xTier™ Inversion of Control

xTier™ uses IoC in two ways: 1) All of the services provided by xTier™ utilize IoC in the way that they are build and configured, and 2) xTier™ provides a robust IoC Service that the developer can utilize to build their applications using the principles of IoC.

Key Benefits:

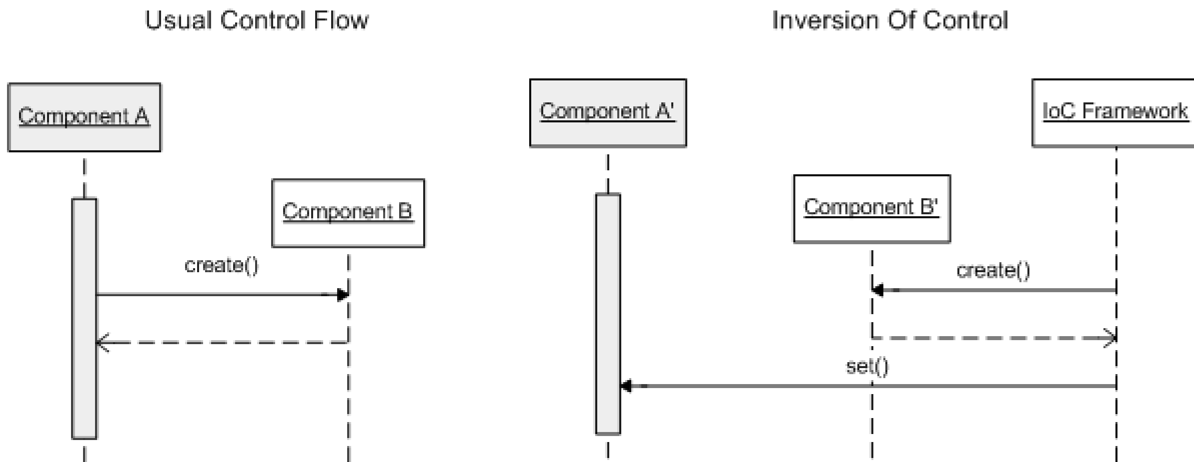
- Creates a loosely coupled set of services that can be "mixed and matched" to meet project requirements.
- Provides a robust environment to achieve loosely coupled architecture for applications developed with xTier™

The Principles of Inversion of Control

Inversion of Control (IoC) is a design pattern that helps to reduce coupling and dependencies between components. In general, IoC is a principle that separates an API from a framework, and is based on who is "in control". The word "inversion" from the pattern name refers to inverting the "normal" way in which components obtain references to each other. This principle can be illustrated by example through the difference between an IoC framework and a class library. The major difference between an IoC framework and a class library is that the framework calls the application code, whereas normally the application code would call the class library.

This principle of inverting control is sometimes referred to as the Hollywood Principle, ("Don't call us, we call you!") or alternatively is also known as Dependency Injection. The ideas behind IoC aren't especially new; in fact, some have remarked that IoC is a new acronym for the older Dependency Inversion Principle. In any case, the main principle behind IoC is that a given component no longer needs to know how to obtain the dependency on another component and as a result, the code that uses these components is cleaner, more focused and more flexible. Further, the applications that are built using these types of components are loosely coupled and are generally more scalable, more maintainable, and more testable ultimately improving overall system architecture and quality. IoC can be simply described as follows: There are two components A and B, and A depends on (uses) B. In a conventional scenario A will need to explicitly obtain B and establish its dependency on it. Using an IoC design pattern, however, a system could automatically obtain and establish a dependency on B without any explicit request on the part of component A - hence reversing the control of dependency.

The main principles of IoC are illustrated by following diagrams.



The following code is a very simple example that illustrates the main goals and advantages of IoC design principles.

```
class SomeComponent {
	private Logger logger;

	public doSomething () {
		. . .

		logger.log("Message");
	}
}

interface Logger {
	public void log(String message);
}
```

Assume that some component needs logging, but that the logger implementation can be completely different depending on where this component will ultimately be used. For instance, it could be console logging, file logging or xml formatted logging, for instance, depending on the project requirements and the deployment environment. Generally speaking, the component should somehow obtain a logger instance and will result in the component being dependent on a specific and concrete logger implementation.

Instead, the basic idea when using IoC for this scenario, is that the IoC framework populates a field in the component with an appropriate implementation for the logger interface. As a result there is no dependency on a specific Logger implementation.

