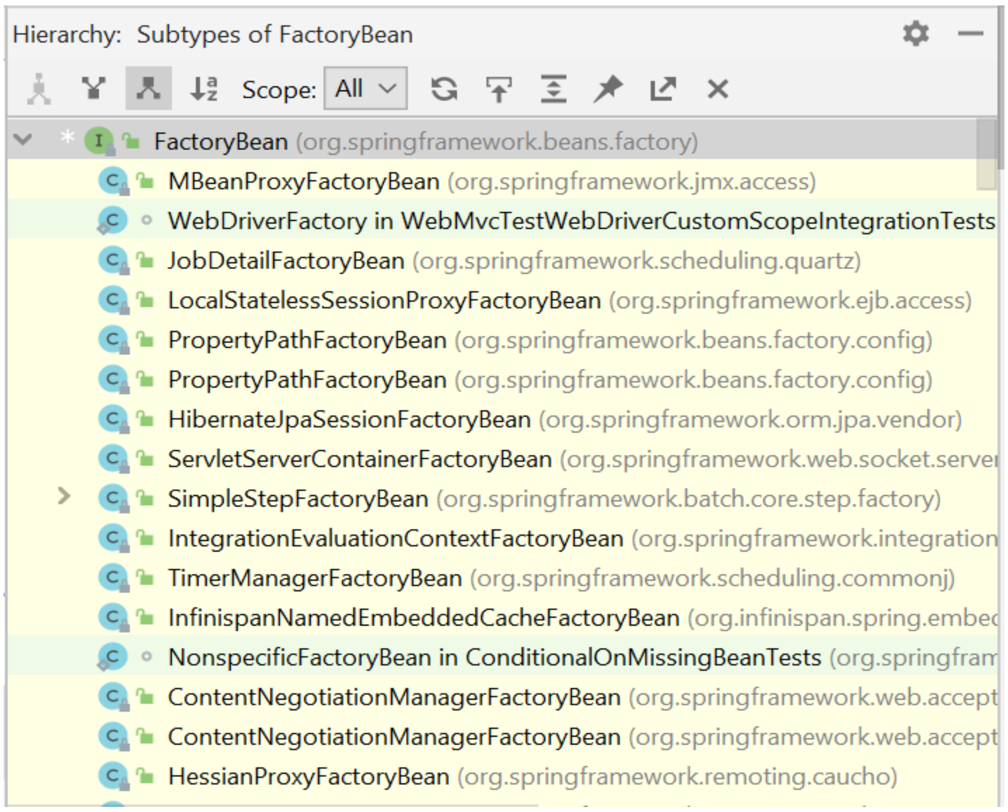
# Design Patterns and First Contribution

## 1. Design Patterns

### Abstract Factory Pattern

The factory pattern is widely used in Java development. The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes. The abstract factory only provides the abstract method. The actual implementation of business logic hides in the concreate implementation. From the implementations, we could find the actual bean factory.



Hierarchy: Subtypes of FactoryBean

Scope: All

```
FactoryBean (org.springframework.beans.factory)
    MBeanProxyFactoryBean (org.springframework.jmx.access)
    WebDriverFactory in WebMvcTestWebDriverCustomScopeIntegrationTests
    JobDetailFactoryBean (org.springframework.scheduling.quartz)
    LocalStatelessSessionProxyFactoryBean (org.springframework.ejb.access)
    PropertyPathFactoryBean (org.springframework.beans.factory.config)
    PropertyPathFactoryBean (org.springframework.beans.factory.config)
    HibernateJpaSessionFactoryBean (org.springframework.orm.jpa.vendor)
    ServletServerContainerFactoryBean (org.springframework.web.socket.server
    SimpleStepFactoryBean (org.springframework.batch.core.step.factory)
    IntegrationEvaluationContextFactoryBean (org.springframework.integration
    TimerManagerFactoryBean (org.springframework.scheduling.commonj)
    InfinispanNamedEmbeddedCacheFactoryBean (org.infinispan.spring.embec
    NonspecificFactoryBean in ConditionalOnMissingBeanTests (org.springfram
    ContentNegotiationManagerFactoryBean (org.springframework.web.accept
    ContentNegotiationManagerFactoryBean (org.springframework.web.accept
    HessianProxyFactoryBean (org.springframework.remoting.caucho)
```

`FactoryBean` is the interface for all beans. It has two methods:

```java
public interface FactoryBean<T> {
    T getObject() throws Exception;
    Class<?> getObjectType();
}
```

`ScannedServiceWrapper` is an implementation of `FactoryBean` and return the actual type for building

```java
@Override
        public Object getObject() throws Exception {
            return cloudServiceConnectorFactory.getObject();
        }

        @Override
        public Class<?> getObjectType() {
            return cloudServiceConnectorFactory.getObjectType();
        }
```

## Adapter Pattern

Adapter is used to convert one type of interface to another type of interface. In other words, adapter pattern works as a bridge between two types of interface.

In spring-boot, `HandlerAdapter` works by this pattern. For example, `DispatcherServlet` will send request to `HandlerAdapter` based on the handler returned by `HandlerMapping`, and `HandlerAdapter` will find the corresponding Handler and execute it. Handler will return a model and view and then `HandlerAdapter` will send the model and view to `DispatcherServlet`.

