










Design Patterns

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

 armahmood786@yahoo.com
 alphapeeler.sf.net/pubkeys/pkey.htm
 pk.linkedin.com/in/armahmood
 www.twitter.com/alphapeeler
 www.facebook.com/alphapeeler
 abdulmahmood-sss  alphasecure
 armahmood786@hotmail.com
 <http://alphapeeler.sf.net/me>

 alphasecure@gmail.com
 <http://alphapeeler.sourceforge.net>
 <http://alphapeeler.tumblr.com>
 armahmood786@jabber.org
 alphapeeler@aim.com
 mahmood_cubix  48660186
 alphapeeler@icloud.com
 <http://alphapeeler.sf.net/acms/>

Object dependencies, Coupling& Cohesion

Lecture 03

Design Tip of the Day

Reduce Class Dependency

- Simpler Structural Design
- Simpler Behavioral Design
- Maintainable Code
- Easy Extensibility

Objects and Classes

Object
Recognition

Class
Creation

Top Down
Approach

Bottom Up
Approach

Abstraction

Dependency

Object Dependency

There are number of ways an object can be used

It depends on

- which class is creating an object
- when the object is being created, and
- where it is being created

We need to understand 3 factors

- Scope
- Creator class
- Location and event of creation

Object Dependency

Scope

- The object is defined at the class level
- The object is defined at the method level

Creation

- The class is creating the object where it is defined
- Another class is creating the object and the reference is provided to the class where the object is defined

Location of creation

- Constructor
- Within the method

Object Dependency Examples

```
class ClassA
{
    private ClassB classB = new ClassB();
}
```

Object Dependency Examples

```
class ClassA
{
    private ClassB classB;

    public ClassA()
    {
        classB = new ClassB();
    }
}
```


Object Dependency Examples

```
class ClassA
{
    private ClassB classB;

    public ClassA()
    {
    }

    public void processClassB()
    {
        classB = new ClassB();
    }
}
```

Object Dependency Examples

```
class ClassA
{
    private ClassB classB;

    public ClassA()
    {
        classB = Factory.GetClassB();
    }
}
```

Object Dependency Examples

```
class ClassA
{
    private ClassB classB;

    public ClassA()
    {
    }

    public void processClassB()
    {
        classB = Factory.GetClassB();
    }
}
```

Object Dependency Examples

```
class ClassA
{
    private ClassB classB;

    public ClassA(ClassB objClassB)
    {
        classB = objClassB;
    }
}
```

Object Dependency Examples

```
class ClassA
{
    private ClassB classB;

    public ClassA()
    {
    }

    public void processClassB(ClassB objClassB)
    {
        classB = objClassB;
    }
}
```

Object Dependency Examples

```
class ClassA
{
    public ClassA()
    {
    }

    public void processClassB()
    {
        ClassB classB;

        classB = new ClassB();
    }
}
```

Object Dependency Examples

```
class ClassA
{
    public ClassA()
    {
    }

    public void processClassB()
    {
        ClassB classB;

        classB = Factory.GetClassB();
    }
}
```

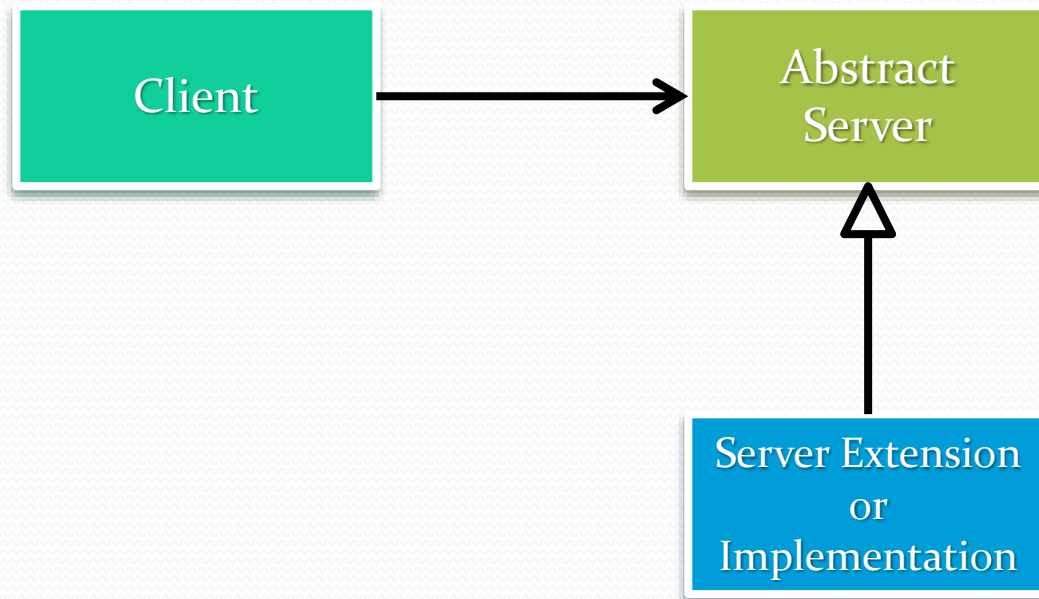
Object Dependency Examples

```
class ClassA
{
    public ClassA()
    {
    }

    public void processClassB(ClassB objClassB)
    {
        ClassB classB;

        classB = objClassB;
    }
}
```