The Basic types of IoC

There are three types of IoC implementations that are currently available and are described, along with brief examples, in the following sections. This discussion does not provide an exact implementation of the way in which an IoC framework should be configured to utilize each type because this would be dependent on the particular implementation. In later sections, the way in which xTier™ utilizes IoC is discussed in detail.

Type 1: Interface Injection

This type of IoC is based on providing a specific interface for each specific instance of a dependency injection (i.e. for each component that needs that behavior). At runtime these interfaces will be analyzed and the proper dependencies will be set for each component. The major problem with this approach is that it requires too many "marker" interfaces to be created which creates a heavy maintenance burden and polluted design. Interface injection IoC is used by the open source Avalon framework. One of the distinct capabilities of such an approach is that its implementation does not require external non-Java configuration.

Based on the example above, interface injection can be illustrated as follows. First define the interface for injecting a logger into an object:

```
public interface InjectLogger {
   void injectLister(Logger logger);
}
```

Next, the component's class should implement ther InjectLogger interface:

```
class SomeComponent implements InjectLogger...
   public void injectLogger(Logger logger) {
      this.logger = logger;
   }
```

This also assumes that there is some sort of custom Logger implementation:

```
class CustomLogger  implements Logger {
		public void log (String message) {
				System.out.println("Custom logger [" + message + "]");
		}
}
```

The final step is to configure the IoC framework to use the appropriate logger implementation. For the purposes of this discussion the concrete manner of doing this will not be discussed, because this will vary for different IoC frameworks. The most common practice is that the IoC framework uses some configuration file (often XML), where the exact implementation to be used can be specified.

Type 2: Setter Injection

Using this type of IoC the dependency injection is achieved by setting certain properties using a "JavaBean type" of convention. Usually, the specific properties are defined in an XML file and at startup or on a per request basis they are set or "wired-up" to form a dependency. This is a most common type of IoC and is used by a variety of software systems including some open source frameworks such as Spring and PicoContainer.

To accept injection our **SomeComponent** class should define a setter method which receives a Logger instance as parameter:

```
class SomeComponent...
   private Logger logger;
            public void setLogger(Logger logger) {
                        this.logger = logger;
            }
```

This injected property is then specified (i.e "logger", as in the JavaBean convention, the **SomeComponent** class designed the **setLogger(...)** method) for the class **SomeComponent** and the specific logger implementation class, which is to be instantiated and should be used for this property. The IoC framework performs the instantiation of this class and passes the instance to the **setLogger** method.

Type 3: Constructor Injection

Unlike Type 1 and 2, Type 3 IoC makes use of constructors to pass in the component's dependencies instead of using setters or per-component injection interfaces. Although, seemingly an elegant solution that should in theory lead to a better design by enforcing that each component is fully initialized during creation, the real applicability of Type 3-only IoC is very limited. In fact, it goes against the very nature of IoC as it basically requires each component to be designed to fit such an arrangement which defeats the purpose of decoupling dependency control from the component. Thus most IoC implementations use some combination of Type 3 and Type 2 or Type 1. The PicoContainer provides an attempt to develop an almost Type 3-only IoC system.

Continuing the example code introduced above, for such an approach the **SomeComponent** class needs to declare a constructor that includes the objects it needs injected:

```
Class SomeComponent...
   public SomeComponent(Logger logger) {
      this.logger = logger;
   }
```

Then the IoC framework should be configured to use the proper logger implementation during **SomeComponent** creation.

The xTier™ IoC Service

Not only does xTier™ utilize IoC internally, it provides an IoC service that fully exposes the IoC functionality and features to the developer allowing the creation of highly configurable and flexible software systems. The xTier™ IoC Service combines traditional Type 2 and Type 3 IoC with some unique features:

- Direct support for .NET and Java in the IoC descriptor that allows using the same XML definition for an IoC object in a cross-paradigm fashion.
- Support for 'new', 'singleton' and a user-defined 'keyed' creation policies. (Creation policies will be described below in detail.)
- Generalized support for setters and init methods. The xTier™ IoC Service allows calling any arbitrary methods on the object as long as its signature can be properly constructed. That feature is extremely important for supporting "legacy" components that may require a constructor call, setting one or many parameters via JavaBean type of setters or via other non-compliant methods, and then calling one or more initialization methods that may also take an arbitrary number of input parameters. Furthermore, xTier™ supports making certain calls that must be performed in a certain order, repeated or intermixed with each other.
  - For example, the IoC Service can create an IoC object as a ArrayList with a certain size passed into the constructor, populate it with certain pre-defined values via calls to the ArrayList.add(java.lang.Object) or ArrayList.add(int, java.lang.Object) methods, call ArrayList.trimToSize() and pass this object to another IoC object as a parameter.
- Support for boxed and unboxed types can be passed to constructors and setters/init methods. The following table shows all supported types and their Java counterparts in boxed and unboxed form:

IoC Type	Unboxed	Boxed
int8	byte	java.lang.Byte
int16	short	java.lang.Short
int32	int	java.lang.Integer
int64	long	java.lang.Long
float32	float	java.lang.Float
float64	double	java.lang.Double
Char	char	java.lang.Character
boolean	boolean	java.lang.Boolean
String	-	java.lang.String
Date	-	java.util.Date

- Uniform API for Java and .NET

Lets examine how the xTier™ IoC Service works. The IoC service acts as a factory for IoC objects. All IoC objects should be defined using XML in the **xtier_ioc.xml** configuration file, which follows the standard xTier™ service configuration

pattern. The interface **com.fitechlabs.xtier.services.IoC.IocService** is the key IoC service element. It provides 2 methods for runtime IoC object creation:

```
Object makeIocObject(String) throws IocServiceException;
Object makeIocObject(String, Object) throws IocServiceException;
```

The first form (**makeIocObject(String)**) has one string parameter – UID – the IoC object identifier, which should be unique within the current configuration. This form is used for such objects that are defined with the creation policy of "new" or "singleton". If the creation policy is "new", a new object will be created each time this method is called with the given UID. If the creation policy is "singleton", an object will be created only the first time and stored so that following calls to this method with the same UID will return an already created object instance.

The second form (**makeIocObject(String, Object)**) uses an additional parameter – key. This form is applicable only for objects that are defined with the creation policy "keyed". If this method has already been successfully called with this UID and key, it will return the same instance. If it is a first call for this UID and key, then a new object will be created and returned.

Next, let's examine how IoC objects are defined in the XML configuration file. Here is a simple example:

```
<ioc policy="new" uid="str">
  <java class="java.lang.String">
    <ctor>
      <arg type="string">xTier</arg>
    </ctor>
  </java>
</ioc>
```

This XML fragment defines an IoC object of type **java.lang.String** with UID "str" and creation policy "new" (note that the when working in .NET the **<clr>** tag is used instead of the **<java>**). The **<ctor>** XML tag specifies the constructor and its parameters which are used upon IoC object creation. The **<arg>** tags define the arguments to be passed. This example requires that the one-argument (string) constructor be used. On object creation this constructor will be called and the string argument with value "xTier" will be passed to it. At the result we will have a **java.lang.String** instance representing the string,"xTier".

The following is a short example demonstrating how this would be used.

```
// Get the instance of xTier kernel.
XtierKernel xtier = XtierKernel.getInstance();

// Get the instance of 'IoC' service.
IocService ioc = xtier.ioc();

// Get Ioc object, defined in configuration – in out example "xTier" string, passing its
uid.
String str = (String)ioc.makeIocObject("str");
```

A more complicated example will provide further insight into the facilites provided by the xTier™ IoC Service. Consider the following IoC XML configuration fragment.

```
<ioc policy="keyed" uid="list">
  <java class="java.util.ArrayList">
    <ctor>
      <arg type="int32">4</arg>
    </ctor>

    <call method="add"><arg type="int32" boxed="true">1</arg></call>
    <call method="add"><arg type="int32" boxed="true">2</arg></call>
    <call method="add"><arg type="int32" boxed="true">3</arg></call>
    <call method="add"><arg ref-uid="str"/></call>
  </java>
</ioc>
```

This fragment defines a **java.util.Array** IoC object with the "keyed" creation policy. The constructor requires one argument of type "int32", which corresponds to the "long" java data type, with value '4'. Then we can see four **<call>** tags each defining a call to the 'add' method. On calls 1, 2, and 3 the **<arg>** tags define an argument of type "int32". The attribute "boxed" with value "true" means that a boxed type for the parameter will be passed (i.e. for Java it is **java.lang.Interger**). If "boxed" is set to "false" or omitted for types like "int8", "int16", "int32", "int64", "float32", "float64" and "boolean", this means the constructor or method expects primitive types, such as "int", "float" or "boolean" for Java. The "string" and "date" argument types correspond to the "java.lang.String" and "java.util.Date", as unboxed types are unspecified for these types of arguements.

