ASSIGNMENT ON DESIGN PATTERNS

Exercise – 1:

Creational:

1) (a) java.lang.Runtime

(b) java.lang.Desktop

Singleton

2) com.google.common.collect.MapMaker

Static Factory Method

3) (a) java.util.Calendar

(b) java.text.NumberFormat

(c) java.nio.charset.Charset

Abstract Factory

4) (a) javax.xml.parsers.DocumentBuilderFactory

(b) javax.xml.transform.TransformerFactory

(c) javax.xml.xpath.XPathFactory

Builder

Structural:

1. (a) java.lang.Integer

   (b) java.lang.Boolean

FlyWeight

2) (a) java.io.InputStreamReader

   (b) java.io.OutputStreamWriter

   (c) java.util.Arrays

Adapter

3. (a) java.io.BufferedInputStream

   (b) java.io.DataInputStream

   (c) java.io.BufferedOutputStream

   (d) java.util.zip.ZipOutputStream

   (e) java.util.Collections#checkedList()

Decorator

Behavioural:

1. (a) javax.servlet.FilterChain

Chain of Responsibility

2. (a) java.lang.Runnable

(b) java.util.concurrent.Callable

Command

3. (a) java.util.Iterator

Iterator

4. (a) java.util.Comparator

(b) javax.servlet.Filter

Strategy

5. (a) java.util.AbstractList, java.util.AbstractSet, java.util.AbstractMap

(b) java.io.InputStream, java.io.OutputStream, java.io.Reader,

java.io.Writer

Template Method

6. (a) java.util.EventListener

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

Observer

Exercise 2:

1) 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.

The object of the AccessChecker class are instantiated in the ServerConfig Class which in turn is tightly coupled to the ServerConfig Class.

ServerConfig follows the Singleton design, so 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:

public interface ServerConfigInterface { public String getAccessLevel(User u); }

The interface for AccessCheckerInterface:

public interface AccessCheckerInterface {
public boolean mayAccess(User user, String path); }

3)

```
public class TestDemo {
    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");
 }

}
```

Exercise 3:

Returning a specific response depends on knowledge of the actual class that implements that kind of response. If a new implementation class needs to be used later: 1. Either the clients of the hierarchy need to be changed to use the new class. 2. Or else, the old classes have to become proxies for the new ones. The first point is clearly undesirable, while the second one keeps an API that is no longer relevant, thus bloating the code base.

```
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)
    {
        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());
       }
}
```

This design has its deficiencies, and the challenge addresses some of them.