Assignment 5: Design Patterns

Exercise 1

Examine the listed Java APIs 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. 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) 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) java.io.OutputStreamWriter

(c) java.util.Arrays

3. decorator :

   (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. 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

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();

        public 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;
        private static String configFilePath = "..."; public static
        ServerConfig getInstance() { if (instance == null) { //
        create instance
                } return instance;
        }

        public ServerConfig() {
                // load configuration from file
                // validate
        }

        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?

   SessionManager is intertwined with AccessChecker, which is intertwined with ServerConfig. Since the latter conducts some complex setup operations, making instances of them for the tests might not be feasible. Since the getInstance methods are static and cannot be overridden, the Singleton Pattern as implemented here makes it difficult to switch out classes for tests. By using interfaces instead of concrete classes as field types, coupling can be minimised. Then there's the question of how to instantiate the fields without referencing classes explicitly. The abstract factory pattern or dependency injection are two possible solutions to this problem.

2. Perform a refactoring to solve those problems. Specifically:

   - Create and use interfaces instead of directly referencing classes where possible.
   - The problem remains how and where the dependencies are allocated. Two solutions for this are the abstract factory pattern and dependency injection.
     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.

     Remove the current Singleton Pattern implementation and use dependency injection with Guice instead. You can use the following methods and annotations:

     – 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.

Build the IAccessChecker and IServerConfig interfaces:

```
public interface IAccessChecker {

   public boolean mayAccess(User user, String path);

}

public interface IServerConfig {

   public String getAccessLevel(User u);

}
```

Enable the interfaces to be implemented by the implementation groups. Make a note of them as singletons. Delete the getInstance methods from the code. Create function Object() { [native code] } parameters for all dependencies and annotate the function Object() { [native code] } that Guice can use:

```
 @Singleton

public class AccessChecker implements IAccessChecker {

   private IServerConfig config;

   @Inject

   public AccessChecker(IServerConfig config) {

      this.config = config;

   }

}

@Singleton

public class ServerConfig implements IServerConfig {

}
```

Make a module to link the implementation classes to the new interfaces:

```
public class DefaultModule extends AbstractModule {
   @Override
   protected void configure() {
      bind(IAccessChecker.class).to(AccessChecker.class);
      bind(IServerConfig.class).to(ServerConfig.class);
   }
}
```

In the SessionManager, delete the getInstance calls, render the dependencies clear in the function Object() { [native code] }, and label them as injected:

```
public class SessionManager {
   private IAccessChecker access;
   @Inject
   public SessionManager(IAccessChecker access) {
      this.access = access;
   }
   public Session createSession(User user, String accessedPath) {
      if (access.mayAccess(user, accessedPath)) {
      }
   }
```

Clients may either insert an instance of the session manager into a field of another class or call it directly.

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

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.

```
}
```

Develop a mock IAccessChecker implementation and a new module that connects the interface to this class. This module can be used to evaluate the method:

```
public class SessionManagerTest {
  public static void main(String[] args) {
    Module module = new AbstractModule() {
      @Override
      protected void configure() {
        bind(IAccessChecker.class).to(AccessCheckerMock.class);
      }
    };
    SessionManager mgr
= Guice.createInjector(module).getInstance(SessionManager.class);
    User user = new User();
    try {
      mgr.createSession(user, "some path");
      assert false;
    } catch (InsufficientRightsException e) {
      System.out.println("Success!");
    }
  }
}
public class AccessCheckerMock implements IAccessChecker {
  @Override
  public boolean mayAccess(User user, String path) {
    return false;
  }
}
```

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 subclasses of Response do not extend its interface and therefore they are just an implementation detail. In order to improve the maintainability, your task is to

1. Identify the coupling in the usage of this hierarchy.
2. Try to remedy the coupling problems by applying the static factory methodpattern.
3. Replace the hierarchy of classes with a single implementation representing all possible responses.

Returning a particular answer necessitates familiarity with the class that implements that response. If a different class is needed to return this form of answer later, then:

1. either the hierarchy's clients must be modified to use the new class;

2. or the old class must be converted into a proxy for the new one.

The first argument is obviously unacceptable, while the second retains an API that is no longer usable, bloating the code base. This can be remedied by adding a static factory:

```
public class Responses {

  public static Response markdownResponse() {

   return new MarkdownResponse();

  }
```

```
   public static Response fileResponse() {

    return new FileResponse();

   }

   public static Response notFoundResponse() {

      return new NotFoundResponse();

   }

}
```

Client code that explicitly called constructors (e.g. return new NotFoundResponse();) will now call the factory method (return Responses.notFoundResponse();), and the factory method can be modified if a different class is needed.

Instead of the previous hierarchy of classes, a single implementation class may be used:

```
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));
      5
      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());
   }
}
```