The forth **<call>** tag in this series demonstrates an additional feature of the xTier™ "IoC" Service. As we can see, the **<arg>** tag has a "ref-uid" attribute with a value of "str". This attribute specifies that all parameters for this argument (type and value) should be taken from the other IoC descriptor whose UID is specified in this attribute. This attribute is the essence of establishing the dependency by one IoC component (descriptor) to another. In this example, this attribute points to the IoC object with the uid "str", defined in the previous fragment. This means that on this call the IoC object with the uid "str" will be created and passed as an argument to the add (Object) method. (Which was previously defined as an IoC object that was a **java.lang.String** instance with the value "xTier".

Simply stated, with this definition, a **java.util.Array** instance with size "4" will be passed into the constructor, populated with 3 **java.lang.Integer** instances (with values 1, 2 and 3) and one **java.lang.String** instance with the value of "xTier", which will also be created using its IoC definition. The following IoC service call will provide this IoC object.

```
// Get the instance of xTier kernel.
XtierKernel xtier = XtierKernel.getInstance();

// Get the instance of 'ioc' service.
IocService ioc = xtier.ioc();

// Gets java.util.Array instance, defined in example XML configuration:
List list = (List)ioc.makeIocObject("list", Thread.currentThread());
```

Remember, that in this example the IoC object was defined with the "keyed" creation policy, and as such the **makeIocObject(String, Object)** method form must be used. This form returns the same instance for the same id and key. In this case, assume that the current thread is used for per-thread object creation.

For simplicity sake, in these examples the IoC objects were standard **String** and **ArrayList** object, however, any non-abstract class can be instantiated in a similar way using the IoC service.

Using IoC for xTier™ Service Configuration

The configuration for each service provided by xTier™ is based on IoC. Every XML configuration file uses the same syntax and semantic to define IoC components that are used in the configuration of a given service. Any other configurable xTier™ services will use exactly the same semantic and configuration syntax for defining their IoC components as the IoC service itself uses for user defined objects.

An examination of the 'jobs' service will serve to illustrate the way in which xTier™ services are configured using IoC. The 'Jobs' service allows executing a unit of work according to certain schedule. It provides a uniform and functionally rich facility for work time-based scheduling. One of the key job service elements is job. A job defines a task that should be executed, with such parameters as name, execution times, counts of execution, etc. The thing most interesting from developer's perspective is the job body. The job body determines exactly what will be executed on job invocation. The job body is specified by the **com.fitechlabs.xtier.services.jobs.JobBody** interface. This interface contains only one method **invoke(JobContext)**. Developers are free to determine any tasks they want in their implementation, and this task will be executed on job invocation.

Another key job service element is the job scheduler. The job scheduler defines the time when a job is to be executed. Every job scheduler should implement the **com.fitechlabs.xtier.services.jobs.JobScheduler** interface. This interface consists of 2 methods – one should return the next execution time for a job, and the second method resets the scheduler. The developer can provide a custom scheduler implementation or alternatively, use one of the pre-defined schedulers. (The predefined schedulers include implementations that can invoke jobs on a daily, weekly, monthly basis, through fixed time rate or fixed delay, or one-time only basis). Using this service provides a typical task where an IoC framework can demonstrate its main advantages – the job service should allow use of an unlimited number of different job body and job scheduler implementations without creating an explicit dependency on them. With a traditional non-IoC approach it would not be possible to use custom-defined schedulers without modifying the service's implementation code.

The Jobs IoC-based configuration works in the following manner (remember that IoC object definitions are identical to the ones used by the IoC service itself, described in the previous paragraphs). Suppose a task is to be executed on a daily basis. In this case the pre-built **JobScheduler** implementation for scheduling daily tasks, **com.fitechlabs.xtier.services.jobs.schedulers.JobDailyScheduler**, can be utilized. Let's examine how it is defined in the jobs XML configuration for runtime usage:

```
<!--
    Daily job declaration.
-->
<job name="daily.job" start="false">
    <scheduler>
        <ioc policy="new">
            <java class="com.fitechlabs.xtier.services.jobs.schedulers.JobDailyScheduler">
```

```
                <ctor>
                    <!-- Day times. -->
                    <arg type="string">03:43:00 am, 12:42:12 am, 7:05 pm, 9:40 pm</arg>
                </ctor>
            </java>
        </ioc>
    </scheduler>

    <body>
        <ioc policy="new">
            <java class="com.fitechlabs.xtier.examples.services.jobs.SimpleJobBody"/>
        </ioc>
    </body>
</job>
```

