RC24092 (W0610-159) October 30, 2006
Computer Science

IBM Research Report

Best Practice Guide for Portable Java Components: The Eleven Best Practices Based on the Inversion-of-Control Principle

Woody Yun-Wu Huang, John Ponzo
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Matt R. Hogstrom, Dain Sundstrom
IBM Software Group

Tom Musta
IBM Systems and Technology Group

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Background

The J2EE community has seen a growing trend toward lightweight containers that provide an integration framework to wire Java services. Examples of these frameworks include Geronimo's GBean, the Spring Framework (or Spring), Jakarta's HiveMind and Pico. Common to all these frameworks is that they are all more or less based on the *Inversion-of-Control* (IoC) principle.

The IoC principle is based on the *separation of concerns* concept which in computer science specifies that a complex software system should be partitioned such that each component has a clearly defined focus or interest while the overlap of functions between components is minimized [1]. IoC in particular focuses on the separation of concerns between a framework and its hosted services. A service should focus entirely on its business logic and not on how it interfaces with the framework and other services. The framework's concern is then to make sure all its services are properly wired and their lifecycle managed.

Based on this principle, one of the most prominent design patterns employed by IoC is *Dependency Injection* [2]. Applying this pattern, the container resolves and creates all the dependency references for a Java service and all configuration data, and "injects" these references into the said service. When a service is started by the framework, all its references are already created and bound to local variables. Thus, services are mutually encapsulated as their implementation details are known only by the framework, not between the services themselves.

Along with this IoC container trend comes a school of thought that emphasizes non-invasive frameworks. A non-invasive framework means that its components need not implement any framework-related code in order to be managed by the framework. In other words, non-invasive frameworks can manage components that have no knowledge of the underlying framework. This non-invasive concept advocates encapsulation between components and frameworks that allow components to be highly portable across frameworks.

The GlueCode team, having worked on various versions of Geronimo's GBean framework, is the first team in IBM to advocate building portable components based on the IoC principle with a non-invasive framework. More specifically, the GlueCode team identifies eleven best-practice principles in Java programming that serve as the guidelines to develop portable components. This document describes in detail the eleven principles while discussing what they are and how they can be adopted.

Terminology

- **Framework** - the platform that hosts the components. It may include an IoC container that injects dependencies into services. OSGi and GBean are examples of frameworks.
- **IoC Container (or IoC)** – the container that wires services through dependency injection (Best Practices 1,2 and 3). IoC also performs *configuration injection* (Best Practice 9) and *lifecycle injection* (Best Practice 5). Spring and XBean Reflect are examples of IoC Containers.
- **Module** – a collection of services, their supporting classes and some configuration data. OSGi bundles are examples of modules. A module typically has dependency on other modules that is specified in the configuration data. Module dependency is typically modeled at runtime through delegation in class loading; it is out of the scope of this document.
- **Component** – loosely defined as a module or a service, depending on context.
- **Configuration** – the meta information about a module and its services. It is usually captured in a document file such as OSGi's manifest file or Geronimo's configuration plans.
- **Service** – a public Java class implementing some interfaces which define the types seen from other services. A service may depend on other services. Depending on context, *service* may also refer to an object of its class. In other literature, *service* is also known as *bean*.

Best Practice 1: Use constructor injection.

Dependency injection comes in several types. Two most common types are *constructor injection* and *setter injection*. In the IoC model, all services must be properly configured in their respective configuration files[1]. IoC relies on the information in a configuration file to create services listed in the file. While the actual constructor injection procedure may differ from IoC to IoC, the following example depicts the Spring-like way in constructor injection [3].

Example 1a depicts the `ServiceAImpl` Class which implements the `ServiceA` interface. `ServiceAImpl` has a constructor with three parameters `serviceB`, `serviceC` and `name` of respective types `ServiceC`, `ServiceD` and `String`. `ServiceC` and `ServiceD` are interfaces. The three parameters are respectively bound to three local variables with the same names. Since the three variables are bound to services injected by IoC, they are declared as `private` and `final` to ensure that they cannot be changed by component code.

