openKura

TechnicalPrimer

OpenKura v0.2 / VPatch 0.2
carrotsrc.org

2014-02-07

Interaction Process

OpenKura and the engine it runs on – VPatch – is a highly abstracted environment; this means everything from the code, through the functional implementation, to the high level web objects are all as integrated as possible within the model in an attempt to keep everything consistant.

Let us step through the model of interaction:

The original inspiration for the model was audio production software where a signal is manipulated from the input to the output- this is why there is a signal, channels and jacks within the system but don't get too stuck on the audio analogy. A signal is essentially a form of user interaction and it's data, which will be channelled to it's directed end point. A channel is made up of plugins that read and modify the signal's data. Keep with it, it will make some sense soon.

Panels and Components

So here are two main parts of the system- a panel and a component. A panel is the front end interface and it has a corresponding component. When a user interacts with a panel, in the abstracted sense, it is sending a signal to the component which modifies the signal as output. This signal is not only directed at the component but also one of its jacks. Now, a jack is a connection on the component that accesses a specific type of functionality. You could think of them as ports accessing services; So for example, you might have a Board component that 'adds a post' and 'looks up all the posts'- these two functions will be accessible via their respective jacks. At the time of writing the jacks - like ports - are distinguished by a number.

Instances

There is not just a single version of a component (that would be quite restrictive); a little like a service, a component is instantiated. This means you have many different versions of a component within the system, each with access to its respective data. To take the Board example- a single instance of Board will recall all the posts based on it's instance.

Layout

Now, how does a user access a panel to interact with a component? Through a layout. A Layout is a combination of panels, each “wired” to interact with a specific instance of a Component, which gives the Panel the context it needs. So if you have the Bulletin Board component and each instance is a specific board for a topic, a panel will be “wired” to interact with an instance of a bulletin board, with all it's respective posts in that topic, which in turn allows the panel to be the front end of that specific board.

Areas

To start to push the audio analogy beyond breaking point and more into computing realm - how does a user access the layout? Via an Area. An area is a bit like a domain (if you're reading this I'm sure you are familiar with a domain) which contains a bunch of layouts and/or other domains. A user directs themselves to an area which has a specific purpose within the site. A bit like a web domain, an area can have a default index layout with any other layout needing to be specified.

Application

The bit that pulls these parts together is the App object . When a user interacts with the application, they are accessing it through several different channels which each have a different function. So you might have the “_on_page_request” channel that is the normal channel for when a user requests a page. Another may be “_on_asset_request” which is the channel used for requesting a particular asset in the system.

Follow the Signal

Let's try and bring it together by following the 'signal' (in an abstracted sense), based on a generic page request. When a user goes to the index of a VPatch system, the App object is instantiated. The app channels the signal down several channels, each channel is made up of a series of plugins. At this stage you can assume the core engine doesn't know anything beyond generating channels and directing the signal down them - everything else is implemented via plugins.

The panel and component communication is abstracted the same whether on the server or via asynchronous JavaScript. This allows the functionality to be the same whether the user has JavaScript or not. Again, it's all part of an integrated environment.

So the user has requested an area. A plugin could read data from user input provided and generate respective parameters for the 'signal packet' which is then passed onto the next plugin; in this instance a plugin has read the location and generated the area and layout parameters in a way that future plugins understand, and placed them in the packet.

In the example channel, the next plugin in the series is an Area loader- it reads the 'signal packet', gets the area parameter and checks what channels – if any – direct to the area. This is fairly critical- an area can have a channel, which itself has a bunch of plugins, where the signal is passed down. The channel could have a series of plugins to see if the user is logged in or in a certain group, which can block the signal if unsatisfied. If the signal is not blocked, it is passed into a layout loader plugin, followed by a layout generator.

Now the layout generator will load the instances of the components that the panels (in the layout specified by the parameter from the first plugin) are wired to. Each instance of a component might have it's own channel to modify the 'signal packet'- maybe blocking it or modifying some form of input. Then again, it may just be a direct link to the instance with no channel. Once the signal is passed through the channels, – and everything is satisfied -- the components are fully instantiated within the system. In abstract terms- this means the user has full access to that instance of the component. The next plugin deals with direct signal between the panel and the component.

The idea of a fully instantiated component is critical because this is where we can shift more towards services. When a component is fully instantiated and is accessible to a user, it is a bit like a service running in a domain or in this case a component running in an area. At that point the channel needed for signalling the component is a fairly direct one. On the other hand- if it's not instantiated, there is nowhere for the signal to go.

