Design Patterns Assignment

Himanshu Jain

Exercise 1

Creational

1. Singleton design pattern

2. Static factory method design pattern

3. Abstract factory design pattern

Structural

1.Flyweight design pattern

2.Adapter design pattern

3.Decorator design pattern

Behavioural

1. Chain of responsibility design pattern
2. Command design pattern
3. Iterator design pattern
4. Strategy design pattern
5. Template method design pattern
6. Observer design pattern

Exercise 2

1)

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);

}
```

3)

```
public class MainTest {


        public static void main(String[] args) {

                Module module = new AbstractModule() { @Override


                        protected void configure() {

                                bind(AccessCheckerInterface.class).to(AccessCheckerMock.cl
                        ass);

                        }

                };

                SessionManager mgr =
                Guice.createInjector(module).getInstance(SessionManager.class);

                User user = new User();

                mgr.createSession(user, "any path");

                }

        }
```

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) {

                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());

} }
```