Inversion of Control

Authors: Paul Hammant

Overview

Inversion of Control (IoC) is a design pattern that addresses a component's dependency resolution, configuration and lifecycle. Note to confuse things slightly, IoC is also relevant to simple classes, not just components, but we will refer to components throughout this text. The most significant aspect to IoC is dependency resolution and most of the discussion surrounding IoC dwells solely on that.

Types of IoC

There are many types of IoC, but we'll concentrate on the type of IoC that PicoContainer introduced to the community. Formerly known as type-3, now known as Constructor Injection. We'll ignore Setter Injection and Contextualized Lookup as they are described in the IoC Types page.

IoC Synonyms

DIP

One well-known synonym for IoC is DIP - described in Robert C. Martin's excellent Dependency Inversion Principle paper.

Hollywood Principle

A second nickname for IoC is The Hollywood Principle (Don't call us we'll call you).

IoC History

Some detail about the history of Inversion of Control - IoC History

Component Dependencies

It generally favors loose coupling between components. Loose coupling in turn favours:

- More reusable classes
- Classes that are easier to test
- Systems that are easier to assemble and configure

Explanation

Simply put, a component designed according to IoC does not go off and get other components that it needs in order to do its job. It instead *declar es* these dependencies, and the container supplies them. Thus the name IoC/DIP/Hollywood Principle. The control of the dependencies for a given component is inverted. It is no longer the component itself that establishes its own dependencies, but something on the outside. That something could be a container like PicoContainer, but could easily be normal code instantiating the component in an embedded sense.

Examples

Here is the simplest possible IoC component :

```
public interface Orange {
  // methods
}
public class AppleImpl implements Apple {
  private Orange orange;
  public AppleImpl(Orange orange) {
    this.orange = orange;
  }
  // other methods
}
```

Here are some common smells that should lead you to refactor to IoC :

```
public class AppleImpl implements Apple{
  private Orange orange;
  public Apple() {
    this.orange = new OrangeImpl();
  }
  // other methods
}
```

The problem is that you are tied to the OrangleImpl implementation for provision of Orange services. Simply put, the above apple cannot be a (configurable) component. It's an application. All hard coded. Not reusable. It is going to be very difficult to have multiple instances in the same classloader with different assembly.

Here are some other smells along the same line :

```
public class AppleImpl implements Apple {
  private static Orange orange = OrangeFactory.getOrange();
  public Apple() {
  }
  // other methods
}
```

Component Configuration

Sometimes we see configuration like so ...

```
public class BigFatComponent {
  String config01;
  String config02;
  public BigFatComponent() {
    ResourceFactory resources = new ResourceFactory(new File("mycomp.properties"));

    config01 = resources.get("config01");
    config02 = resources.get("config02");
  }
  // other methods
}
```

In the IoC world, it might be better to see the following for simple component designs :

```
public class BigFatComponent {
  String config01;
  String config02;
  public BigFatComponent(String config01, String config02) {
    this.config01 = config01;
    this.config02 = config02;
  }
  // other methods
}
```

Or this for more complex ones, or ones designed to be more open to reimplementation ..

```
public interface BigFatComponentConfig {
  String getConfig01();
  String getConfig02();
}
public class BigFatComponent {
  String config01;
  String config02;
  public BigFatComponent(BigFatComponentConfig config) {
    this.config01 = config.getConfig01();
    this.config02 = config.getConfig02();
  }
  // other methods
}
```

With the latter design there could be many different implementations of BigFatComponentConfig. Implementations such as:

1. Hard coded (a default impl)
2. Implementations that take config from an XML document (file, URL based or inlined in using class)
3. Properties File.

It is the deployer's, embeddor's or container maker's choice on which to use.

Component Lifecycle

Simply put, the lifecycle of a component is what happens to it in a controlled sense after it has been instantiated. Say a component has to start threads, do some timed activity or listen on a socket. The component, if not IoC, might do its start in its contructor. Better would be to honor some start/stop functionality from an interface, and have the container or embeddor manage the starting and stopping when they feel it is appropriate:

```
public class SomeDaemonComponent implements Startable {
  public void start() {
    // listen or whatever
  }
  public void stop() {
  }
  // other methods
}
```

Notes

The lifecycle interfaces for PicoContainer are the only characterising API elements for a component. If Startable was in the JDK, there would be no need for this. Sadly, it also menas that every framework team has to write their own Startable interface.

The vast majority of components do not require lifecycle functionality, and thus don't have to implement anything.

IoC Exceptions

Of course, in all of these discussions, it is important to point out that logging is a common exception to the IoC rule. Apache has two static logging frameworks that are in common use: Commons-Logging and Log4J. Neither of these is designed along IoC lines. Their typical use is static accessed whenever it is felt appropriate in an application. Whilst static logging is common, the PicoContainer team do not recommend that developers of reusable components include a logging choice. We suggest instead that a Monitor component interface is created and default adapters are provided to a number of the logging frameworks are provided.

Subpages