Here the **<job>** tag that provides daily job definition. The "Name" attribute defines the job ID for run-time job access, which should be unique within the current configuration. The "Start" attribute specifies if this job is scheduled automatically on service start or not. When a task it scheduled, it means that it is executed according to its scheduler. The "False" value specifies that this task is not to be scheduled automatically, but the service provides the possibility to schedule it at runtime with any start time. The **<scheduler>** tag defines which **JobScheduler** implementation this task uses. It points to a class via the nested **<ioc>** tag, which was described in the previous section. In this example, it points to **com.fitechlabs.xtier.services.jobs.schedulers.JobDailyScheduler**. As it was mentioned above, it is possible to use any pre-built or user defined scheduler here (i.e. any class implementing the **JobScheduler** interface). The **<ctor>** tag defines which constructor is used for IoC object creation and contains the values of the arguments to be passed. For example, "03:43:00 am, 12:42:12 am, 7:05 pm, 9:40 pm" is passed, which is parsed by the daily scheduler, with the result that this task will be executed 4 times a day: at 03:43:00 AM, at 12:42:12 AM, at 7:05 PM and at 9:40 PM.

The **<body>** tag points to the job body implementation. In this example **com.fitechlabs.xtier.examples.services.jobs.SimpleJobBody** is used. This is a pre-build job body implementation that does nothing but prints diagnostic messages when invoked and is used for demonstration purposes only. The user, however, would define the appropriate task to be carried out by implementing the **JobBody** interface.

The following code example demonstrates what should be done at runtime to make the jobs service execute this task:

```
// Get the instance of xTier kernel.
XtierKernel xtier = XtierKernel.getInstance();

// Get the instance of 'jobs' service.
JobsService jobs = xtier.jobs();

// Get daily job, defined in XML configuration, by its name.
Job dailyJob = jobs.getJob("daily.job");

// Schedule daily job. Note that if job is defined to be scheduled automatically
// (attribute "start" = "true" in xml configuration), there is no need to schedule it as
// below – such call is performed by service on startup.
 jobs.schedule(dailyJob.getName(), true);
```

Since the daily job is scheduled, it starts executing 4 times each day, as defined by the XML configuration by the time-string argument (in actual fact the execution days are determined by the calendar instance, but for the purposes of this example, this is left to a discussion of the jobs service itself, to concentrate on the IoC configuration).

This example considered configuring the pre-defined implementation for the job body and job scheduler, but developers are free to define their own implementations – the point being that the custom implementation and the pre-build implementations are configured in the exact same way. There are just two things needed to do to add a user defined job with user scheduling:

- implement JobBody and JobScheduler interfaces
- add <job> element describing configuration for a new job.

Additionally, the developer can use a custom defined job, (i.e. **JobBody** implementation) with any of the pre-built schedulers (and in this case a custom scheduler is not needed).

Conclusion

The xTier™ IoC service provides two significant functions. First, it allows other services to be easily configured. The developer can write custom classes (jobs or jobs schedulers in the preceding example), or choose among pre-built implementations, and customize the xTier™ services to use these classes simply by making changes to the relevant services' XML configuration). Second, developers can work with IoC service directly, which provides a convenient possibility to use all the advantages of an IoC approach for their xTier™-based applications.

Fitech Laboratories Inc.

Corporate Headquarters
300 Montgomery St., Suite 621
San Francisco, CA 94104
USA

Phone: 1-415-371-8234
Fax: 1-415-371-8237

East Coast Sales Office
330 Madison Ave., 9th floor
New York, NY 10017
USA

Phone: 1-646-495-5076

Fitech Laboratories Japan
Toranomon40 MT Bldg. 3F
5-13-1 Toranomon Minato-ku
Tokyo 105-0001, Japan

Phone: +81-3-5402-7711
Web: www.fitechlabs.co.jp

Copyright © 2003, Fitech Labs, Inc All rights reserved. All trademarks and registered trademarks of other products and services mentioned in this report are the property of their respective owners.