Software Design



Definition of Coupling

- Coupling is the degree of interaction between two modules
- There are six kinds of module coupling
 - Content Coupling
 - Common Coupling
 - External Coupling
 - Control Coupling
 - Stamp Coupling
 - Data Coupling

Content Coupling

- Two or more modules exhibit **content coupling** if one refers to the “inside” – the “internal” or “private” part – of the other in some way.
- occurs when you have one instance stored inside another instance, and you modify the inner instance from the outer instance in a way that isn't intended or transparent.
- also known as **pathological coupling**.

```
public class Line
{
    private Point start, end;
    ...
    public Point getStart() { return start; }
    public Point getEnd() { return end; }
}

public class Arch
{
    private Line baseline;
    ...
    void slant(int newY)
    {
        Point theEnd = baseline.getEnd();
        theEnd.setLocation(theEnd.getX(),newY);
    }
}
```

Common Coupling

- Two or more modules exhibit **common coupling** if they refer to the same global data area – that is, to something that corresponds to a data store on a “register” that must be shared by several processes
- also known as **global coupling**.

External Coupling

- This coupling occurs when there is a dependency on an external library or system, it is best to reduce the number of places in the code where such dependencies exist, also creating a layer between your code and this external library may reduce this type of coupling
- the "**Facade pattern**" and "**Repository pattern**" may help

Control Coupling

- Two modules exhibit **control coupling** if one (“module A”) passes to the other (“module B”) a “parameter” that is intended to control the internal logic of the other
 - This will often be a value used in a test for a case statement, if-then statement, or while loop, in module B’s source code

Stamp Coupling

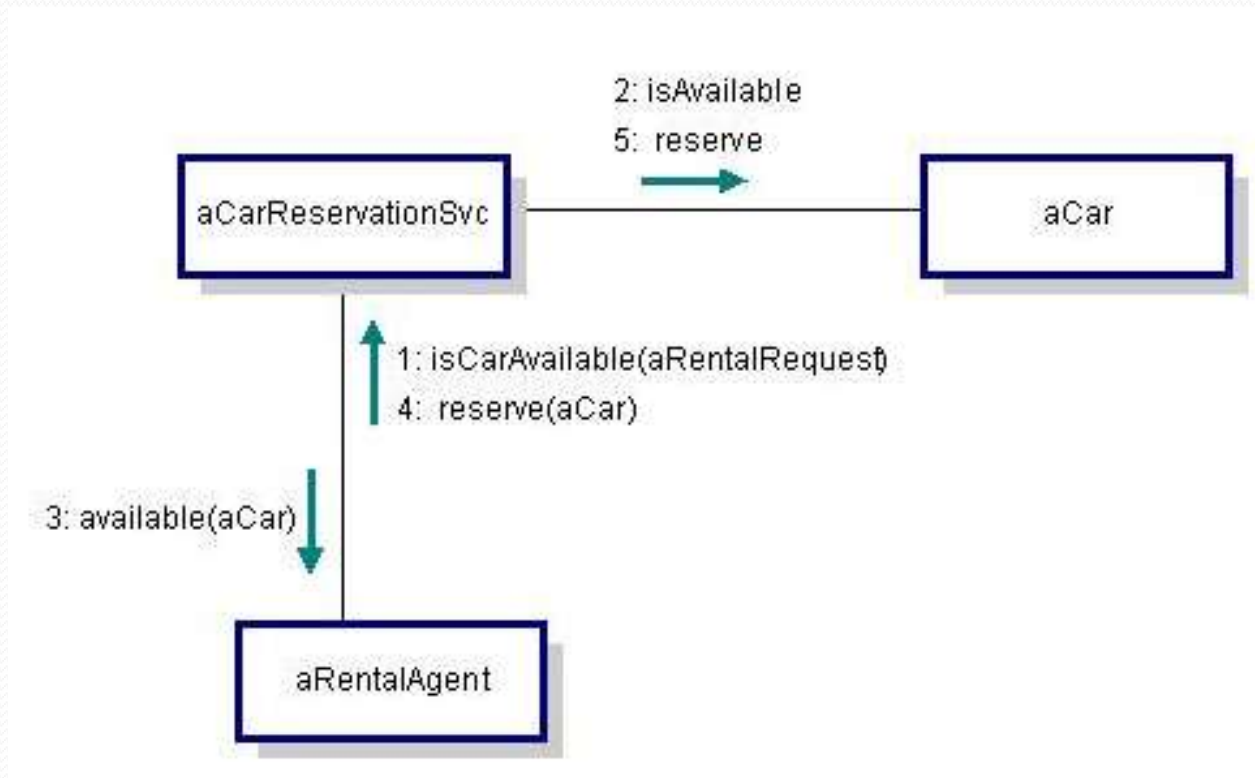
- Two modules (“A” and “B”) exhibit **stamp coupling** if one passes directly to the other “composite” piece of data – that is, a piece of data with meaningful internal structure – such as a record (or structure), array, or (pointer to) a list or tree

