**Factory Pattern**

Factory pattern is a creational pattern. This is used to abstract the process of instantiating the objects during run-time.

**Scenario 1:**

In one of the projects that I worked on called “GameZone” which is a subscription based eCommerce application, I have used the factory pattern to handle different line items based on the incoming request and each line item is resolved during the run time and the corresponding executor methods “doExecution()” (as shown below) are invoked.

|  |
| --- |
| public interface BaseOfferExecutorFactory{  public OfferExecutor getExecutor(String offerType);  }  public class OfferExecutorFactory implements BaseOfferExecutorFactory{  private OfferExecutor executor;  public OfferExecutor getExecutor(String offerType){  switch (offeringType) {  case “RECURRING”:  executor = new RecurringSubscriptionExecutor();  case “RENEWABLE”:  executor = new RenewableSubscriptionExecutor();  default:  executor = null  }  return executor;  }  }  public interface OfferExecutor{  public void doExecution();  }  public class RecurringSubscriptionExecutor implements OfferExecutor{  public void doExecution(){  // processing logic specific to recurring subscription.  }  }  public class RenewableSubscriptionExecutor implements OfferExecutor{  public void doExecution(){  // processing logic specific to renewable subscription.  }  } |

**Scenario 2:**

We used Factory pattern to generate the email payload based in the event.

Example, if the event is a Purchase Event, then the email payload generated is order complete email payload. When the event is Cancel Subscription, then subscription cancellation and expiration email payload is generated. In such cases, based on the event occurred during the run time, the template of the email is resolved and corresponding email payload is generated. Factory pattern fits well in this scenario as well.

Also, when doing the code refactoring, we used factory pattern in few scenarios to eliminate lot of if…elseif…elseif…else conditions. The refactoring is done using factory pattern (with run time polymorphism) and the code looked much cleaner and maintainable.

**Singleton Pattern**

Singleton pattern is generally used where the object is created only once and shared across different threads or an application. This is also a creational pattern.

In one of our projects at Wipro we used singleton pattern in one of the scenarios where there was a need to read the configuration parameters that were loaded during the application boot-up time. This config file can be shared across the application and all the clients (other classes) can access the same object to read the configuration parameters.

In the below sample snippet from one of the applications that I worked, the application config parameters such as Payment URL, Refund URL etc are injected using spring injection by the container.

|  |
| --- |
| import org.springframework.beans.factory.annotation.Value;  public class AppConfig {  @Value("${paymentUrl}")  private String paymentUrl;  @Value("${refundUrl}")  private String refundUrl;  @Value("${auditLoggingFeatureFlag}")  private String auditLoggingFeatureFlag;  @Value("${numOfMaxRetryAttempts}")  private int numOfMaxRetryAttempts;  **private static final AppConfig INSTANCE = new AppConfig();**  **public static AppConfig getInstance() {**  **return INSTANCE;**  **}**  public String getPaymentUrl() {  return paymentUrl;  }  public String getRefundUrl() {  return refundUrl;  }  public String getAuditLoggingFeatureFlag() {  return auditLoggingFeatureFlag;  }  public int getNumOfMaxRetryAttempts() {  return numOfMaxRetryAttempts;  }  } |

In the above example, only one instance is created and will be returned by the getInstance() method. Another variation of the singleton class creation is as shown below. Creation of Enum instance is thread-safe. Hence this is more efficient way of creating the singleton object.

|  |
| --- |
| import org.springframework.beans.factory.annotation.Value;  public **enum** AppConfig {  **INSTANCE**;  @Value("${paymentUrl}")  private String paymentUrl;  @Value("${refundUrl}")  private String refundUrl;  @Value("${auditLoggingFeatureFlag}")  private String auditLoggingFeatureFlag;  @Value("${numOfMaxRetryAttempts}")  private int numOfMaxRetryAttempts;  public String getPaymentUrl() {  return paymentUrl;  }  public String getRefundUrl() {  return refundUrl;  }  public String getAuditLoggingFeatureFlag() {  return auditLoggingFeatureFlag;  }  public int getNumOfMaxRetryAttempts() {  return numOfMaxRetryAttempts;  }  } |
|  |