Design Patterns and First Contribution

- ***Revision details***

1. For 1.1 Factory Pattern

> 1. Revised from *Abstract factory pattern* to *Factory Pattern*, because previous one is not a standard Abstract factory pattern
> 2. Add benefits of Factory Pattern
> 3. Add code example and explanation
> 4. Add usage example

2. For 1.2 Adapter Pattern

> 1. Add benefits for Adapter pattern
> 2. Supply code example.

3. For 1.3 Observer Pattern

> 1. Add benefits for Observer Pattern
> 2. Add code example and code explanation in the comments
> 3. Add usage example

4. For 1.4 Singleton Pattern

> 1. Add benefits for Singleton Pattern
> 2. Add code example
> 3. Add usage example

5. For 1.5 Builder Pattern

> 1. Add benefits for Builder Pattern.
> 2. Modify code example from `BeanDefinitionBuilder.class` to `Health.class` because `BeanDefinitionBuilder.class` is not a standard Builder pattern
> 3. Add usage example

1. Design Patterns

1.1 Factory Pattern

What is Factory pattern and its benefits

Factory pattern is widely used in Java development, which enables clients don't need to realize the concrete process of instantiation and simplifies the implementation of creating object. Clients can get specified object by pass a specified type. It removes the instantiation of actual implementation classes from client code and encapsulates the instantiation in only one class. Factory pattern makes our code more robust, less coupled and easy to extend.

Code Example

`BeanFactory` is a factory, which provides the interface to create different types of beans based on class name.

```
public interface BeanFactory {
    /**
     * Return an instance, which may be shared or independent, of the specified bean.
     */
    Object getBean(String name) throws BeansException;

    /**
     * Return an instance, which may be shared or independent, of the specified bean.
     */
    <T> T getBean(String name, Class<T> requiredType) throws BeansException;
```

`SimpleJndiBeanFactory` a concrete factory, which implements the `BeanFactory` and return the actual type for building

```
public class SimpleJndiBeanFactory extends JndiLocatorSupport implements BeanFactory {
    @Override
    public <T> T getBean(String name, Class<T> requiredType) throws BeansException {
        try {
            if (isSingleton(name)) {
                return doGetSingleton(name, requiredType);
            }
            else {
        //search specified bean's name in the cached map
                return lookup(name, requiredType);
            }
        }
        ...
    }
}
```

Usage Example

Users can get a specified bean object by calling function `getBean` with a class name.

```
RabbitAdmin amqpAdmin = context.getBean(RabbitAdmin.class);
```

```
FilterRegistrationBean<?> registration = this.context.getBean(FilterRegistrationBean.class);
```

1.2 Adapter Pattern

What is Adapter pattern and its benefits

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. It has some benefits as below:

- It enables two or more unrelated interfaces to interact and be compatible.
- It allows reusability of existing functionality.

Code example

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 object of ModelAndView and then `HandlerAdapter` will send the ModelAndView object to `DispatcherServlet` .

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

```
public class DispatcherServlet {
...
    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
            ...

                // Get different Controller by HandlerMapping
                mappedHandler = getHandler(processedRequest);

                // Determine handler adapter for the current Cotroller.
                HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

                // Call the handle function in the controller to handle the request
                mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

                ...
    }
}
```

- *Adapter class*

> It provides the `handle` function for each Controller.

```
public class SimpleControllerHandlerAdapter implements HandlerAdapter {

    //Check whether current handler implement the Cotroller interface
    public boolean supports(Object handler) {
        return (handler instanceof Controller);
    }


  //If current is a controller object, call the handleRequest function in the controller
```

```
    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {

        return ((Controller) handler).handleRequest(request, response);
    }
}
```

> ***Code Explanation***
>
> This part of code shows the details of `DispatcherServlet`. `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.

1.3 Observer Pattern

What is Observer pattern and its benefits

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. It has some benefits as below:

- It supports the principle of loose coupling between objects that interact with each other.
- It allows sending data to other objects effectively without any change in the Subject or Observer classes.
- Observers can be added/removed at any point in time.

