# Behaviour Chain Pattern / Russian Doll / BMVC

## Why is it Different to CoR

*It is a CoR in form, but not necessarily in intent. CoR is usually used to   
find a class that can process a given request.*

*In Fubu Behaviors, it has the same form as a CoR, but the intent is not to   
find a responsible class, but to allow many classes in the chain to   
participate in the processing of a request (to add context and information   
to the request, or to process the request and contribute to the output).*

*So in that respect, it's a decorator pattern.  You might call Fubu   
Behaviors a "Chain of Decorators" pattern since it has elements of both   
patterns.  In fact, we originally called them decorators until Steve Harman   
suggested we use the term "Behaviors" and it stuck.*

*-Chad*

## Slides Plan

1. Slide showing a diagram – not necessarily UML
   1. Also known as Russian Doll Model , BMVC
   2. Similar to CoR in implementation
   3. Similar to decorator in implementation
   4. Differs by intent
      1. Explain how the intent is like a pipeline
      2. You go to the end of the pipeline – then you come all the way back up
      3. Behaviour controls the invocation of the next behaviour, not defers responsibility to it
2. Configuration vs Runtime
   1. Configuration time is like creating blueprints
   2. By having a set of chains can modify and manipulate them at configuration time
   3. At runtime the behaviour nodes will be constructed in the sequence of behaviours
      1. Converted into Russian doll model
3. Show the first demo
   1. Show the abstract behaviour
   2. Show the configuration – highlighting appending, and wrapping
   3. Show how the objects will be converted from configuration to runtime
   4. Show how we would expect the output to be rendered
   5. Show the actual output
4. Separate configuration model also allows conventions to be applied
   1. Can wrap all chains that talk to database with an nhibernate session
   2. Can catch errors and halt execution of the chain and respond in another way
   3. Can add actions to the end of the chain based on a given predicate
   4. The examples all use the type of the objects, but if a richer model of metadata is available, conventions can be based on that too
   5. Show the results are what we expected
5. IoC for channelling context and intelligent creation
   1. Demo is like a simple web request
   2. Regsiter some services with our container
   3. We’ll register an object that’s like a singleton, but is simulating an http-request scoped to a particular web request
   4. Don’t forget, we could add conventions here based on the type of output being rendered or headers in the request if we tweaked our model – this is exactly what fubumvc does
   5. Show how we’ve built up the chain like a an example request processing pipeline, but now we have behaviours that have dependencies and require the context object
   6. Show each behaviour’s implementation
   7. Talk about the expected output
   8. Show the output
   9. Point out how this allows us to test each behaviour and follow SOLID principles

## Links

* <http://www.google.com/url?sa=D&q=http://lostechies.com/chadmyers/2011/06/23/cool-stuff-in-fubumvc-no-1-behaviors&usg=AFQjCNFX0PV8FO5sE0kFkvB9UzTVqYJg-Q>
* <http://lostechies.com/josharnold/2011/02/01/fubumvc-primer-configuration-vs-runtime/>

## Configuration Model

**Wrap**

* “Policies” on the FubuRegistry
* Pass the type of behaviour into WrapBehaviorChainsWith
* \*\*\*\*\* Need to do more research here \*\*\*\*\*

**Enrich Calls With**

* addPolicy
  + Adds the **type** of behaviour to all chains (action calls) satisfying the given predicate
    - Adds it after the last action (Behaviour Node.Add After)
* **IActionBehavior**

## Runtime Model

IConfigurationActions are applied to setup the graph

**BehaviorAggregator**

* Finds all the actions
  + For each one
    - creates a new chain
    - Actions the actions to the end of it
    - Adds the chain to the graph
* **BehaviorChain**
  + Implement as IEnumerable and use decorator-style nested recursion to get each object in the graph
* **BehaviorNode**
* **IBehaviorFactory** “Asked to build the behaviour from the arguments and the chains unique id”
  + **StructureMapContainerFacility** – looks like the implementation
    - It just takes args and id of chain and wraps it in a custom implementation of IActionBehavior – **NestedStructureMapContainerBehaviour**
  + **BehaviorInvoker** then invokes the behaviour

**NestedStructureMapContainerBehavior**

**Invoke –** Get’s the appropriate behaviour and invokes it

* Get’s a nested structure map container – has all the original services from the original container
* Registers all the services with it from the passed in “arguments” of type ServiceArguments
* Get’s the instance of IActionBehavior using the behaviourid
* Invokes that behaviour

## Problem – How does the invoke partial work

* Object node returns a type
* Type is the behaviour
* Instance of behaviour is created
* Behaviour invokes partial
* How does behaviour get partial?
  + So the partial behaviour is just an enum telling what to happen when the partial is invoked
* The **“InsideBehavior”** is not explicity set in code – it must also be done by structuremap