By this pattern, reuse and extension of handler become easier. In spring, every controller has a `HandlerAdapter`.

```java
public class Dispatcher {
...
private final List<HandlerMapper> mappers;
public boolean handle(ServerHttpRequest request, ServerHttpResponse response) throws IOException {
        for (HandlerMapper mapper : this.mappers) {
            Handler handler = mapper.getHandler(request);
            if (handler != null) {
                handle(handler, request, response);
                return true;
            }
        }
        return false;
    }
...
}
```

> ***Code Explanation***
>
> This part of code shows the details of `Dispatcher`. `mapper` contains the right `handler` for specific request. If we do have a handler for the request, we call handle function to deal with these request.

## Observer Pattern

Observer pattern provides the implicit invocation of the callback functions. When a subscribed event is post in the event bus, the callback function in the subscriber would be called.

In spring-boot, event model is implemented in observer pattern. It's divided into three parts: event, event listener and event publisher.

- `Application Event`

  ```java
  public abstract class ApplicationEvent extends EventObject {
      private static final long serialVersionUID = 7099057708183571937L;
      private final long timestamp;
      public ApplicationEvent(Object source) {
      super(source);
      this.timestamp = System.currentTimeMillis();
      }
      public final long getTimestamp() {
          return this.timestamp;
      }
  }
  ```

  This class is to be extended by all application events, and get the event through `source`

- `ApplicationListener`

  ```java
  public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {
      void onApplicationEvent(E event);
  }
  ```

  All listener should implement this class. And for each type of event, we should implement one type of listener to handle the event. And because of that, `onApplicationEvent()` only needs one parameter, which is `ApplicationEvent` or its child.

- `ApplicationEventPublisher`

  ```java
  public interface ApplicationEventPublisher {
      default void publishEvent(ApplicationEvent event) {
          publishEvent((Object) event);
      }
      void publishEvent(Object event);
  }
  ```

  All `Applicationcontext` should implement this interface and publish events by `publishEvent()`. And specific listener will get the events and execute the business logic.

  ```java
  public interface ApplicationContext extends EnvironmentCapable, ListableBeanFactory, HierarchicalBeanFactory,
          MessageSource, ApplicationEventPublisher, ResourcePatternResolver {
      ...
      String getId();
      String getApplicationName();
      ...
  }

  ```

## Singleton Pattern

Singleton pattern means that there is only one instance for a class which could be accessed globally.  All beans in the Spring/Spring-boot are designed in this pattern. They are responsible for dependency injection for specific type applications.

```java
public interface FactoryBean<T> {
    ...
    default boolean isSingleton() {
        return true;
    }
    ...
}
```

For example, In auto-configuration package, `SharedMetadataReaderFactoryBeanis` designed in this pattern.

```java
private static class CachingMetadataReaderFactoryPostProcessor
            implements BeanDefinitionRegistryPostProcessor, PriorityOrdered {
    ...
    BeanDefinition definition = BeanDefinitionBuilder

.genericBeanDefinition(SharedMetadataReaderFactoryBean.class,
SharedMetadataReaderFactoryBean::new)
                    .getBeanDefinition();
    ...
}
```

> ***Code Explanation***
>
> `definition` is built in a Builder Pattern(will be discussed in next section. It is built with an instance of `SharedMetadataReaderFactoryBean` , and there isn't another instance built in the framework.

## Builder Pattern

Builder Pattern could help user build complex pattern in a easier way. It hides the build process and details. Users could build object only with the feature and context.

As shown above, the build process of `BeanDefinition` is a builder pattern. And `BeanDefinitionBuilder` is in charge of build the real object.

```java
public final class BeanDefinitionBuilder {
    // instance to build
    private final AbstractBeanDefinition beanDefinition;

    // build function, take genericBeanDefinition as example
    public static <T> BeanDefinitionBuilder genericBeanDefinition(Class<T>
beanClass, Supplier<T> instanceSupplier) {
        BeanDefinitionBuilder builder = new BeanDefinitionBuilder(new
GenericBeanDefinition());
        builder.beanDefinition.setBeanClass(beanClass);
        builder.beanDefinition.setInstanceSupplier(instanceSupplier);
        return builder;
    }

    // get built object
    public AbstractBeanDefinition getBeanDefinition() {
        this.beanDefinition.validate();
        return this.beanDefinition;
    }

    // set properties
    public BeanDefinitionBuilder addAutowiredProperty(String name) {
        this.beanDefinition.getPropertyValues().add(name, AutowiredPropertyMarker.INSTANCE);
        return this;
    }
```

> ***Code Explanation***
>
> `BeanDefinitionBuilder` has four parts:
>
> - build function: Build the properties of `BeanDefinition`. Before the process is finished, build function only return `this.builder` for further build operations.
> - build instance: `beanDefinition` is the instance we aim to build.
> - build complete function:  After build process is completed, `getBeanDefinition()` is called and return the `beanDefinition`  we need.
> - property setting function: We could set attributes of the `beanDefinition`  with these methods.

## 2. First Contribution

### Issue

When we looked through Spring-boot, we found this part of code. The if condition `existing == null` is unnecessary, because when `existing != null`, this statement is determined by the second condition, and when `existing = null`, the second condition `!(existing instanceof CompositeProxySelector)` will return True as well. So the value of it statement solely depends on the condition.

```
public class DefaultRepositorySystemSessionAutoConfiguration implements RepositorySystemSessionAutoConfiguration {

    @Override
    public void apply(DefaultRepositorySystemSession session, RepositorySystem repositorySystem) {
        ...
        if (existing == null || !(existing instanceof CompositeProxySelector)) {
            JreProxySelector fallback = new JreProxySelector();
        ...
        }
    }
}
```

In conclude, we could remove `existing == null` to keep a dry and clean code. And this line is updated to:

`if ( !(existing instanceof CompositeProxySelector)) {`. Pull request is [here]().