Naval Patrol Vehicle (UNPV). One of the services in the system was dubbed "Common Operational Picture" or COP for short. The COP 's responsibility was to handle anything that is detected by sensors e.g. ships, planes, whatever (There's technical jargon for all that like targets, detections, tracks etc. but that's not important here). One of the main UI representations of the COP was a map ,such as the one in figure 6.4 below, which showed all the detections. Clicking on map icon, say a ship, presents some information the COP knows about it, such as id, nationality, course etc.

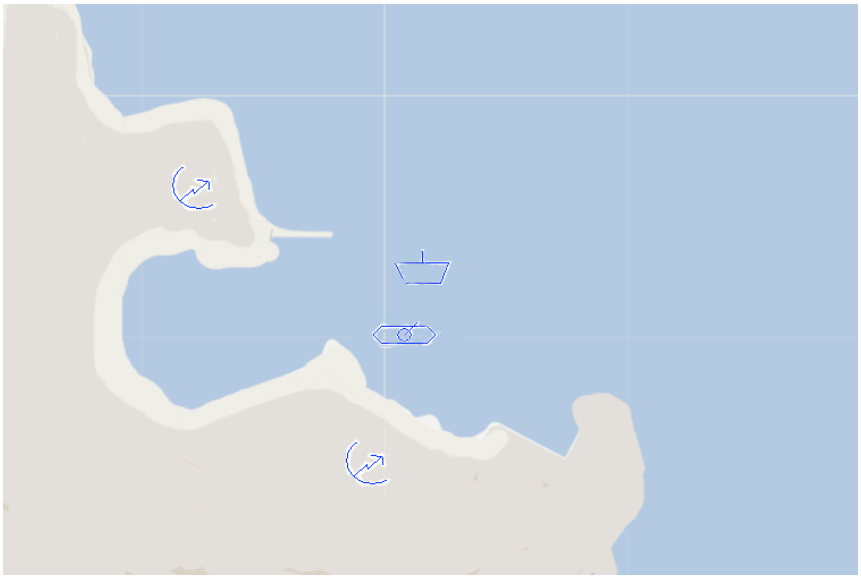

Figure 6.4 A simplified illustration of a front end for a "Common Operational Picture" service of a naval command and control system. We see a shore line and then using NATO symbology: 2 radars, a submarine and an a ship (the Unmanned Naval Patrol Vehicle)

The system had a few other services in addition to the COP, amongst them there was the "UNPV service". The UNPV service was responsible for anything related to the UNPV itself for instance, setting it on a navigation course, turning it around, etc. The UNPV service had several UI screens to allow managing and monitoring these functions. Another responsibility of the UNPV service was to send its location to the COP (locations are the COP's responsibility, remember?) so back in the UI, one of the icons of the map is that of the UNPV.

What happens when a user clicks on the UNPV icon on the map? Remember, while the desired outcome is to display a popup with options related to controlling the UNPV, the click is on the map, a control serviced by another service (the COP). In an Object Oriented system the UNPV might be a sub-class of a detection so that it would accept the same event and respond a little differently (in a more specialized way). Here, however the COP and the UNPV are completely different services, developed by two different group and maybe even two different companies.

We may be able to dismiss this specific example and just solve it with a specific solution e.g add an 'if' statement somewhere to call up the correct commands and to interact with the correct service(s). The problem, however, is that this example is just the tip of the iceberg. For instance: how do we handle security? – do we need to login for each service separately? How do we handle things that all the services need? SOA's premise is that we'd get a sort of lego-like enterprise where we can compose different business processes easily. Is there any way we can get that in the user interface? In summary:

> **How do you we interact with multiple services, get an integrated, cohesive user interface and still preserve SOA principles and modularity benefits?**

One option is, as mentioned above, write specific client code. Using this approach "an application", is any specific composition of services. For the example above, the application would include the two service (COP and UNPV) and a UI that ties them together. The up-side of this approach is that each application delivers a consistent experience for the user. After all, a specific or tailored application can be made to be very cohesive. Additionally there are many tried and tested ways to build flexible UIs with a proper separation of concerns, e.g. using Model-View-Controller and its variations (in a multitude of rich client and web technologies) so we'd probably be able to reuse some of the UI-side logic even going from application to application. Nevertheless, we do lose on flexibility. For one, any service change that has UI aspects needs to be redone for each of its UI instances (applications). More so, since the UI specifically ties multiple services changes in one service may cause another to mal-function within the unified UI. We also lose on Composability, or the ability to replace services and to create new business flows (relatively) easy. Overall it's a bad option long-term but it can be made to work as a short term solution.