[1] Usually each module is associated with a configuration file that contains the metadata of the module and its services.

```
package packageA.pathA;
...
public class ServiceAImpl implements ServiceA {
    private final ServiceB serviceC;
    private final ServiceC serviceD;
    private final String name;

    public ServiceAImpl(ServiceC serviceC,
                        ServiceD serviceD,
                        String name) {
         this.serviceC = serviceC;
         this.serviceD = serviceD;
         this.name = name;
     }
...
}
```

Example 1a: The `ServiceAImpl` class

```
...
 <bean id="serviceA" name="serviceA"
                     class="packageA.pathA.ServiceAImpl">
     <constructor-arg index="0" ref="serviceC"/>
     <constructor-arg index="1" ref="serviceD"/>
     <constructor-arg index="2" value="aName"/>
 </bean>

 <bean id="serviceC" name="serviceC"
                     class="packageC.pathC.ServiceCImpl">
 ...
 </bean>

 <bean id="serviceD" name="serviceD"
                     class="packageD.pathD.ServiceDImpl">
 ...
 </bean>
 ...
```

Example 1b: Configuration of `serviceA`, `serviceC` and `serviceD`.

Example 1b depicts the configuration of service `serviceA` using the Spring Framework's configuration format. It also shows the partial configuration of services `serviceC` and `serviceD`. The configuration of service `serviceA` defines a constructor for `serviceA` that has three parameters. The first two parameters are references to services `serviceC` and `serviceD`, whereas the third parameter is a `String` whose value is `aName`. This constructor declaration specifies that the two references `serviceC` and `serviceD` are to be resolved and together with a `String` valued `aName` are to be

injected when `ServiceA` is instantiated by the IoC container. Note that the `index` attributes in the `<constructor-arg>` tags are optional in this case as the three parameters all have a different type; hence they unambiguously define a construction signature without being in a particular order. But the inclusion of the `index` attribute is recommended as it helps improve readability[2].

When IoC needs to create the `serviceA` service, it first checks the configuration data for `serviceA`. Based on the `<constructor-arg>` tags, IoC determines that `serviceA` has dependency on `serviceB` and `serviceC`. IoC then tries to locate the two services[3] that `serviceA` depends on and checks to see if they have already been created and started. If they have been started, IoC uses Java reflection to inspect all constructors signatures of `packageA.pathA.ServiceAImpl`. If a constructor of the same signature as depicted in Example 1a is found, IoC evokes the constructor to create a new instance of `ServiceAImpl` by passing in the two service objects `serviceC` and `serviceD`, as well as a `String` with a value of `aName`.

In the constructor, all three objects passed in through parameters are bound to local variables promptly. The `serviceAImpl` class code needs only to import the interfaces `ServiceC` and `ServiceD`, and is encapsulated from their implementation classes.

If `serviceC` or `serviceD` or both has not been created, IoC temporarily halts the process of creating the `serviceA` service and proceeds to first create the dependent service that has yet been created. This could begin a recursive process until all dependency services are created and started. Once a service is successfully created, the framework starts to manage its lifecycle by evoking its start method (Best Practice 5).

Best Practice 2: Use setter injection.

The basic idea of setter injection is that the service code includes setter methods to inject dependency references. IoC, instead of using construction parameters to injection references, uses the default constructor to create the service, and calls the service's setter methods to pass in the references.

A properly formed setter must follow the JavaBeans [4] specification. Specifically, a setter must return a void type, have the method name that starts with "set" and take a single argument. Example 2a depicts the `ServiceBImpl` Class which implements the `ServiceB` interface. `ServiceBImpl` has three setter methods that each sets the value of a property to the parameter of the method.

---

[2] Other frameworks such as XBean-reflect and Spring 2.0 support named constructor arguments. The names of the constructor arguments are obtained from the debugging information in the class file or from an external properties file created during the build process.

