# Software Engineering Design 2015

* + 1. Refactoring is the iterative process of modifying and improving the design of some code without changing the external behaviour.
    2. Automated testing allows the programmer to identify when it is possible to refactor code safely, as they can see when code is working as intended and can then begin the process of simplifying or modifying the underlying structure.

d

Automated tests also provide assurances that ro

* + 1. WILT stands for ‘Whitespace Integrated over Lines of Text’ and is a measure of complexity within code. A high WILT indicates that a program features many levels of indentation, and subsequently many nested conditionals and branches. A high WILT score could be improved by breaking out code into different methods, thus separating responsibility of different parts of the code.
  1. DataFileJeffersonTransform extends the JeffersonTransform class which introduces tight coupling between the two - composition would be preferred over inheritance. Further, using the Singleton pattern for InputFile and OutputFile also creates a lot of interdependency between the Singleton class itself and its collaborators, since it makes it very difficult to switch out implementations on the fly. In readInput() and writeOutput() in the DataFileJeffersonTransform class, assumptions are made about the internal structure of InputFile and OutputFile, by the use of external iterators. It would be preferable to use an internal iterator or separate method within those classes to reduce any coupling issues caused by this (for example, if the implementation of these classes changed). Finally, specific references have been made to the InputFile and OutputFile classes, which is undesirable and it would be better if it was replaced with interfaces to specify the roles of said classes. This would make it easier to switch out the classes with alternate implementations.  
     ---  
     The JeffersonTransform class should also only apply the calculation itself, without having to worry with I/O tasks. Coupling between the calculation and the I/O process is not required and leads to fragility of the code. (sorry for my bad writing skillz)
  2. For the tests, you can now mock InputReader and OutputWriter and test TransformApplier in isolation. Also Strategy pattern has been used to apply different kinds of Transform. Dunno if this was required by the question, but seems like a good refactor to do in this case.

* + 1. Ca (Afferent Coupling): A measure of the responsibility of a class or module. It refers to how many other classes or modules depend on it.  
       Ce (Efferent Coupling): A measure of the independence of a class or module. It refers to how many other classes or modules that it depends on.
    2. A module with high Ca but low Ce might be something like a Utilities class. It would be relatively self contained and contain highly reusable code which has been used by other parts of the codebase. This code would be easiest to reuse since it has relatively few dependencies.  
       This code is also “stable”: this means that it should not have many reasons for changing, as changing it could lead to breaking other code that uses it.
    3. A module with high Ce but low Ca might be something like a Controller class. This is code that other classes do not depend on, but which itself has lots of dependencies. This code would be harder to reuse in different context because its dependencies would also have to be present for the code to be functional.  
       This code is also “unstable”: this means that it can easily be changed without affecting other parts of the system.

c.

An adapter would remove the dependence of the system on that particular central pricing system, so that in the event it changes (or if we decide to swap it with another pricing system), it would not break the rest of the system.

B. *Doesn’t it ask for UML sequence diagram not code??*

public class Model

{

private View view;

// Business logic

void updateModel(Information info)

{

//request information from info

view.updateDisplay(this);

}

}

public class View implements Updatable

{

public View()

{

JFrame frame = new JFrame();

//...

}

@Override

public void updateDisplay(Model model)

{

//update Jframe according to information in model

}

}

public class Controller

{

View view = new View();

Model model = new Model(view);

public static void main(String[] args)

{CentralBankInfo info = new CentralBankInfo();

info.requestInfo();

//Information is an interface,

//CentralBankInfo is the actual class for request

while (true)

{

model.updateModel(info);

//sleep(1000)

}

}

}

d)

Unit tests are used to test the behaviour of Individual components of a system. E.g. Check the model behaves correctly/as expected.

Integration tests are used to test that the entire system is connected correctly (each part communicates with the others as they are supposed to).

To test that the components of the system communicate correctly via the use of mocking, one example is checking if the adapter is working with a mock central pricing system.

Therefore this allows us to design our system using a hexagonal structure, as we only use unit tests to ensure that core components work without being dependent on other objects or third party APIs. Then to ensure that the objects communicate correctly(testing our adapters) we use integration tests with mocking, this makes it easier to modify the system if we choose to change third party components or core components.