So finally the panel signal is delivered to it's respective components. It doesn't quite work this way in the code, but we're talking in terms of the abstraction. The signal is handled by a plugin, directing the panel signal to the component for processing and redirecting the output back to the panel. We'll go into this in a bit more detail later. For now, the component has modified the signal and the results are used by the panel for feedback. Fairly standard I/O. However, the point is, from the view of a panel all you're doing is sending a signal to a component, the component is sending a signal back, the rest of the processing is done by the configuration of VPatch and plugins.

Finally the page is generated by the page generator plugin. The panel feedback is rendered within the layout and the resultis dispatched to the user as an HTML document.

Essentially that is VPatch's basic powerhouse for interaction. It has a few more bits implemented on top to enable the model to be more dynamic but generally everything at this level is done via plugins. One thing to also note is that plugins are instances as well; this becomes useful when you have the same functional plugin requiring different attributes in a different context but that comes later. For now, this section of the primer is what you need to get onto the next bit.

Resources and Networks

The first part of the primer introduced the abstracted process by which interaction is processed into feedback. Now the primer will delve into resource networks. It will seem disparate at first but hang on in there.

In Vpatch, as many things as possible – from plugins, through channels all the way up to users and comments - are represented in the system as generic “Resource”. This abstraction allows the system have a basic recall of all the objects it contains. The resources can then have associations with one another, which enables the system to recall relational information.

What it boils down to is the entire site structure and user generated data is a huge graph (or network, as it shall hence forth be referred to) of nodes (resources). Some connections (relationships or associations) and nodes are used for signal processing, others are used contextually. The core system doesn't care what the configurations are, it will use the bits it knows and recall the bits requested.

Every interaction the user makes can be represented on the graph in the context that it was made. This is very useful for OpenKura because it potentially creates a rich environment for a student as they generate their own network of information through just using the system. It also allows for open ended relationships between different resources.

Structure of a Resources

A resource is made up of several different components- a base, which defines a descriptive common group, it's type which represents the specific form of resource, it's label which is a textual representation and a handler reference which is the link to the actual data it represents.

This is the notation for a user resource:

```
User('zoe') => 2
```

This is saying theres a User type resource, the label is 'zoe' and it's handler reference is 2.

The User type has the following notation:

```
[entity]User()
```

This is saying the resource type User has the base of entity.

Finally a relationship, or association, between two resources can have an extra dimension- an edge. The edge gives a description of the relationship beyond a generic parent-child connection. This allows the system to be far more specific. Andedge is based off the parent type in the relationship. So a user can have an edge to say it created a child resource. The notation is as follows:

```
User():spoken
```

At this stage, this is enough to know about resources. Lets see how they relate to the different fundamental parts of the system.

Components and Resources

As components are used, what they're doing is creating their own meaningful sub-networks of resources. The component is self-contained in that it only understand how to manage it's own resources and is totally unaware of the wider network; Beyond that, it has no idea about the database (which is handled by it's library). It is able to understand relationships between it's resources, create and destroy resources, while acting as a pipeline for the actual data that is finally handled by the library. This creates a very clear separation of functionality.

Libraries and Resources

In general a library manages the actual data. It has no idea what a resource is, all it receives is the reference it needs for a particular bit of data. This reference is what the component understands as a resource, which itself has a place on the wider network. Sometimes libraries don't manage data, they might implement particular functionality but that is not relevant.

Plugins and Resources

Now, a plugin – like a component – can understand meaningful configurations of resources and sometimes create relationships but in general it won't create resources. It's fairly open ended as to how it understands resources depending on it's purpose. A plugin might check the relationship between a user resource and an instance resource of a component to see if they are linked; if there is no relationship then it may block the signal. Another plugin may take what a component has generated and create meaningful relationships between the newly generated resource and other resources.

Panels and Resources

panels – like libraries – have no clue about resources. All a panel understands is how to interact with a component and produce feedback.

Subsystems and Resources

A subsystem is essentially a working, self contained system running on VPatch; there's no strict definition, it's easier just to point at working groups and class them as subsystems. It could even be groups of subsystems. A subsystem may have a configuration of areas, with layouts and panels, managed by a component or a group of components; it could be a totally abstract sub-network but what they all have in common is the management of, and meaningful relationships between, resources. Because it is a network of resources it is very easy to understand a subsystem visually. Let's bring the last few topics together in a worked example.

