IoC Types

Overview

In recent years different approaches have emerged to deliver an IoC vision. Latter types, as part of a 'LightWeight' agenda have concentrated on simplicity and transparency.

IoC Types - Family Tree

Devised at ThoughtWorks' London offices in December of 2003. Present at the "Dependency Injection" meeting were Paul Hammant, Aslak Hellesoy, Jon Tirsen, Rod Johnson (Lead Developer of the Spring Framework), Mike Royle, Stacy Curl, Marcos Tarruela and Martin Fowler (by email).

Inversion of Control

- Dependency Injection (former type 3)
  - Constructor Dependency Injection
    Examples: PicoContainer, Spring Framework, (EJB 3.1 ?)
  - Setter Dependency Injection (former type 2)
    Examples: Spring Framework, PicoContainer, EJB 3.0
  - Interface Driven Setter Dependency Injection
    Examples: XWork, WebWork 2
  - Field Dependency Injection
    Examples: Plexus (to be confirmed)
- Dependency Lookup
  - Pull approach (registry concept)
    Examples: EJB 2.x that leverages JNDI, Servlets that leverage JNDI
  - Contextualized Dependency Lookup (former type 1)
    AKA Push approach
    Examples: Servlets that leverage ServletContext, Apache's Avalon, OSGi (to be confirmed), Keel (uses Avalon)

See also Constructor Injection, Setter Injection, Contextualized Lookup for more information.

Note Field Injection was provisionally known as 'type 4'. There was really no interest 'type 4' until EJB3.0. Getter Injection flourished for a while, but did not take and was never supported by the PicoContainer team.

Examples of Common Types

Constructor Dependency Injection

This is where a dependency is handed into a component via its constructor :

```
public interface Orange {
  // methods
}

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

Setter Dependency Injection

This is where dependencies are injected into a component via setters :

```
public interface Orange {
  // methods
}

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

Contextualized Dependency Lookup (Push Approach)

This is where dependencies are looked up from a container that is managing the component :

```
public interface Orange {
  // methods
}

public class AppleImpl implements Apple,
DependencyProvision {
  private Orange orange;
  public void
doDependencyLookup(DependencyProvider dp)
throws DependencyLookupExcpetion{
    this.orange = (Orange)
dp.lookup("Orange");
  }
  // other methods
}
```

Terms: Service, Component & Class

Component is the correct name for things managed in an IoC sense. However very small ordinary classes are manageable using IoC tricks, though this is for the very brave or extremists 😃

A component many have dependencies on others. Thus dependency is the term we prefer to describe the needs of a component.

Service as a term is very popular presently. We think 'Service' dictates marshaling and remoteness. Think of Web Service, Database service, Mail service. All of these have a concept of adaptation and transport. Typically a language neutral form for a request is passed over the wire. In the case of the Web Service method requests are marshaled to SOAP XML and forward to a suitable HTTP server for processing. Most of the time an application coder is hidden from the client/server and marshaling ugliness by a toolkit or API.

Obsoleted Terms

Types 1, 2 and 3 IoC were unilaterally coined earlier in 2003 by the PicoContainer team and published widely.

Type 1 becomes Contextualized Dependency Lookup
Type 2 becomes Setter Dependency Injection
Type 3 becomes Constructor Dependency Injection

Dependency Injection versus Contextualized Lookup

Dependency Injection is non-invasive. Typically this means that components can be used without a container or a framework. If you ignore life cycle, there is no import requirements from an applicable framework.

Contextualized Dependency Lookup is invasive. Typically this means components must be used inside a container or with a framework, and requires the component coder to import classes from the applicable framework jar.

Note that Apache's Avalon (and all former type-1 designs) are not Dependency Injection at all, they are Contextualized Dependency Lookup.

What's wrong with JNDI ?

With plain JNDI, lookup can be done in a classes' static initialiser, in the constuctor or any method including the finaliser. Thus there is no control (refer C of IoC). With JNDI used under EJB control, and concerning only components looked up from that bean's sisters (implicitly under the same container's control), the specification indicates that the JNDI lookup should only happen at a certain moment in the startup of an EJB application, and only from a set of beans declared in ejb-jar.xml. Hence, for EJB containers, the control element should be back. Should, of course, means that many bean containers have no clue as to when lookups are actually being done, and apps work by accident of deployment. Allowing it for static is truly evil. It means that a container could merely be looking at classes with reflection in some early setup state, and the bean could be going off and availing of remote and local services and components. Thus depending whether JNDI is being used in an Enterprise Java Bean or in a POJO, it is either an example of IoC or not.