[3] Locating services can typically be done by executing the service lookup queries provided by the kernel.

```
package packageB.pathB;
...
public class ServiceBImpl implements ServiceB {
    private ServiceB propertyC;
    private ServiceC propertyD;
    private String name;

    public void setPropertyC(ServiceC propertyC) {
        this.propertyC = serviceC;
    };

    public void setPropertyD(ServiceD propertyD) {
        this.propertyD = serviceD;
    };

    public void setName(String name) {
        this.name = name;
    };
...
}
```

Example 2a: The `ServiceBImpl` class

```
...
<bean id="serviceB" name="serviceB"
                    class="packageB.pathB.ServiceBImpl">
    <property name="propertyC" ref="serviceC"/>
    <property name="propertyD" ref="serviceD"/>
    <property name="name" value="aName"/>
</bean>

<bean id="serviceC" name="serviceC"
                    class="packageC.pathC.ServiceCImpl">
...
</bean>

<bean id="serviceD" name="serviceD"
                    class="packageD.pathD.ServiceDImpl">
...
</bean>
...
```

Example 2b: Configuration of `serviceB`, `serviceC` and `serviceD`.

Example 2b depicts the configuration of service `serviceB`, and the partial declaration of services `serviceC` and `serviceD` in their configuration file. It is based on the Spring format [3]. The configuration of the service `serviceB` defines three properties named `propertyC`, `propertyD` and `name`. The first two parameters are references to services

`serviceC` and `serviceD` respectively, whereas the third parameter is a `String` whose initial value is `aName`. These property declarations specify that the two references `serviceC` and `serviceD` are to be resolved and together with a `String` valued `aName` are to be injected through setter methods right after `ServiceB` is instantiated by the IoC container.

Like constructor injection, the referenced services that a service depends on must be created and started before this service can be created. The same recursive process in creating dependency services as in constructor injection also applies to the setter injection. The order of execution of these setter methods is unspecified.

Constructor Injection vs Setter Injection

The arguments for constructor injection generally include:

- With constructor injection, all dependencies are bound within the constructor block, making the exception handling, testing and debugging easier than the setter injection. Furthermore, the constructor creates a well-formed instance which is in general a good practice in OO programming design. In contrast, instances based on setter injection only become well-formed when all their setter methods are executed, making the use of them more error prone in a non-IoC environment.
- Some existing components are written without IoC in mind but use the constructor to pass in references to create well-formed instances. Constructor injection can take advantage of this practice and port them to IoC without being invasive. To refactor them for setter injections requires adding setter methods to them. As for components that depend on other methods to obtain dependency references, such as those using locator or setter injection, refactoring them for constructor injection will be invasive.
- If a service needs to bind references to immutable local variables, it can only achieve this in the constructor. Constructor injection allows the component to separate data into mutable and immutable fields as designated by the *final* access modifier. Designating fields as immutable makes multithreaded programming easier due to the reduced number of mutable fields that must have synchronized access. A drawback of using is that a field that is immutable must be marked as mutable. To make matters worse the class contains a setter for the field and other components that do not properly handle the setter after the start method may erroneously reset the field. Commonly the field is simply set without synchronization or the field is set but side-effects such as rebinding a protocol to a new port are not carried out.

The arguments for setter injection usually include:

- The number of constructor parameters can become unpleasantly large if a service has many dependency references. This makes code hard to read and maintain. Compound this with a service class that has a non-trivial class hierarchy and needs to pass parameters to super classes, then the constructor code become very bloated, making the system error prone. Setter injection in contrast avoids this

pitfall as it can accommodate a large number of references by simply adding more setter methods. However, a service needing a large number of references may be the symptom of a bad design which can be improved by splitting this service into smaller, more manageable units. Adding more setter methods may only hide the issue.

- Setter injection can support circular service references; constructor injection cannot. However, circular references are a sign of a design flaw which should be investigated.
- Setter injection more readily supports dynamicity in service references, e.g., a referent service can theoretically be replaced without re-instantiating the dependent. This is however a very tricky procedure as it requires pausing the service as the data is updated. A pause in the service may be impossible if there is any outstanding work such as a transaction. Currently, there are no IoC frameworks that support this, so it is not listed as a best practice.
- Setter injection is recommended by the Spring framework which is one of the most popular IoCs. This was largely due to the clumsy indexing in Spring 1 and in Spring 2.0. The Spring framework is introducing named constructor arguments so their recommendation of using setters will likely be removed.

