Question:

What is the design pattern used here:

1)java.lang.Runtime#getRuntime() (http://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html#getRuntime--)

getRuntime() functionality of java.lang.Runtime class uses Singleton design pattern.

The documentation of this class states clearly that -

“Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime method. “

From the source code of this method(java.lang.Runtime#getRuntime() ):

private static Runtime currentRuntime = new Runtime();

public static Runtime getRuntime() {

return currentRuntime;

}

This is the implementation of singleton design pattern. For one application only one instance of Runtime class is created and every time the getRutime method is called, it returns the same instance. It uses eager initialization, i.e. the instance is created before it's used.

This is made singleton since in the whole application only one Runtime environment can be possible.

2)java.util.Calendar#getInstance() (http://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html#getInstance--)

This method uses the Abstract Factory design pattern as it returns object of Calendar type which is an abstract class.

The method is made static so that it can be called directly without creating instance of Calendar class.

Abstract family is used for creating families of related or dependent objects without specifying their concrete classes. For this reason it’s called factory of factories. This method is responsible for creating object of Gregory, Buddhist or Japanese calendar type and return an instance of abstract class Calendar.

getInstance() is an overloaded method where the incoming parameters are the timezone and locale. The calender values based on current time are returned for corresponding timezone and locale. If either timezone or locale is not provided, defaults are set for them through overloading.

Gregorgy, buddhist or japanese type calendar object is returned based on locale.

The overloaded getInstance methods are as follows( from java source code):

public static Calendar getInstance(){

return CreateCalendar(TimeZone.getDefault(),

Locale.getDefault(Locale.Category.FORMAT));

}

public static Calendar getInstance(TimeZone zone){

return createCalendar(zone, Locale.getDefault(Locale.Category.FORMAT));

}

public static Calendar getInstance(Locale aLocale){

return createCalendar(TimeZone.getDefault(), aLocale);

}

public static Calendar getInstance(TimeZone zone, Locale aLocale){

return createCalendar(zone, aLocale);

}

The createCalendar is the method which decides which type of Calendar object to be returned based on parameters provided.

The concrete classes are:

1. GregorianCalendar extends Calendar
2. JapaneseImperialCalendar extends Calendar
3. BuddhistCalendar extends GregorianCalendar

Question:

How does this class javax.servlet.http.HttpServletRequestWrapper (http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequestWrapper.html) implement the Decorator pattern.

The class declaration is:

public class HttpServletRequestWrapper extends ServletRequestWrapper implements HttpServletRequest

public interface HttpServletRequest extends ServletRequest

public class ServletRequestWrapper implements ServletRequest

A design must be open for extensions but closed to modifications. This is the basis of decorator design pattern.

Wrapper classes are used for adding of new functionalities in already existing base class. This way the purpose is solved and base class remains unaffected. For adding or changing functionalities, developers can inherit the wrapper class. Since previous code is unaffected, so this pattern is convenient for testing classes.

Here the HttpServletRequestWrapper is a decorator to HttpServletRequest.

The HttpServletRequestWrapper provides default implementation of each method in the implemented HttpServletRequest. By extending HttpServletRequestWrapper, developers just need to overwrite methods that they wish to change or add new methods. Without the wrapper the HttpServletRequest had to be implemented directly and all its methods had to be defined.

Similarly ServletRequestWrapper is wrapper to ServletRequest class.

The constructor constructs a request object (HttpServletRequestWrapper) wrapping the given request (HttpServletRequest):

public HttpServletRequestWrapper(HttpServletRequest request) {

super(request);

}