Data Coupling

- Two modules exhibit **data coupling** if one calls the other directly and they communicate using “too many parameters” – a simple list of inputs and outputs
 - The modules exhibit stamp coupling if “composite” data types are used for parameters as well
- Ideally, this **data coupling** is the usual type of interaction between modules that need to communicate at all
- You can spot such coupling when you start passing "**null**" to a function while calling it.
- Use data coupling when argument list is small (≤ 3)

Data coupling – Collaboration diagram

- messages sent between a Rental Agent object and a CarReservationService object to reserve a Car (the Car object is an argument passed in the messages shown in the object message diagram).



Definition of Cohesion

- Cohesion refers to the strength of a method as it relates to the routines within it.
- level of cohesion refers to the “functional independence” of a module.
- There are following categories of cohesion
 - Functional
 - Sequential
 - Communicational
 - Temporal
 - Procedural
 - Logical
 - Coincidental

Functional Cohesion

- A method has strong functional cohesion when it does just one thing
- Examples
 - Compute cosine of angle
 - Read transaction record
 - Determine customer mortgage repayment
 - Calculate net employee salary
 - Assign seat to airline customer

Functional Cohesion

- All of the procedures below have functional cohesion:
 - (1) Sort an array
 - (2) Search an array
 - (3) Find largest value in an array.
 - (4) Find the mean of the numbers in an array.
 - (5) Merge the data in one array with another.
 - (6) Print data entry format on the CRT screen.
 - (7) Input all values from the screen.
 - (8) Print report headings.
 - (9) Find square root
 - (10) Compute random number.
 - (11) Process Payroll (BOSS MODULE)
- Note: This procedure's single function is to call other procedures. Thus, its function is CONTROL of other procedures.
- **General rule:**
 - If you can say that the "procedure does this AND it does that" then the procedure may be doing more than one function.
 - If you can say that the "procedure does this OR it does that" then the procedure may be doing more than one function.

Sequential Cohesion

- A sequentially cohesive module is one whose elements are involved in activities such that output data from one activity serves as input data to the next
- Examples (Repaint a car)
 - Clean car body – remove old paint
 - Fill in holes in car
 - Sand car body
 - Apply primer

Sequential Cohesion

- EXAMPLES:
- Ex (1) Functions in one procedure:
 - a. Search array for a certain name.
 - b. Place the name in another array and sort the array.
 - c. Change the name from last-first to first-last.
 - d. Print first-last name.
- Ex (2) Functions in one procedure:
 - a. Read numbers from a disk file into an array.
 - b. Find the mean of an array.
 - c. Find deviation of each number from the mean.
 - d. Sort deviations from the highest to the lowest
 - e. Print the deviations in order from highest to the lowest.

