Assignment 5: Design Patterns

Exercise 1

Examine the listed Java APIs (see e.g. https://docs.oracle.com/javase/7/docs/api/ for more information) and identify some of the design patterns present. For each pattern found describe why the API follows it. Each bullet-point group corresponds to one pattern to be identified. Note that most of the patterns have not been covered on lectures. You may need to look them up on the web.

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

1. (a) `java.lang.Runtime`
   (b) `java.awt.Desktop`
2. (a) `com.google.common.collect.MapMaker`
3. (a) `java.util.Calendar`
   (b) `java.text.NumberFormat`
   (c) `java.nio.charset.Charset`
4. (a) `javax.xml.parsers.DocumentBuilderFactory`
   (b) `javax.xml.transform.TransformerFactory`
   (c) `javax.xml.xpath.XPathFactory`

Structural (adapter, decorator, flyweight)

1. (a) `java.lang.Integer`
   (b) `java.lang.Boolean`
2. (a) `java.io.InputStreamReader`
   (b) `java.io.OutputStreamWriter`
   (c) `java.util.Arrays`
3. (a) `java.io.BufferedInputStream`
   (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. (a) `javax.servlet.FilterChain`
2. (a) `java.lang.Runnable`
   (b) `java.util.concurrent.Callable`
3. (a) `java.util.Iterator`
4. (a) `java.util.Comparator`
   (b) `javax.servlet.Filter`
5. (a) `java.util.AbstractList`, `java.util.AbstractSet`, `java.util.AbstractMap`
   (b) `java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader`, `java.io.Writer`
6. (a) `java.util.EventListener`
   (b) `java.util.Observer`/`java.util.Observable`

Exercise 2

Consider a simple server implementation which uses the Singleton Pattern. The following code deals with the creation of new user sessions:

```
public class AccessChecker {

    private static AccessChecker instance;
    public static AccessChecker getInstance() {
        if (instance == null) {
            // create instance
        }
        return instance;
    }

    private ServerConfig config = ServerConfig.getInstance();

    private AccessChecker() {
        // initialization..
    }

    public boolean mayAccess(User user, String path) {
        String userLevel = config.getAccessLevel(user);
        // check if level suffices
    }
}

public class ServerConfig {
    private static ServerConfig instance;
    public static ServerConfig getInstance() {
        if (instance == null) {
            // create instance
        }
        return instance;
    }

    private ServerConfig() {
        // load configuration from file
    }

    public String getAccessLevel(User u) { ... }
}

public class SessionManager {

    private AccessChecker access = AccessChecker.getInstance();
    public Session createSession(User user, String accessedPath) {
        if (access.mayAccess(user, accessedPath)) {
            return new Session(user);
        } else {
            throw new InsufficientRightsException(user, accessedPath);
        }
    }
}
```

Your job is to write unit tests for the `createSession` method. The following exercises can be done either on paper or in an IDE; we provide pre-configured projects for Eclipse and IntelliJ IDEA.

1. Why is it hard to create proper unit tests for the current implementation? Which options are available for solving this problem?

2. Perform a refactoring to make the code more easily testable. Specifically:

   - Create and use interfaces instead of directly referencing classes where possible.
   - Remove the current Singleton Pattern implementation and use dependency injection with Guice instead.

   > Guice is a dependency injection framework. It allows the user to create so-called modules which specify which interfaces are bound to which implementations, and can inject instances of those implementations into annotated fields or constructors at runtime.
   >
   > To use Guice in a program, perform the following steps:
   >
   > – Extend the abstract class `AbstractModule` to create a module that will later hold our class bindings.
   > – In the new module, override the `void`-method `configure()` which defines the bindings.
   > – To bind an interface to an implementation, call the methods `bind(Interface.class).to(Implementation.class)`.
   > – Annotate class constructors with `@Inject` to tell the injector that it should use this constructor to create instances of the class, and automatically create the required arguments.
   > – Annotate classes with `@Singleton` to tell the injector it should create only one instance of them.
   > – To get an instance of an interface or a class from the injector, call `Guice.createInjector(module).getInstance(ClassOrInterface.class)`.
   >
   > For some code examples, have a look at https://github.com/google/guice/wiki/GettingStarted.

3. Using the new infrastructure, write a `main()` function that checks if `createSession` returns the correct exception if the given user does not have sufficient rights to access the given path. Create and use mock implementations of the required interfaces.

Exercise 3

A web application can return one of many kinds of HTTP responses to the user-agent.

```
public interface Response {
    String getStatus();
    Map<String, String> getHeaders();
    String getBody();
}

public class FileResponse implements Response {
    public FileResponse(String path) {
        this.path = Paths.get(path);
    }

    @Override
    public String getStatus() {
        return "200";
    }

    @Override
    public Map<String, String> getHeaders() {
        HashMap<String, String> headers = new HashMap<String, String>();
        headers.put("content-type", Files.probeContentType(path));
        return headers;
    }

    @Override
    public String getBody() {
        byte[] bytes = Files.readAllBytes(path);
        String body = new String(bytes);
    }

    private Path path;
}

public class NotFoundResponse extends FileResponse {
    public NotFoundResponse() {
        super(app.Assets.getInstance().getNotFoundPage());
    }

    @Override
    public String getStatus() {
        return "404";
    }
}

public class MarkdownResponse implements Response {
    public MarkdownResponse(String body) {
        this.body = body;
    }

    @Override
    public String getStatus() {
```

```
        return "200"
    }

    @Override
    public Map<String, String> getHeaders() {
        HashMap<String, String> headers = new HashMap<String, String>();
        headers.put("content-type", "text/html");
        return headers;
    }

    @Override
    public String getBody() {
        return Markdown.parse(body).toHtml();
    }

    private String body;
}
```

And so on. If an application would like to return a "404 Not Found" response it would do something like:

```
return new NotFoundResponse();
```

The classes `MarkdownResponse`, `NotFoundResponse` and `FileResponse` all implement the same interface and may themselves be considered implementation details. In order to improve the maintainability, your task is to

1. Identify the coupling issues faced by clients of this hierarchy.
2. Try to remedy the coupling problems by applying the static factory method pattern.
3. Replace the hierarchy of classes with a single implementation representing all possible responses.