A related option is taking a similar approach of tying several services together but instead of integrating them on the client side we integrate the services together on the server side. This approach shares it pros and cons with the previous solution. However there are specific circumstances where it does make sense and you can read about them in the next pattern (section 6.5 Client/Server/Service)

Lastly, we have the option to have independent UI components per service. This would overcome the limitations we've mentioned above since each service's corresponding UI can evolve independently and you can just cram a as many of these as you like to create an application. Unfortunately, with all its benefits this is a non even an option here as, by definition, we won't have the mechanisms for UI components that work cross services. i.e. it won't actually solve problems like the one in the example and we can't get a cohesive UI.

6.1.2 The Solution

What we need, essentially, is a way to compose services together while keeping their respective autonomy on one hand and providing mechanisms to glue them together as a cohesive whole – That's what the Composite Frontend pattern is about:

> **Apply the Composite Frontend pattern to aggregate services while providing them unified client-side services like layout and theming as well as coordination services for client-side service integration**

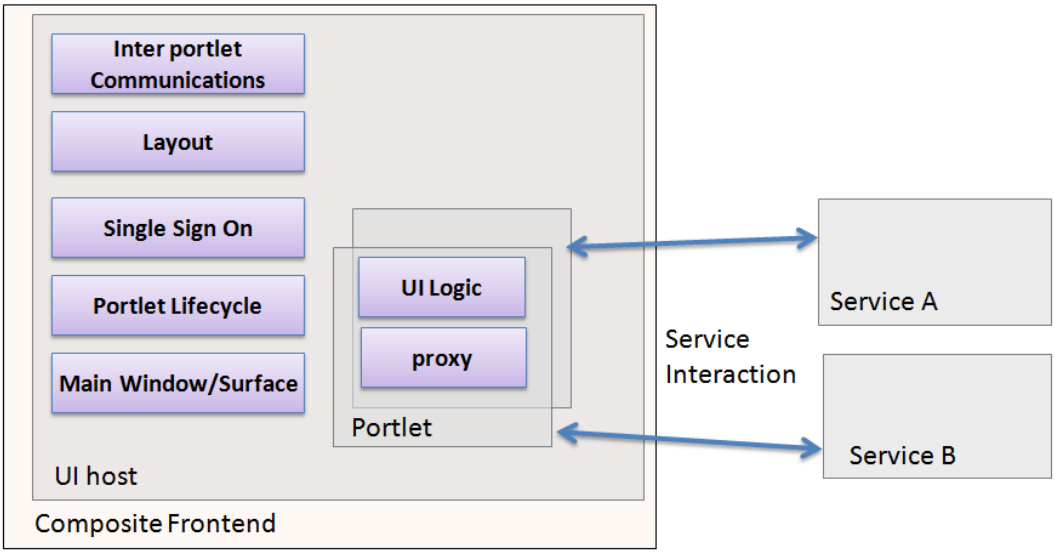

Figure 6.5 The Composite Frontend pattern. Each Service has a Portlet which is a Service Agent combined with a UI logic (most likely Model in a MVC UI pattern). The UI host provides services for the different potlets to weave them together into a coherent UI.

The Composite Frontend pattern is about taking the ideas (and sometimes the technologies) behind web portal and applying them to SOA services. Web portals provide unified access point that aggregates multiple web pages. They also provide single sign-one and personalization. SOA interfaces need that and more.