Communicational Cohesion

- Several functions are combined into one procedure.
Each function in the process does something different to the same data.
- A method is said to have communicational cohesion when it access and modify the same part of a data structure.
- A communicational cohesive module is one whose elements contribute to activities that use the same input or output data
- Examples (Book)
 - Find title of book
 - Find price of book
 - Find publisher of book
 - Find author of book

Communicational Cohesion

- EXAMPLE:
- DATA STRUCTURE: TRANSACTION FILE
- PROCEDURE FUNCTIONS:
 - 1. Read transactions into array.
 - 2. Sort transactions
 - 3. Find the mean of the transactions.
 - 4. Print the transactions onto paper.
 - 5. Write the transaction to a CD-Writer

Procedural Cohesion

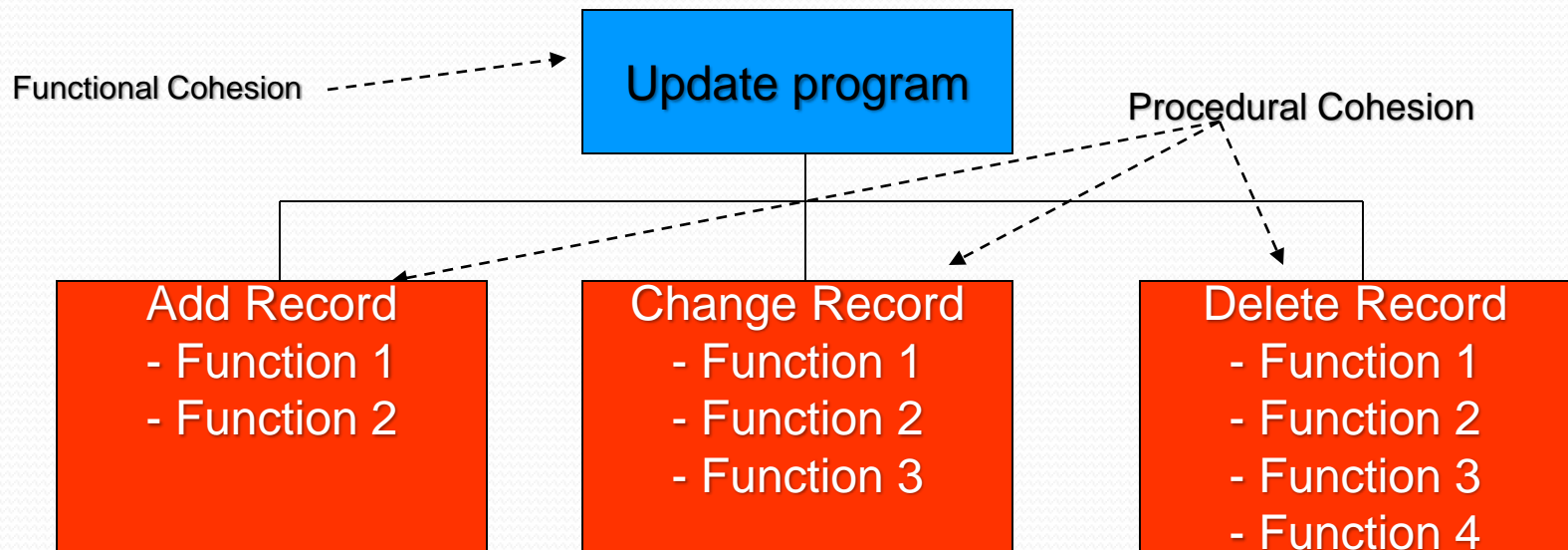
- A method is said to have procedural cohesion when all the routines within the method need to occur in a specified order and the routines don't share data
- A procedurally cohesive module is one whose elements are involved in different and possibly unrelated activities in which control flows from each activity to the next
 - Remember that in a sequentially cohesive module data, not control, flows from one activity to the next

Procedural Cohesion

- Example
 - Clean utensils from previous meal
 - Prepare chicken for roasting
 - Make phone call
 - Take shower
 - Chop vegetables
 - Set table