While a service cannot use setter injection to bind references to immutable variables, it should declare `private` variables for reference binding. Furthermore, it should never publish interfaces that expose setter methods used for service injection. While this best practice does not exclude potential bugs within the service class itself, it prevents other classes from corrupting the variables that are bound to dependency references.

In summary, both constructor injection and setter injection have their advantages and disadvantages. This document argues that constructor injection should be first considered as it is more aligned with legacy components while offering greater ease in testing and debugging. The components written to constructor injection have well-formed instances which helps in system robustness in a non-IoC environment. But in situations when a legacy component is already a JavaBeans component, or has many constructor parameters as well as a deep class hierarchy, then setter injection should be considered. Regardless of the injection technique chosen, it is recommended that the component be investigated for possible design flaws. It is recommended that components use constructor injection for immutable data and provide setters for mutable data.

Best Practice 3: Use static factory injection.

The service references injected by both constructor injection and dependency injection are singleton services (AKA singleton beans), meaning each of these services has only one shared instance and all requests to each of them will cause the same instance being injected. To request the injection of non-singleton services, a client service may use static factory injection. Assume `propertyC` in Example 2a needs to be a reference to a non-singleton service, then the setter method of `propertyC` will pass in a brand new service instance of type `ServiceC` even if other instances of that service had already been created. The examples below show a minimalist static factory injection setup.

The Example 3a depicts the `ServiceCFactory` class with the static method `createServiceC` method that creates and returns an instance of the `ServiceCImpl` class that implements the `ServiceC` interface.

```
package packageC.pathC;
...
public class ServiceCFactory {
...
    public static ServiceC createServiceC() {
        return new ServiceCImpl();
    };
...
}
```

Example 3a: The `ServiceCFactory` class

Example 3b depicts the configuration of static factory injection of `serviceC` into `serviceB` based on the Spring format [3]. As shown in Example 3b, the value of service `serviceC's` attribute `factory-method`, is `set to createServiceC`. This means that `serviceC` is created by calling the static `createServiceC` method on the `ServiceCFactory` class.

```
...
<bean id="serviceB" name="serviceB"
                    class="packageB.pathB.ServiceBImpl">
    <property name="propertyC" ref="serviceC"/>
    <property name="propertyD" ref="serviceD"/>
    <property name="name" value="aName"/>
</bean>

<bean id="serviceC"
      class="packageC.pathC.ServiceCFactory"
      factory-method="createServiceC">
...
</bean>

<bean id="serviceD" name="serviceD"
                    class="packageD.pathD.ServiceDImpl">
...
</bean>
...
```

Example 3b: Configuration of static factory injection of `serviceC`.

Note that Example 3a depicts a very simplistic service factory class. A more sophisticated setup may include an abstract factory class with static methods each returning a subclass modeling some specialized behavior. The methods that create the services may each

redirect to an abstract method implemented by a subclass. This technique uses a generalized factory with specialized subclasses to create service instances with different behavior.

Static factory injection allows for injection of non-singleton services. The client service is oblivious to whether its injected references are singleton or not. Static factory injection is highly customizable and hence can be applied to call legacy static factories without being invasive. However unlike singleton services, non-single services may not be lifecycle-managed by an IoC container. If this is the case, then it is the client's responsibility to manage the lifecycle of its non-singleton service references. As a result, non-singleton services with lifecycle make system less robust and its use should be avoided unless necessary.

Any service code unless necessary should not assume it is a singleton or otherwise. It should rely on the framework to create singleton instance or multiple instances based on configuration. This provides flexibility for service lifestyle (singleton or otherwise) changes in the future.

Best Practice 4: Use instance factories to create objects.

Instance factory is a design pattern in which object creation is done by factories, not by the `new` method called from the requesting classes. Example 4a depicts a typical factory class `ServiceAFactory` factory class has a static method `getInstance()` which creates and return the singleton instance of itself, and an instance (non-static) method `createServiceA()`that returns a new instance of the `ServiceAImpl` class.

```
...
public class ServiceAFactory {

    static ServiceAFactory instance;
...
    public static ServiceAFactory getInstance() {
        if (instance == null)
            instance = new ServiceAFactory();
        return new instance();
    }

    public ServiceA createServiceA() {
        return new ServiceAImpl();
    }
...
}
```

Example 4a: The `ServiceAFactory` Class.