Let's design a bulletin board subsystem. At this stage we're discussing resources and their relationships.

Our bulletin board subsystem will first have an area which will have two purposes: to be somewhere the user directs their signal and also a root node for the rest of the system. Within this area, each topic will have a layout; each layout will have a side bar panel that will point to all the other layouts in the area and also have a panel that will interact with the instance of the bulletin component that represents the topic.

There may be another component that acts as an index for all the bulletin boards- let's call this the BulletinIndex. Each instance of BulletinIndex has a bunch of instances of BulletinBoards associated with it, which are used as the contents of it's index.
Each BulletinBoard instance has a Layout associated with it which defines where it is represented. This is used by the BulletinIndex to know where to point the user.

When the user goes to the area, one Layout is defined as the area index which is the default one. This will hold the panel for the BulletinIndex which will show the user all the bulletin boards in the subsystem.

So now let's move onto an instance of the BulletinBoard. When a user makes a new post, the data will be handled by the library but the actual component will generate a resource for the Post and associate it with the resource of the BulletinBoard instance. Now the post has a context within the sub-network we're creating.

The following diagram represents the subsystem which can be implemented rapidly. Most of the subsystems in OpenKura were originally designed like this before implementation.

Now this subsystem can have a place in the wider network. The bulletin area may be contained within a parent area. The bulletin area may also have a channel attached to it which will block the signal of a user who is not logged in. The instance of the BulletinBoard component may have a channel attached which will take the Post resources generated by an interaction and create a relationship with another resource giving those posts a different meaningful context.

Implementation

We'll move from the theory and design parts of the VPatch to different aspects of implementation.

Resource Queries

As previously stated, the component doesn't really understand the concept of a database but what it does do is interact with the Resource Manager. Through this is can manage it's resources but make special queries called Resource Association queries, or RQL statements. This is an abstracted query syntax with no superficial relationship to the underlying database query language. This is how components and plugins test relationships. It also pulls out unique Resource Ids or RIDs.

Let's see some examples of RQL statements but remember they are dealing only with resources that are representative of the data handled by libraries.

In the spirit of OpenKura- let's find all the users that are in a course:

```
User()<Course('Biology');
```

This is saying "Find all user resources that are children of the biology course"

There's a base that represents all user made content- expression. This covers posts, comments, quiz results. We can find all resources of base [expression] associated with (subject) modules within a course:

```
[expression]<(Module()<Course('History'));
```

This is saying "First find all the Module resources in the History course and then find all the resources of base expression within those modules". If you're familiar with any query language you'll see the sub-query.

A component might need a bit more information than the RID. So here we get the handler reference of all modules in a course:

```
Module(){r}<Course('Sociology');
```

This is saying "Find all the Module resources and their handler references within the Sociology course.

You have two resources, their handler reference and you want to test their relationship:

```
User('4')>Comment('8');
```

This is saying “Is User with handler reference 4, associated with Comment with handler reference 8 as a parent”.

Finally, let's incorporate the edge into a query. You might want to find all resources that have a base of [expression] that were 'spoken' by a user (for some reason, a user may be associated with a resource but not be the original creator). This is handled in the query by specifying an edge:

```
[expression]<User('12'):spoken;
```

That wraps up the basics of RQL statements. It's simple but effective.

Compromises

Already you may start to see some pros and cons. A pro is the resource network, through simple mechanisms, is creating a very rich landscape of contextual information. However a con is the vastness of the network. There is much done to create as many unique qualities to a relationship as possible for a more directed lookup but design consideration requires the pure model be compromised at some stages. For example, the posts of a bulletin system may also have a reference to the instance they are associated with in it's actual database table which allows for a quick recall of the posts without needing to refer to the resource network- the relationship between the representative resources is implied. However, at the same time the resource for the post is in the network to keep it's context consistent with the wider system.

It may be that a subsystem only represents structural parts as resources (used for signal processing; areas, instances etc.) but doesn't generate resources for posts. The down side is this reduces the richness of contextual information.

Sometimes it's necessary for a subsystem to be represented by a single resource, to merely give it structural context within the wider system- this could be an instance of a private message inbox. Here it makes sense that the instance is associated with a particular user but the private messages aren't accessible to the wider network; this reduces the load on the network as a whole.