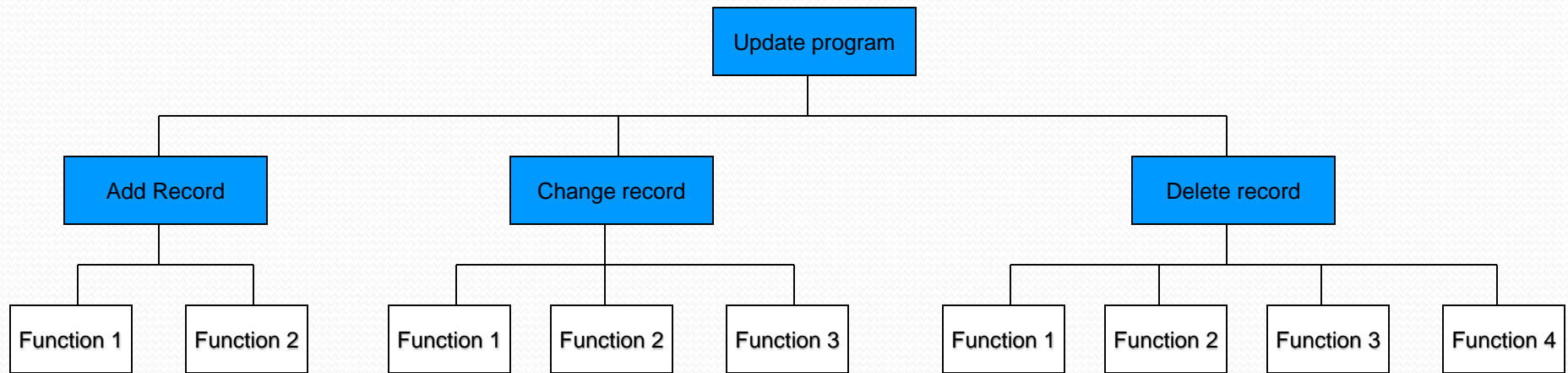
Procedural Cohesion

- EXAMPLE:
- (1) Consider a file update program in which additions, changes and deletions of records are made to the file, Each procedure consists of two or more single functions. A hierarchy chart might look as below. The **blue** procedure's single function is CONTROL. The **red** procedures are doing several functions...the entire "procedure".



Procedural Cohesion

- To eliminate the "procedural" cohesion in this example, the program could be re-designed as indicated in the hierarchy chart below.



Temporal Cohesion

- A method is said to have temporal cohesion when all the routines within the method need to occur at the same time, but not necessarily in order
- A temporally cohesive module is one whose elements are involved in activities that are related in time
- Example
 - Put out milk bottles
 - Put out cat
 - Turn off TV
 - Brush teeth

Logical Cohesion

- A method is said to have logical cohesion when the routines within the method are not related, don't share data, and the routine is selected by a flag either passed in as a parameter or, worse, existing outside the method.
- This type of procedure performs one or more logically related functions.

Logical Cohesion

- A logically cohesive module is one whose elements contribute to activities of the same general category in which the activity or activities to be executed are selected from outside the module
- Example
 - Go by car
 - Go by train
 - Go by boat
 - Go by plane

Logical Cohesion

- EXAMPLE:
- (1) A "print all" procedure might consist of:
 - a. Printing a local sales report.
 - b. Printing a regional sales report.
 - c. Printing a national sales report.
 - d. Printing a international sales report
- A "print all" procedure might consist of:
 - If the user, just wanted a national sales report, and not any of the others, a flag (a switch) would need to be passed to the procedure (using arguments and parameters) to tell the procedure which report to print.
- It is easy to spot this level of cohesion, because one or more flags are passed to the procedure to tell it which function to perform.
- **FLAGS ARE TO BE AVOIDED, BECAUSE THEY ADD COMPLEXITY TO THE CODE.**