A requesting class uses `getInstance()`method to obtain the singleton instance of the factory class whose `createServiceA()` is called to create a new instance of the `ServiceAImpl` class.

```
...
    ServiceA serviceA =
        ServiceAFactory.getInstance().newInstance();
...
```

Example 4b: A call to the factory class to create an `ServiceAImpl` instance.

```
...
<bean id="serviceB" name="serviceB"
                    class="packageB.pathB.ServiceBImpl">
    <property name="propertyA" ref="serviceA"/>
    <property name="propertyD" ref="serviceD"/>
    <property name="name" value="aName"/>
</bean>

<bean id="serviceAFactory"
      class="packageA.pathA.ServiceAFactory"
       factory-method="getInstance"/>

<bean id="serviceA" factory-bean="serviceAFactory"
                    factory-method="createServiceA">
...
</bean>

<bean id="serviceD" name="serviceD"
                    class="packageD.pathD.ServiceDImpl">
...
</bean>
...
```

Example 4c: Configuration of static factory and instance factory injection of `serviceA`.

The instance factory can also be used to inject a service dependency. Example 4c depicts the configuration of static factory injection of `serviceA` into `serviceB` based on the Spring format. As shown in Example 4c, the values of service `serviceA's` attributes, `factory-bean` and `factory-method`, are `serviceAFactory` and `createServiceA` respectively. This means that `serviceA` is created by calling the `createServiceA` method on an instance of the `serviceAFactory` factory service. In this case, an instance of `serviceAFactory` is created by call the static method `getInstance()` on the `packageA.pathA.ServiceAFactory` class (Best Practice 3).

Instance factories decouple the requesting classes from the implementation classes of the requested objects. The requesting classes only need to import the interfaces not the actual implementation classes of the requested objects. In example 4b, the request class imports `ServiceA` but not `ServiceAImpl`. This practice makes the system more robust. Instance factories also provide a layer of indirection that can be used to customize and hide complexity in object creation.

In general, service injection (e.g., constructor injection, dependency injection and static factory injection) is the preferred method for coupling between modules. Within a module, both service injection and instance factories can be used for object referencing. The direct calling of the `new` method to create an object should be avoided.

Best Practice 5: Use start and stop lifecycle methods

All services should participate in lifecycle management by implementing a *start* and a *stop* method. While a well-formed services should be ready to use after it is created, many legacy systems make heavy use of an initialization routine outside of constructor. This initialization routine should be mapped in the configuration data as the start method. Services typically hold resources or references to other services. These resources and references should be released or nullified in the stop method in order to avoid unnecessary locking of resources as well as memory leaks.

The actually names of the two methods can be arbitrary as long as they are properly declared in the configuration file. The framework can query the configuration data to get the names of the start and stop methods of a service. It uses Java reflection to invoke the start method after the service is created, and the stop method when the service is being shut down.

```
package packageA.pathA;
...
public class ServiceAImpl {
...
    public void doStart() {
    ...
    }

    public void doStop() {
    ...
    };
...
}
```

Example 5a: The lifecycle methods of the `ServiceAImpl` class

```
...
<bean id="serviceA" name="serviceA"
                    class="packageA.pathA.ServiceAImpl"
                    init-method="doStart"
                    destroy-method="dostop">
...

</bean>
```

Example 5b: Configuration with lifecycle methods of `serviceA`.

Example 5a depicts the service class `ServiceAImpl` with two lifecycle methods `doStart()`and `doStop()`. Example 5b shows the configuration of service `serviceA` with the `init-method` and `destroy-method` attributes mapping to the lifecycle methods of the `packageA.pathA.ServiceAImpl` class (Example 5a) based on the Spring format. The services themselves should not implement any interface in order to be life-cycle managed by the framework. However, frameworks like OSGi and GBean violate this principle by forcing components to implement the framework lifecycle interface.