Code example

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

- `Application Event`

  ```
  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 listeners 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 class.

- `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();
    ...
}

```

Usage example

- *Step 1: Publisher class receives a subscriber object as input*

```java
public class MongoMappingEventPublisher implements ApplicationEventPublisher {
    /**
    * MongoMappingEventPublisher receives MongoPersistentEntityIndexCreator as input and                    save it to its inner global variable
    */
    public MongoMappingEventPublisher(MongoPersistentEntityIndexCreator indexCreator) {
      ...
      this.indexCreator = indexCreator;
        }
}
```

- *Step2: Publish event to its subscriber*

```
/**
* indexCreator is the subscriber we save at Step 1, so we call the subsriber's      onApplicationEvent function so that subscriber will receive the event and handle it
*/
public void publishEvent(ApplicationEvent event) {
        if (event instanceof MappingContextEvent) {
indexCreator.onApplicationEvent((MappingContextEvent<MongoPersistentEntity<?>, MongoPersistentProperty>) event);
        }
}
```

1.4 Singleton Pattern

What is Singleton pattern and its benefits

Singleton pattern means that there is only one instance for a class which could be accessed globally. It restricts the only one instance to be instantiated. It has some benefits as below:

- Singleton avoids the memory is allocated many times for each instance.
- Singleton prevents other objects from instantiating their own copies of the Singleton object, ensuring that all objects access the single instance
- Flexibility: The class has the flexibility to modify the process of instantiation since the class controls the process of instantiation.

Code example

```
public class ServiceLocator {

    private static ServiceLocator instance;

    /**
    * Hide the constructor to prevent create multiple instances
    */
    protected ServiceLocator() {
        this.classResolver = defaultClassLoader();
        setResourceAccessor(new ClassLoaderResourceAccessor());
    }

        /**
        * Provides a static method to access the singleton in global scope
        */
    public static ServiceLocator getInstance() {
        return instance;
    }

    ...
}
```

Usage Example

```java
private Object getClassResolver() throws IllegalAccessException {
        ServiceLocator instance = ServiceLocator.getInstance();
    ...
}
```

1.5 Builder Pattern

What is Builder pattern and its benefits

Builder pattern is an easy way to construct a complex object. It hides the build process and details. It can be used to replace a constructor with multiple input parameters. Instead, builder pattern allows user to construct the complex object step by step, which simplifies the process construction a complex object.

Code example

- ***Health class***

```java
public final class Health extends HealthComponent {

    private final Status status;

    private final Map<String, Object> details;

    /**
     * Create a new {@link Health} instance with the specified status and details.
     * @param builder the Builder to use
     */
    private Health(Builder builder) {
        Assert.notNull(builder, "Builder must not be null");
        this.status = builder.status;
        this.details = Collections.unmodifiableMap(builder.details);
    }
```

- ***Builder class***

```java
public static class Builder {

        private Status status;

        private Map<String, Object> details;

        /**
         * Create new Builder instance.
         */
        public Builder() {
            this.status = Status.UNKNOWN;
            this.details = new LinkedHashMap<>();
        }

        /**
         * Create new Builder instance, setting status to given {@code status}.
         * @param status the {@link Status} to use
         */
        public Builder(Status status) {
            Assert.notNull(status, "Status must not be null");
            this.status = status;
            this.details = new LinkedHashMap<>();
        }

        /**
         * Create new Builder instance, setting status to given and details
         */
        public Builder(Status status, Map<String, ?> details) {
            Assert.notNull(status, "Status must not be null");
            Assert.notNull(details, "Details must not be null");
            this.status = status;
            this.details = new LinkedHashMap<>(details);
        }

        /**
         * Record detail using given key and value.
         */
        public Builder withDetail(String key, Object value) {
            Assert.notNull(key, "Key must not be null");
            Assert.notNull(value, "Value must not be null");
            this.details.put(key, value);
            return this;
        }

        /**
         * Record details from the given details map. Keys from the given map
         * replace any existing keys if there are duplicates.
         */
        public Builder withDetails(Map<String, ?> details) {
            Assert.notNull(details, "Details must not be null");
            this.details.putAll(details);
            return this;
        }

      /**
         * Create a new Health instance with the previously specified code and details.
         */
        public Health build() {
            return new Health(this);
        }
```

Code Explanation

> For the outer `Health.class`, it has two properties, `Status status` and `Map<String, Object> details`. Health's constructor receives a `Builder` object as input and copy the value from the builder to its own properties.

> For the inner `Builder.class`, it contains the same two properties and provides series of funtions to add the key-value pair to the map `details`. So user doesn't need to realize the inside implementation of how to add to the map `details`.

> When users finish setting the properties, they just need to call `builder.build()` and builder will call the constructor of `Health.class` and copy all the values of properties to the target. Which simplifies the process of constructing.

An example of usage

```
private Health up(Health.Builder builder, Document document) {
        return builder.up().withDetail("version", document.getString("version")).build();
}
```

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. This PR has been merged.