The composite front-end pattern is composed of two main components: the portlet and the host. The portlets are the building blocks, "the composites" which are fused together to form the user interface. The portlets are made of at least two components. The user interface logic (views and controllers in MVC lingo). The second component, the service proxy (or agent) , is the more interesting one from an SOA perspective. The service proxy is a client-side representation of a service. The proxy serves as the model for the user

interfaces components. It is usually recommended to have a single proxy per service -just like it is recommended for each service to maintain its own data store. Furthermore, *from a product management perspective it can be seen as part of the service itself*

The host is the "value-add" part of the composite frontend pattern. The host provides the glue that ties the different portlets into a cohesive whole. As such, the host provides several roles. Firstly it provides the canvas or surface on which the portlets are displayed. Additionally it controls the life-cycle of the portlets and lastly it provides capabilities (avoiding the loaded term services...) like inter-portlet communications and single-sign-on.

Lets revisit the problem discussed in the previous section. We had a right-click on a ui component, which should have produced a context menu with options from two services. How would that would with the composite frontend pattern? One option is that a click would be first intercepted by the host which would then dispatch it to any registered portlet. Another option illustrated in figure 6.7 below , is for the click to be intercepted by the first portlet the Common Operation Picture (or COP), have it notify the host and have the host ask all the iixnvolved portlets to render the right-click menu. The COP portlet should pass enough information as part of the event for the other porlets to be able to do something meaningful with.

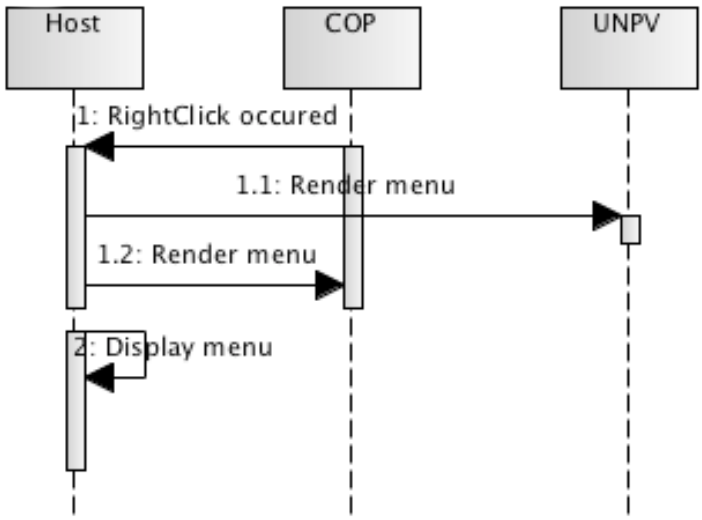

Figure 6.7: Sample event flow in a Composite Frontend. Events are intercepted by the user interface

components of individuals portlets , The events are transferred to the host which dispatches them to registered protlets for handling. The host can then render the results for display

7

The composite frontend pattern is a service consumer pattern so the proxy will utilize the various service interaction patterns like saga, request/reply etc. (see chapter 5) and it can benefit from the various service composition patterns like Service Registry and Servicbus (see chapter 7)

You've probably notices the use of the term portlets to describe the service agents and you might be wondering why the pattern is named Composite Frontend rather than Portal. The main reason for that is that the pattern can also be used with rich client implementations and not just web ones – let's explore that further in the technology mapping section

6.1.3 Technology Mapping

Normally, you'd won't be developing your own Composite Frontend container and instead use existing products that provide the framework and usually the tooling to help build the portlets.

The obvious example for that are web portal frameworks. Modern enterprise web portals usually support anything from JSR 168/286 (Java Portlet specification) to WSRP (Web Services for Remote Portlet) to open web standards like RSS, plain REST services or standards like open social. There are a lot of products in this area both commercial like ibm WebSphere portal server and Microsoft Sharepoint and open source options like Jboss Gatein and Lifray. Figure 6.8 below show he layout functionality of the UI host as it is implemented in Jboss Gatein

