**Question 1:**

**Section a:**

1. We downloaded Mockito. It can be found here:

<https://github.com/mockito/mockito>

1. This tool provides test writers with the ability to isolate the code that they are trying to test from its dependencies.
2. There were 94,787 lines in this Git repository. We obtained this number by running the command ‘git ls-files | xargs wc -l’ which we found here: <https://gist.github.com/mandiwise/dc53cb9da00856d7cdbb>

**Section b:**

1) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_BEGIN OUTPUT\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Found 33 files that possibly contain design patterns.

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\configuration\injection\MockInjectionStrategy.java

Possible patterns: Strategy

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\creation\instance\InstantiatorProvider2Adapter.java

Possible patterns: Adapter

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\creation\instance\InstantiatorProviderAdapter.java

Possible patterns: Adapter

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\creation\util\MockitoMethodProxy.java

Possible patterns: Proxy

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\handler\MockHandlerFactory.java

Possible patterns: Factory

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\invocation\DefaultInvocationFactory.java

Possible patterns: Factory

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\invocation\MatcherApplicationStrategy.java

Possible patterns: Strategy

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\junit\ExceptionFactory.java

Possible patterns: Factory

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\matchers\apachecommons\EqualsBuilder.java

Possible patterns: Builder

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\runners\RunnerFactory.java

Possible patterns: Factory

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\session\DefaultMockitoSessionBuilder.java

Possible patterns: Builder

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\session\MockitoLoggerAdapter.java

Possible patterns: Adapter

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\session\MockitoSessionLoggerAdapter.java

Possible patterns: Adapter

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\internal\verification\VerificationModeFactory.java

Possible patterns: Factory

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\invocation\InvocationFactory.java

Possible patterns: Factory

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\session\MockitoSessionBuilder.java

Possible patterns: Builder

C:\Users\grayd\Documents\ESOF\mockito\src\main\java\org\mockito\verification\VerificationStrategy.java

Possible patterns: Strategy

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockito\InvocationFactoryTest.java

Possible patterns: Factory

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockito\StateMaster.java

Possible patterns: State

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockito\exceptions\base\StackTraceBuilder.java

Possible patterns: Builder

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockito\exceptions\base\TraceBuilder.java

Possible patterns: Builder

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockito\internal\InvalidStateDetectionTest.java

Possible patterns: State

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockito\internal\handler\MockHandlerFactoryTest.java

Possible patterns: Factory

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockito\internal\invocation\InvocationBuilder.java

Possible patterns: Builder

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockito\internal\invocation\MatcherApplicationStrategyTest.java

Possible patterns: Strategy

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockito\internal\junit\ExceptionFactoryTest.java

Possible patterns: Factory

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockito\internal\matchers\apachecommons\EqualsBuilderTest.java

Possible patterns: Builder

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockito\internal\progress\VerificationModeBuilder.java

Possible patterns: Builder

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockito\internal\session\DefaultMockitoSessionBuilderTest.java

Possible patterns: Builder

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockitousage\bugs\BridgeMethodsHitAgainTest.java

Possible patterns: Bridge

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockitousage\junitrule\RuleTestWithFactoryMethodTest.java

Possible patterns: Factory

C:\Users\grayd\Documents\ESOF\mockito\src\test\java\org\mockitousage\puzzlers\BridgeMethodPuzzleTest.java

Possible patterns: Bridge

C:\Users\grayd\Documents\ESOF\mockito\subprojects\android\src\main\java\org\mockito\android\internal\creation\AndroidLoadingStrategy.java

Possible patterns: Strategy

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_END OUTPUT\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2) This tool looks for instances of design patterns by looking over all of the files in the project, and then comparing the names of those files to a list of design pattern names stored in a .xml file. If the file name contains the name of one of the design patterns, then the program writes out the name of the file and says that it could possibly contain that design pattern.

3) I think that the process used by this tool is a start. It does seem like a good idea to take an initial pass over the code and find any places where the code may refer to a design pattern. However, I think that this approach is flawed because it seems like an oversimplification of a complicated problem.

The success of this program is based entirely on whether or not the individual writing the coder was consistent in naming his files with the names of the design patterns he included in them. This method begins to break down as soon as the coder implements design patterns in his code that he does not include in the file names. This is a very real possibility. Take for example a file that has two design patterns in it. Including both patterns in the file name may not be practical.

This method further breaks down when the coder uses the names of design patterns to refer to things that are not design patterns. For instance, say that a coder is writing a program to model a household electrical circuit. He could use the name “Adapter” to refer to a component in an electrical circuit, without even thinking about the adapter pattern.

Writing a program to detect design patterns in an efficient and thorough manner would be difficult. However, if I were tasked with this, I think that I would create a graph out of the tree. I would have two types of nodes in the tree, one to represent abstract classes and interfaces, and the other to represent non-abstract classes. There would also be two types of relationships in this graph, “IS-A” and “HAS-A”. “IS-A” relationships would represent inheritance relationships, and “HAS-A” relationships would represent associations and dependencies.

I would have binary decision trees for each pattern. These would be stored in a .xml file so that they are easily extensible by end users. The point of the decision tree would be to identify if a design pattern existed in a graph by answering “yes” or “no“ to a series of questions. The .xml would also have a configuration for each tree of whether to begin with the interfaces and abstract classes, or to begin with the non-abstact classes.

When my pattern-spotting program was run, it would create the graph of the classes by running through the code provided as an input. It would then check all of the interfaces, abstract classes, and non-abstract classes against the relevant binary trees. If the binary trees resulted in a “yes” decision, my code would add that design pattern to a “results list”. If my code did not detect the design pattern, it would not be added to the “results list”. The contents of the “results list” would then be printed out.

**Question 2:**

**Section a:**

1. I uploaded my files to my Github account by first creating a new repository on Github called ESOF322. I initialized it with a README file that told users what the repository was about. I then created an ssh key and added it to my Github account so that I could push and pull without having to enter my username and password. I had all of my homework files saved in a directory called “softeng” on my computer. I moved all of my homework files out of the “softeng” directory, and then deleted it. I cloned down the ESOF322 respository and named it “softeng”. I then placed all of my homework files inside of the “softeng” repository. I added a “.gitignore” file to avoid commiting files that I did not want to commit. I then added my changes by running the command “git add .”. I then committed my changes with “git commit -am ‘uploading hw’s 1-4’”. I then ran “git push origin master” to push my changes to the remote repository.
2. Commands I used:
   1. git status
   2. git add dummy.txt
   3. git commit -am “Added dummy.txt”
   4. git push origin master

