DESIGN PRINCIPLES/PATTERN FOUNDATION

Divyansh khatri

Q1.

Creational (abstract factory, builder, singleton, static factory method)

1. Singleton

(a) java.lang.Runtime
(b) java.lang.Desktop

2. Builder

(a) com.google.common.collect.MapMaker

3. Static factory method

(a) java.util.Calendar
(b) java.text.NumberFormat
(c) java.nio.charset.Charset

 4. Abstract factory

(a) javax.xml.parsers.DocumentBuilderFactory
(b) (b) javax.xml.transform.TransformerFactory
(c) javax.xml.xpath.XPathFactory

Structural (adapter, decorator, flyweight)

1. Flyweight

(a) java.lang.Integer
(b) java.lang.Boolean

2. Adapter

(a) java.io.InputStreamReader
(b) (b) java.io.OutputStreamWriter
(c) java.util.Arrays

3. Decorator

(a) java.io.BufferedInputStream
(b) (b) java.io.DataInputStream
(c) java.io.BufferedOutputStream
(d) java.util.zip.ZipOutputStream
(e) java.util.Collections#checkedList()

Behavioural (chain of responsibility, command, iterator, observer, strategy, template method)

1. Chain of responsibility

(a) javax.servlet.FilterChain

2. Command

(a) java.lang.Runnable
(b) java.util.concurrent.Callable

3. Iterator

(a) java.util.Iterator

4. Strategy

(a) java.util.Comparator
(b) javax.servlet.Filter

5. Template method

(a) java.util.AbstractList, java.util.AbstractSet, java.util.AbstractMap
(b) java.io.InputStream, java.io.OutputStream, java.io.Reader, java.io.Writer

6. Observer

(a) java.util.EventListener
(b) java.util.Observer/java.util.Observable

Q2.

1. SessionManager class has objects getting instantiated in the AccessChecker class and that is why it is tightly coupled to the AccessChecker class. Similarly, the object of the AccessChecker class are instantiated in the ServerConfig Class which in turn means that it is tightly coupled to the ServerConfig Class. The ServerConfig class has methods like setting and loading the configuration file, etc, so it might not be practical to create instances of them for the tests. AS the ServerConfig follows the Singleton design, we can override its getInstance method. So, instead of going with the classes, we should go with declaring the interfaces.We can use abstract factory pattern or dependency injection to directly instantiate referencing classes

2.
The interface for ServerConfig will be :

    public interface ServerConfigInterface {

        public String getAccessLevel(User u);

    }

The interface for AccessCheckerInterface will be :

    public interface AccessCheckerInterface {

        public boolean mayAccess(User user, String path);

    }

NOTE: All the code remains the same as provided in the assignment,

Just to get the instance of the session manager, we will use the following command -

Guice.createInjector(new DefaultModule()).getInstance(SessionManager.class).

3.
```
public class MainTest {
    public static void main(String[] args) {

        Module module = new AbstractModule() {
          @Override
          protected void configure() {
             bind(AccessCheckerInterface.class).to(AccessCheckerMock.class);
          }
        };
         SessionManager mgr = Guice.createInjector(module).getInstance(SessionManager.class);
        User user = new User();
        mgr.createSession(user, "any path");
    }
}
```

Q3.

Part 1 and 2 :

We can create the static factory class in the following way -

```
public class Responses {

    public static Response notFoundResponse() {

      return new NotFoundResponse();
   }
    public static Response markdownResponse() {

      return new MarkdownResponse();
}
    public static Response fileResponse() {

      return new FileResponse();
} }
```

One can then use a single implementation class instead of the previous hierarchy of classes:

```
public class Response {

    private String status;

    private Map<String, String> headers;

    private String body;

}
public class Responses {

    public static Response response(String status, Map<String, String> headers, String body) {
```

```java
            return new Response(status, headers, body);

    }

    public static Response file(String status, String path) {

        Path filePath = Paths.get(path);

        HashMap<String, String> headers = new HashMap<String, String>();

        headers.put("content-type", Files.probeContentType(filePath));

        byte[] bytes = Files.readAllBytes(filePath);

        String body = new String(bytes);

        return response(status, headers, body);

}
public static Response notFound() {

        return file("404", app.Assets.getInstance().getNotFoundPage());

}

public static markdown(String body) {

    HashMap<String, String> headers = new HashMap<String, String>();

    headers.put("content-type", "text/html");

    return response("200", headers, Markdown.parse(body).toHtml());

}
}
```