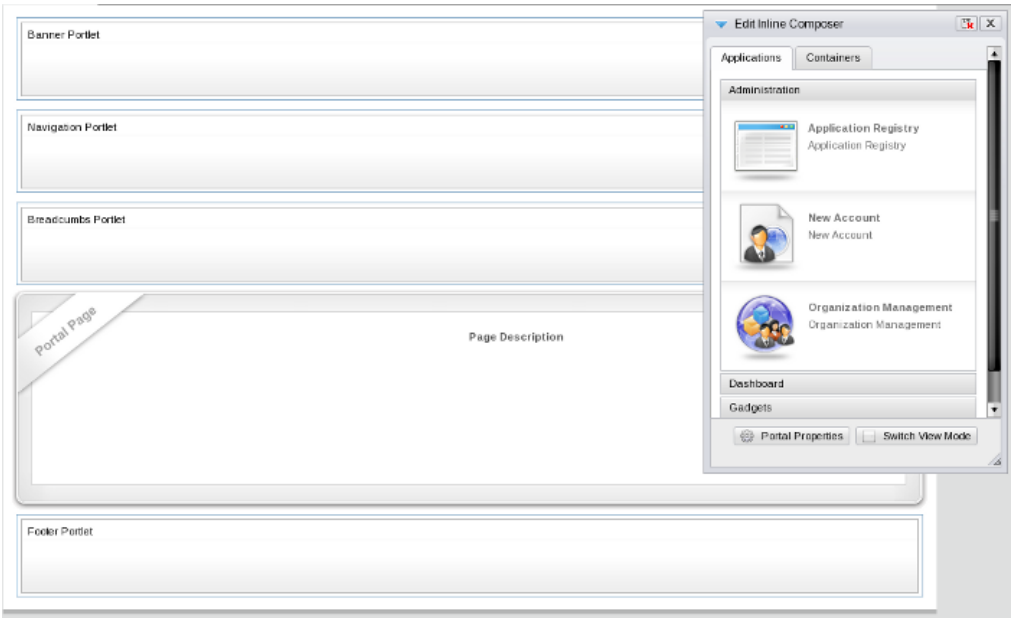

Figure 6.8: The layout capability of a Composite Frontend UI host as it is implemented in Jboss Gatein portal.

Web portals are not the only option for implementing composite frontends. You can also implement the concept for desktop ("rich client") applications. An example for that is the prism framework made by Microsoft's Pattern and Practices group. Prism implements the Composite Frontend pattern for both Silverlight and WPF applications. Prism provides all the functionality of a user interface host and lets you write portlets that consume these capabilities. Code snippet 6.4 below demonstrate using an EventAggregator facility that allows inter-portlet communications (such as the one needed for the sample scenario in the map component example above):

Code listing 6.4 Sample use of Prism's EventAggregator to send events between different porlets unified by the prism shell

```
[Export(typeof(SampleView))]
  public partial class SampleView : UserControl
  {
      [ImportingConstructor]
      public SampleView([Import] IEventAggregator eventAggregator)
      {
          InitializeComponent();

          eventAggregator.
```

```
                GetEvent<CompositePresentationEvent<ItemSelectedEvent>>().
                                Subscribe(OnItemSelectedReceived); #1
        }

        public void ItemSelectedReceived(ItemSelectedEvent item)
        {
                //do something with item...
            }
    }
```

#1 subscribing to an temSelectedEvent. Another portlet can call get on the EventAggregator to get a reference to the event and raise it without knowing if thare are any subscribers

Lastly, in addition to web portal frameworks and desktop framework you can roll your own implementation of Composite Frontend. however, as mentioned above it is usually better to choose one of the available options as it quite an investment to get it right.

6.1.4 Quality Attributes

Before moving on to the next pattern lets examine some business drives (or scenarios) that can drive us to use the Composite Frontend pattern.

In essence, the main drives to Composite Frontend are flexibility to adding and changing services and the desire for an integrated user interface that feels as a whole Table 6.X provides examples for both quality attributes:

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Usability	Operability	Under normal system use end user wants to achieve business tasks fluently. System should reuse entered data (like personal details) between different tasks
Flexibility	Changability	Under normal conditions, changing the billing process to support a new credit card clearance provider, should take less than one week

Table 6.3 composite frontend pattern quality attributes scenarios. These are the architectural scenarios that can make us think about using composite frontend pattern.

Composite Frontend is probably the preferred way to provide an SOA user interface. However one problem we still have to solve with integrating UIs is UIs that are not SOA aware – for instance what happens when we have an existing UI that we want to expose to services? The next pattern will try to answer exactly that.