Jetspeed Cornerstone Concepts

Jun Yang (junyang@cisco.com)

Emad Benjamin (ebenjami@cisco.com)

Disclaimer

This document does not serve as an introduction to the design of frameworks. It assumes the reader's familiarity with other frameworks or their concepts in general.

Purpose

Mass Customization

Mass customizability is the ability to meet new business requirements with minimal impact on existing code base. Cornerstone is designed to support *Mass Customization*.

Concepts

Interface

Jetspeed Cornerstone (henceforth simply Cornerstone) enforces *Design by Interface*. All APIs are interfaces. Relationships between interfaces are of interface types.

Implementation

Interfaces are implemented by classes. However, there is no direct reference to class names in Cornerstone code. For example, there is no `new MyClass()` or even `MyClass.CONSTANT` in the code. All references to implementations (classes) are indirect (through the Registry).

Comparison with Other Work

Cornerstone is built using its own concepts. All of its own core classes (such as `ImplementationManager`, `Registry` and `ServiceManager`, etc.) can be replaced with other implementations.

Implementation Variants

An interface can have one default or many variant implementations, each of which is called an implementation variant. All implementation variants are registered in the registry.

Configuration

Configuration is deployment-time (as opposed to run-time) data that can alter or control behavior of code. Configuration is also called customization. Cornerstone supports customization in four dimensions:

1. Component. All components (classes and instances) are customizable.
2. Relationship. All relationships between components are customizable.
3. Control flow. Control flows of components (e.g. services) are customizable.
4. Preservation of customization. All customizations can be preserved cross upgrades.

Registry

Registry is the store of all configuration data. Customization is always done to the registry. Registry has *planes*, each with a different priority. Values in higher priority planes overwrite those in the lower priority planes. For example, the out-of-box configuration would be in plane 100. Your customizations reside in plane 150. Someone else's customizations on top of yours reside in plane 200. Plane 200 overwrites 150 and 100 and so forth. It is very easy to insert a new plane or take away an existing plane anywhere in the chain.

Indirection

Registry also maps logical entities (e.g. the name of an implementation variant) to physical (e.g. the class name of an implementation variant).

Virtual Class

All implementation variants are registered in the registry. For example, the registry has:

```
Implementation
      com.mycompany.IA
            a1    // one variant
                  class=com.mycompany.MyA
                  configValue1=100
            a2    // another variant
                  class=com.mycompany.MyA
                  configValue1=200
```

Every implementation variant is like a virtual class. For example, for interface `IA` we have a class `MyA` that implements it. We can have two variants of `IA` `a1` and `a2`. Both `a1` and `a2`'s class is `MyA` and yet they define different configurations. So an instance created from `a1` differs from an instance created from `a2` only in configuration. Implementation variants `a1` and `a2` are like virtual classes. Singletons can be based on virtual classes.

Comparison to Other Work

A registry with overwriting planes that supports preservation of customization over time is unique to Cornerstone.

Bean

Business objects are implemented as regular JavaBeans that have no dependency on Cornerstone.

Bean Factory

Bean factories are responsible for creating storing and deleting beans. Factories are implemented by extending either `BaseFactory` or `InversionOfControlFactory`. The concrete type of the other side in a relationship is resolved by factory of the bean at run time through configuration and is not hard-wired in the bean itself. So the business object beans can be reused with a completely different framework.

Most of the time developers don't need to create their own factory classes. They will just create new configurations of Cornerstone's `InversionOfControlFactory`.

Comparison to Other Work

In Cornerstone, you can create an arbitrary number of factories for a bean. There is no special requirement on the constructor of the bean.

Inversion of Control

See *Bean Factory* above.

Context

A context is a list of name value pairs. Its most common use in Cornerstone is as a set of parameters passed into the `invoke` method of a service.

Service

A service is a component that supports the `IService` interface whose main method is `public Object invoke(IContext context). BaseService` is the abstract super class of all service implementations. A subclass actually doesn't implement `invoke()` directly but `public Object invokeDirect(…)` whose argument list can be arbitrary. At run-time, `BaseService` extracts the necessary arguments from `context` and populates the argument list of `invokeDirect()` through reflection. This way Cornerstone supports a generic interface of `invoke()` for all services while allowing a specific type-safe signature for each individual service at the same time.

Services are cached with a switch in the configuration. Developers don't need to do anything to get caching. Of course, cache parameters can be customized when necessary.

Comparison to Other Work

In Cornerstone, all services have the same generic interface `IService` and yet each implementation has its own specific `invokeDirect()` that is type safe. The very fact of all services conforming to the same interface makes it possible to put any services in control flows via service controllers (See *Service Controller*).

Service Configuration vs. Parameters

Service configuration and parameters are unified and handled the same way in Cornerstone (all declared on the argument list of `invokeDirect()`). Configuration is merely default values of parameters and parameters run-time overwrites of configuration.

Service Controller

A service controller is an implementation of `IServiceController` which extends `IService`. So from a client's point of view, a service controller has the same interface as a service. Service controllers provide support for control flows (or orchestration) of services (and other service controllers). Cornerstone out-of-box has `SequenceController` (that calls its member services sequentially) and `SwitchController` (which is like a switch statement). Other implementations can be added. Control flows can be changed easily by changing the configuration of service controllers without affecting any of the services involved.

Comparison to Other Work

Pipelines and Valves

A pipeline is similar to a `SequenceController` and valves its member services. But in Cornerstone, it is just one of the possible service controllers. You can implement your own and mingle it with other services and service controllers with ease.

Interceptors

Interceptor is a powerful concept. However, when it is used to implement control flows that are more than simplistic, it can become hard for a developer to understand the actual flow. Calling control flows out into their separate construct makes it easy to control and understand them.

Service Metric

Every service instance has a metric object associated with it which is the basis for creating the corresponding *Managed Bean* (MBean) in JMX. It can be logged in the database. Even parameters of invocations can be logged. It is very easy to trace run-time parameters that fly through a control flow to pinpoint problems. Developers don't need to understand a thing of JMX to use it.

Interceptor

Cornerstone supports a very simple but effective way of interceptors. `BaseService.invoke()` is implemented as a sequence of `invokeStart()`, `invokeMiddle()` and `invokeEnd()`. Control flows or changes of control flows are implemented using service controllers because control flows implemented using interceptors can quickly get out of hand.

Action, Action Controller and Action Metric

Cornerstone has its own MVC framework. Functionality of Action, Action Controller and Action Metric is exactly like that of Service, Service Controller and Service Metric, respectively. In fact, the Action source code is automatically generated from Service source code by Ant. The difference is only in their names and usage: Actions are used for presentation logic and Services for business logic.

Comparison to Other Work

Action related classes are mirror images of those of Service. When a developer has learned one, s/he has learned the other. An Action Controller *orchestrates* Actions (and other Action Controllers) which don't know which control flows they are participating in.

Sample Code

Refer to the document *Jetspeed Cornerstone Sample Code.*

Change History

Revision	Date	Changes
0.1	11/26/2003	Created.