Assignment 5: Design Patterns (solution)

Exercise 1

Mostly taken from BalusC' answer at `http://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns`

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

1. Singleton
   (a) `java.lang.Runtime`
   (b) `java.awt.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

1. SessionManager is tightly coupled to AccessChecker, which in turn is coupled to ServerConfig. The latter performs some complex set up operations (loading configuration data from an outside file), so it might not be practical to create instances of them for the tests. The Singleton Pattern as implemented here does not allow us to easily exchange the classes for tests, since the `getInstance` methods are static and cannot be overridden.

   Coupling can be reduced by using interfaces instead of concrete classes as the types of fields. Then the problem remains how the fields can be instantiated without directly referencing classes. Possible options for solving this issue are the abstract factory pattern or dependency injection.

2. Create interfaces IAccessChecker and IServerConfig:

```
public interface IServerConfig {
    public String getAccessLevel(User u);
    // ...
}
public interface IAccessChecker {
    public boolean mayAccess(User user, String path);
    // ...
}
```

   Let the implementation classes implement the interfaces. Mark them as singletons. Remove the `getInstance` methods. For all dependencies, create constructor parameters and annotate the constructor to be used by Guice:

```
@Singleton
public class AccessChecker implements IAccessChecker {
    private IServerConfig config;
    @Inject
    public AccessChecker(IServerConfig config) {
        this.config = config;
        // initialization..
    }
    ...
}
@Singleton
public class ServerConfig implements IServerConfig {
    ...
}
```

   Create a module that binds 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);
    }
}
```

Remove the `getInstance` calls in the SessionManager, make the dependencies explicit in the constructor and mark them to be 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)) {
            ...
    }
    // ...
}
```

To get an instance of the session manager, clients can either inject it into a field of a different class or call

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

3. Create a mock implementation of the `IAccessChecker` interface and a new module that binds the interface to this class. Test the method using this module:

```
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, "any path");
            assert false; // we should not get here
        } catch (InsufficientRightsException e) {
            System.out.println("Success!");
        }
    }
}
public class AccessCheckerMock implements IAccessChecker {
    @Override
    public boolean mayAccess(User user, String path) { return false; }
    // ...
```

```
    }
```

Exercise 3

Returning a specific response depends on knowledge of the actual class that implements that kind of response. If later, a different class must be used to return this kind of response, then:

1. either the clients of the hierarchy need to be changed such that they use the new class;
2. or else, the old class has to become a proxy for the new one.

The first point is clearly undesirable, while the second one keeps an API that is no longer relevant, thus bloating the code base.

We can rectify this situation by introducing a static factory:

```
public class Responses {
    public static Response notFoundResponse() {
        return new NotFoundResponse();
    }

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

    public static Response fileResponse() {
        return new FileResponse();
    }
}
```

Client code which called constructors directly (e.g. `return new NotFoundResponse();`) can then call the factory method instead (`return Responses.notFoundResponse();`, and the factory method can be changed if a different class has to be used.

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

```
public class Response {
    ...  // constructor and accessors omitted for clarity
    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());
    }
}
```