Examples 5a and 5b is an example of *lifecycle injection* provided by IoC based on the Spring format. The start and stop methods without parameters for lifecycle injection are likely adequate for most services. Some frameworks also provide a lifecycle interface for components to implement. In most cases, components implement a framework lifecycle interface in order to pass in a context object for dynamically locating services or configuration. This pattern creates tight coupling between the components and the framework, making the components less portable to other frameworks. Components should try to minimize this pattern and use instead dependency injection (Best Practices 1, 2 and 3) and configuration injection (Best Practices 9) if possible.

Some frameworks may also support a more sophisticated lifecycle states, particularly at the modularity level. Components should try to minimize their dependency on lifecycle methods other than start and stop. This is because if a component depends on lifecycle methods other than start and stop, it won't run on frameworks that only support these two lifecycle methods.

Best Practice 6: Use interfaces.

All services and objects should be referenced by interfaces, not implementation classes. Using only interfaces to reference imported objects allows the calling class to decouple from the implementation of the referent classes. This way, as long as the interfaces remain immutable, changing implementation of the referent classes will not break the code of the calling class.

Best Practice 7: Use simple types, simple collections (array, collection, list, set, map), or objects types that comply with these practices.

In this document, an object is defined as well-formed if it complies with all the best practice principles listed here. A component is considered robust, well-encapsulated and hence highly portable if it contains only simple types, simple collections (array, collection, list, set, map), or well-formed objects.

Best Practice 8: Do not use framework interfaces and APIs inside components.

This principle speaks to the framework as well as the components. Often time it is the framework that imposes the condition forcing components to implement framework

interfaces or calls framework APIs. For example, the OSGi, GBean, and WebSphere frameworks require a service implement framework lifecycle interface in order for the frameworks to manage its lifecycle. This requirement locks services to one specific framework rendering them not portable to other frameworks.

Framework such as Eclipse uses the locator design pattern to manage its extension contributions. As a result, components must issue Eclipse APIs to locate extensions. Thus, framework code bleeds into component code and makes the latter platform-specific and not portable to other platforms.

For these frameworks, components may not have an alternative to avoid framework code bleed-through. Some frameworks may have options for components to implement framework interfaces or use framework APIs. A well-formed component should avoid these options.

Best Practice 9: Keep specific configuration methods out of the runtime and use configuration injection.

Configuration is the metadata that describes services. Configuration is typically modeled by external files of various declarative formats. Frameworks like OSGi assume a standard format for its configuration data. As a result, components with a different configuration cannot port to these frameworks unless the components refactor their configurations. The framework should not assume any specific methods (e.g., XML languages or file formats) used to configure the components. Instead, the framework should employ separate services that process and convert configuration in its declarative format into a generalized object model, such as an instance of a general configuration class. These services like any other component services are not part of the framework runtime. Instead, **configuration should be injected just like any other services**.

Best Practice 10: Keep dependencies to a minimum.

A well-formed component should minimize its dependencies to other components. A component with too many dependencies will have bloated code either in constructor parameters or the number of setter methods. The readability and maintainability of the code suffers as a result.

Furthermore, every dependency reference can potentially propagate errors from one component to another. A component with too many dependencies is therefore less robust than the one with a fewer number dependencies. For example, some modern frameworks will recursively shutdown services based on the dependency chain in order to protect system integrity. If a component carries a large number of dependencies, failure from any one of them will cause it to shutdown. Reducing dependencies therefore also reduces potential points of failure.

Best Practice 11: Do not use annotations for framework-specific configuration.

Annotations are becoming more popular as Java 5, EJB 3.0 and Spring all offer annotations as an alternative to XML-based configuration. Annotations are convenient because they are right next to the code element that they configure. Annotations are also more concise than XML. But there are heavy prices to pay in using annotations for framework-specific configuration. First, framework-specific annotations tightly couple the code to the underlying framework, making the code un-consumable by other frameworks. This is no different from components calling framework APIs in terms of invasiveness.

Secondly, although annotations are not the code itself, they are part of the source code. Updating annotations due to a version update of framework switch is difficult because it requires the changing and recompilation of the source code. Components should in general avoid annotations that tie them to specific frameworks.

References

[1] http://wiki.apache.org/excalibur/InversionOfControl
[2] http://www.martinfowler.com/articles/injection.html
[3] http://www.springframework.org/docs/reference/beans.html
[4] http://java.sun.com/products/javabeans/docs/spec.html