# Design Patterns: Factory Pattern

By **Alex Theedom**, dzone.com
March 23rd, 2016

The factory pattern is a creational design pattern whose intent is to provide an interface for creating families of related or dependent objects without specifying their concrete classes. The creational logic is encapsulated within the factory which either provides a method for its creation or delegates the creation of the object to a subclass. The client is not aware of the different implementations of the interface or class. The client only needs to know the factory to use to get an instance of one of the implementations of the interface. Clients are decoupled from the creation of the objects.

Often the factory pattern is implemented as a singleton or a static class as only one instance of the factory is required. This centralizes the object creation.

## CDI Framework

In Java EE, we can take advantage of the CDI framework to create objects without knowing the details of their creation. The decoupling occurs as a result of the way Java EE implements inversion of control. The most important benefit this conveys is the decoupling of higher-level-classes from lower level classes. This decoupling allows the implementation of the concrete class to change without affecting the client: reducing coupling and increasing flexibility.

You could consider that the CDI framework itself is an implementation of the factory pattern. The container creates the qualifying object during application start up and injects it into any injection point that matches the injection criterion. The client does not need to know anything about the concrete implementation of the object, not even the name of the concrete class is known to the client.

```
public class CoffeeMachine implements DrinksMachine {
 // Implementation code
}

@Inject
DrinksMachine drinksMachine;
```

Here, the container creates an instance of the **CoffeeMachine** concrete class, it is selected based on its interface **DrinksMachine** and injected wherever the container finds a qualifying injection point. This is the simplest way to use the CDI implementation of the factory pattern. However it's not the most flexible.

What happens if we have more than one concrete implementation of the **DrinksMachine** interface? Which implementation should be injected? **SoftDrinksMachine** or **CoffeeMachine**? The container does not know and so deployment will fail with an “ambiguous dependencies” error.

```
public class CoffeeMachine implements DrinksMachine {
 // Implementation code
}

public class SoftDrinksMachine implements DrinksMachine {
 // Implementation code
}

@Inject
DrinksMachine drinksMachine;
```

So how does the container distinguish between concrete implementations? Java EE gives us a new tool: Qualifiers. Qualifiers are custom annotations that mark the concrete class and the point where you want the container to inject the object.

Returning to our Drinks machine and the two concrete classes of the same type **CoffeeMachine** and **SoftDrinksMachine** we would distinguish them by the use of two qualifier annotations.

We create one qualifier name **SoftDrink**. This will annotate the **SoftDrinksMachine** concrete class and **Coffee** will annotate the **CoffeeMachine** class.

The **@Target** annotation restricts where we can use these qualifiers to mark injection points, in this case on method and field injection points. The annotation with retention policy **RUNTIME** ensures that the annotation is available to the JVM through the runtime.

The possible values for Target are: **TYPE**, **METHOD**, **FIELD**, **PARAMETER**.

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD})
public @interface SoftDrink

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD})
public @interface Coffee
```

The two concrete implementations of the **DrinksMachine** interface are annotated appropriately. The **CoffeeMachine** class is annotated **@Coffee** while the **SoftDrinksMachine** class is annotated **@SoftDrink**.

```
@Coffee
public class CoffeeMachine implements DrinksMachine {
 // Implementation code
}

@SoftDrink
public class SoftDrinksMachine implements DrinksMachine {
 // Implementation code
}
```

Now you annotate the injection points. Use the qualifier **@SoftDrink** to denote where you want the container to inject the **SoftDrinksMachine** instance and the qualifier **@Coffee** where you want the container to inject the **CoffeeDrinkMachine** instance. Now we have made it clear to the container where our concrete implementations should be injected and deployment will succeed.

```
@Inject @SoftDrink
DrinksMachine softDrinksMachine;

@Inject @Coffee
DrinksMachine coffeeDrinksMachine;
```

We have seen how Java EE's CDI framework allows the creation of a concrete class to be decoupled from its point of use. We have seen how qualifiers are used to select the required implementation without the need to know anything about the objects creation.

It is important to remember that the CDI framework will only instantiate POJOs that meet all of the conditions of the managed beans specification JSR 299. But what if the object you want to inject doesn't, does that mean we cannot take advantage of the CDI framework's injection capabilities for classes that don't comply. No, it doesn't. Java EE

provides us with a solution. Let's dive deeper and look at how we can use the CDI framework to inject ANY class of ANY type into an injection point.

## Producer Methods

Java EE has a feature called producer methods. These methods provide a way to instantiate and, therefore, make available for injection objects that don't conform to the managed bean specifications such as objects which require a constructor parameter for proper instantiation. Objects whose value might change at runtime and objects whose creation requires some customised initialization can also be produced ready for injection via a producer method.

Let's have a look at a producer method which produces a List populated with Book objects.

A list of Book objects will be injected into the injection point annotated **@Library**.

An important feature of the producer method is its scope. This will determine when the method is invoked and for how long the object it produces will live.

By default the producer method scope is **@DependentScoped**. This means that it inherits the scope of its client.

We can extend this example further by giving it a wider scope. If we annotate the producer method **@RequestScoped** it will be invoked only once for each HTTP request in which it participate, lasting for the duration of the request.

```
@RequestScoped
@History
@Produces
public List<Book> getLibrary(){
 // Generate a List of books called 'library'
 return library;
}
```

Possible scopes are:

- RequestScoped – HTTP Request Scope
- SessionScoped – HTTP Session Scope
- ApplicationScoped – Shared across users
- ConversationScoped – Interactions with JSF
- DependentScoped – Default, inherits from client

## The Good, Bad and the Ugly

### The Good

It's become very easy to implement as there is very little if any boilerplate code. Injection of created objects is done by the container and simply works magically. Any object regardless of its conformity to JSR299 can be instantiated and made injectable. Qualifiers help with disambiguation and provide a mechanism to filter the concrete implementation that is injected.

### The Bad

If you use the **@Named** annotation you have to deal with its inherent lack of type safety e.g. **@Named("History")** -> **@History**. However if possible you should use custom qualifiers.

## The Ugly

Object creation is hidden which makes execution flow hard to follow, however, your IDE should provide hints as to the location of the producer method or concrete implementation.