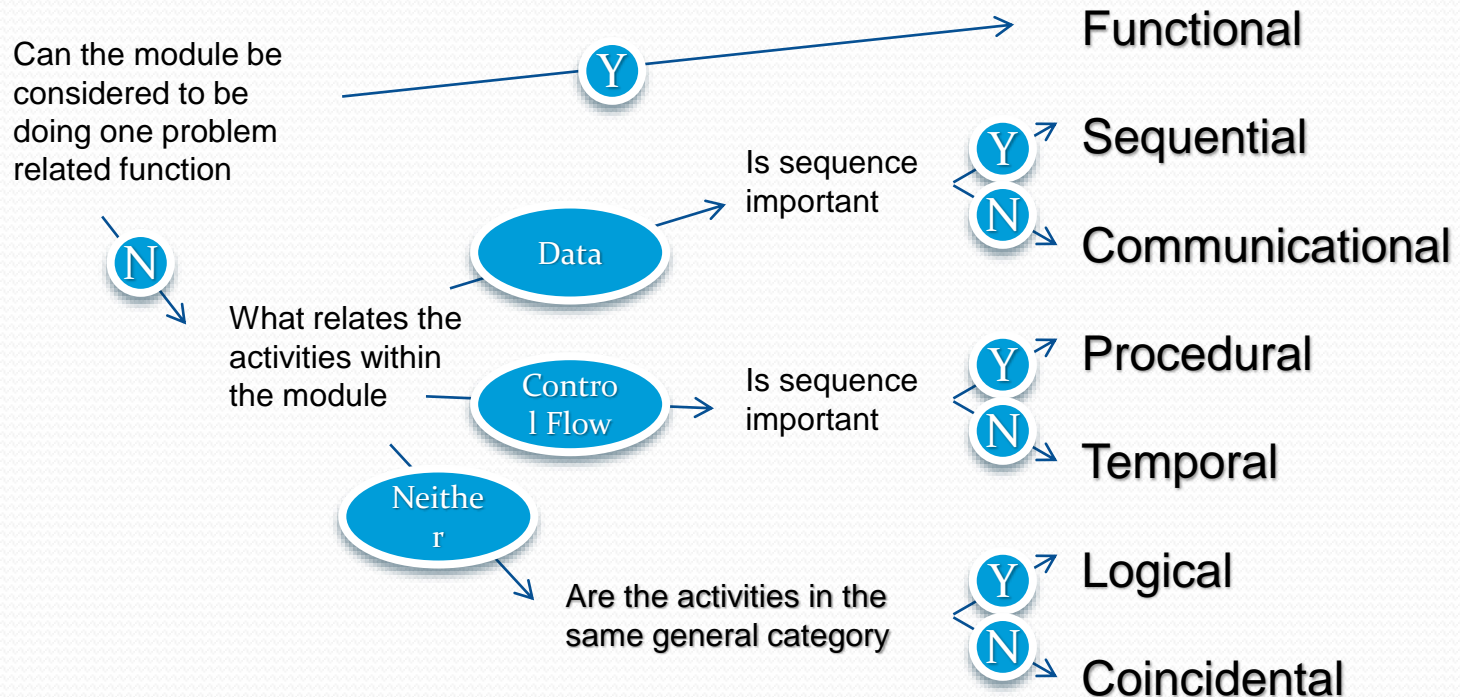
Coincidental Cohesion

- A method is said to have coincidental cohesion when the routines within the method are not related, and don't share data
 - This is a nice term which means the method does not have cohesion (or the cohesion is weak)

Coincidental Cohesion

- A coincidental cohesive module is one whose elements contribute to activities with no meaningful relationship to one another
- Example
 - Fix car
 - Bake cake
 - Walk dog
 - Fill out application form
 - Have a lunch
 - Get out of bed
 - Go to the movies

Decision Tree for Module Cohesion



Comparison of Level of Cohesion

Cohesion Level	Coupling	Cleanliness of Implementation	Modifiability	Understandability	Effect on Overall System Maintainability
Functional	Good	Good	Good	Good	Good
Sequential	Good	Good	Good	Good	Fairly Good
Communicational	Medium	Medium	Medium	Medium	Medium
Procedural	Variable	Poor	Variable	Variable	Bad
Temporal	Poor	Medium	Medium	Medium	Bad
Logical	Bad	Bad	Bad	Poor	Bad
Coincidental	Bad	Poor	Bad	Bad	Bad

Summary of Module Cohesion

- A module may exhibit any of seven levels of cohesion depending on how the activities within the module are related
- In sequence from best to worst, these seven levels are
- **Functional:** Elements contribute to a single, problem related activity
- **Sequential:** Activities within the module are connected in that the output from one serves as the input to another
- **Communicational:** Activities share the same input or output
- **Procedural:** Activities share the same procedural implementation
- **Temporal:** Activities can be carried out at the same time
- **Logical:** Activities appear to belong to the same general category
- **Coincidental:** Activities have no relationship to one another

Quiz 1 – GR1

```
• public int LargestNumber(List numberList)
• {
•     int largest = numberList[0];
•     int index = 0;
•     for(int counter = 0; counter < numberList.length; counter++) {
•         if(numberList[counter] > largest)
•             largest = numberList[counter];
•         if(largest >= 10000) {
•             largest = numberList[index];
•             continue;
•         }
•         if(counter > 99)
•             break;
•         index = counter;
•     }
•     log.debug("Largest number less than 10000 within top 100 is at " + index);
•     return numberList[index];
• }
```

Quiz 1 – GR2

```
• public int SmallestNumber(List numberList)
• {
•     int smallest = numberList[0];
•     int index = 0;
•     for(int counter = 0; counter < numberList.length; counter++) {
•         if(numberList[counter] < smallest)
•             smallest = numberList[counter];
•         if(smallest <= 0) {
•             smallest = numberList[index];
•             continue;
•         }
•         if(counter > 99)
•             break;
•         index = counter;
•     }
•     log.debug("Smallest number greater than 0 within top 100 is at " + index);
•     return numberList[index];
• }
```