IoC Cookbook - Patterns

Tapestry IoC has support for implementing several of the Gang Of Four Design Patterns. In fact, the IoC container itself is a pumped up version of the Factory pattern.

The basis for these patterns is often the use of *service builder methods*, where a configuration for the service is combined with a factory to produce the service implementation on the fly.

Related Articles

- PipelineBuilder Service
- StrategyBuilder Service
- IoC Cookbook - Patterns
- ChainBuilder Service

Chain of Command Pattern

Main Article: Chain of Command

Let's look at another example, again from the Tapestry code base. The InjectProvider interface is used to process the @Inject annotation on the fields of a Tapestry page or component. Many different instances are combined together to form a *chain of command*.

The interface has only a single method (this is far from uncommon):

```
public interface InjectionProvider
{
  boolean provideInjection(String fieldName, Class fieldType, ObjectLocator locator,
      ClassTransformation transformation, MutableComponentModel componentModel);
}
```

The return type indicates whether the provider was able to do something. For example, the AssetInjectionProvider checks to see if there's an @Path annotation on the field, and if so, converts the path to an asset, works with the ClassTransformation object to implement injection, and returns true to indicate success. Returning true terminates the chain early, and that true value is ultimately returned to the caller.

In other cases, it returns false and the chain of command continues down to the next provider. If no provider is capable of handling the injection, then the value false is ultimately returned.

The InjectionProvider service is built up via contributions. These are the contributions from the TapestryModule:

```
public static void contributeInjectionProvider(
    OrderedConfiguration<InjectionProvider> configuration,
    MasterObjectProvider masterObjectProvider,
    ObjectLocator locator,
    SymbolSource symbolSource,
    AssetSource assetSource)
{
  configuration.add("Default", new DefaultInjectionProvider(masterObjectProvider, locator));

  configuration.add("ComponentResources", new ComponentResourcesInjectionProvider());

  configuration.add("CommonResources", new CommonResourcesInjectionProvider(), "after:Default");

  configuration.add("Asset", new AssetInjectionProvider(symbolSource, assetSource), before:Default");

  configuration.add("Block", new BlockInjectionProvider(), "before:Default");
  configuration.add("Service", new ServiceInjectionProvider(locator), "after:*");
}
```

And, of course, other contributions could be made in other modules ... if you wanted to add in your own form of injection.

The configuration is converted into a service via a service builder method:

```
  public InjectionProvider build(List<InjectionProvider> configuration, ChainBuilder chainBuilder)
  {
    return chainBuilder.build(InjectionProvider.class, configuration);
  }
```

Now, let's see how this is used. The InjectWorker class looks for fields with the InjectAnnotation, and uses the chain of command to inject the appropriate value. However, to InjectWorker, there is no chain ... just a *single* object that implements the InjectionProvider interface.

```
public class InjectWorker implements ComponentClassTransformWorker
{
  private final ObjectLocator locator;

  // Really, a chain of command

  private final InjectionProvider injectionProvider;

  public InjectWorker(ObjectLocator locator, InjectionProvider injectionProvider)
  {
    this.locator = locator;
    this.injectionProvider = injectionProvider;
  }

  public final void transform(ClassTransformation transformation, MutableComponentModel model)
  {
    for (String fieldName : transformation.findFieldsWithAnnotation(Inject.class))
    {
      Inject annotation = transformation.getFieldAnnotation(fieldName, Inject.class);

      try
      {
        String fieldType = transformation.getFieldType(fieldName);

        Class type = transformation.toClass(fieldType);

        boolean success = injectionProvider.provideInjection(
            fieldName,
            type,
            locator,
            transformation,
            model);

        if (success) transformation.claimField(fieldName, annotation);
      }
      catch (RuntimeException ex)
      {
        throw new RuntimeException(ServicesMessages.fieldInjectionError(transformation
            .getClassName(), fieldName, ex), ex);
      }

    }
  }
}
```

Reducing the chain to a single object vastly simplifies the code: we've *factored out* the loop implicit in the chain of command. That eliminates a lot of code, and that's less code to test, and fewer paths through InjectWorker, which lowers its complexity further. We don't have to test the cases where the list of injection providers is empty, or consists of only a single object, or where it's the third object in that returns true: it looks like a single object, it acts like a single object ... but its implementation uses many objects.