# Department of Computing

**SE-210: Software Design and Architecture**

**Class:** BESE-9AB

# Lab 05: Cohesion and Coupling Types

# Instructor: Dr. Hasan Ali Khattak

# 

**Ahmed Hassan Ismail – BESE-9B – 237897**

# Lab 05: Cohesion and Coupling Types

### Introduction:

Students have learned the theoretical concepts of *cohesion and coupling types* in lectures. In this lab, students will learn how to identify and fix pieces of codes with low cohesion and tight coupling.

### Lab Objectives:

After the completion of this lab, students will be able to identify which parts of software have low cohesion and tight coupling. Students will also be able to fix such software to achieve a high cohesion and loose coupling.

### Helping Material:

Please consult lectures slides on LMS.

### Lab Tasks

### Task 1

We have learned seven different types of cohesion and seven different types of coupling as outlined below.

|  |  |
| --- | --- |
| Cohesion Types | Coupling Types |
| Functional | Content |
| Sequential | Common |
| Communicational | External |
| Procedural | Control |
| Temporal | Stamp |
| Logical | Data |
| Coincidental | Uncoupled |

Your task:

1. Choose *any three types of cohesion* and *any three types of coupling (excluding ‘uncoupled’)*. *For each of the chosen types*, you have to *give one coding example* where chosen cohesion/coupling type is present.
2. Given *a brief explanation of why the piece of code contains* the type of cohesion/coupling.
3. A *refactored version of the code*, to achieve high cohesion/loose coupling.
4. Given *a brief explanation of how the refactored version of the code is achieving* high cohesion/loose coupling.

**Important Note:**

1. You are not required to write a fully functional code. Only write enough code, which can make your point.
2. You have to give six examples in total (3 of cohesion and 3 of coupling).

**Example**

**Stamp Coupling**

**Violating Code:**

public class Emailer{

public void sendEmail (**Employee e**, String message) {...}

}

**Why the code contains stamp coupling**

The *sendEmail*() function requires only the name and email address of the employee. However, in the above code, the whole object of the *Employee* is passed. The receiver only requires part of the passed data. The receiver function has access to more information than what it needs. Thus the piece of code contain stamp coupling.

**Refactored Code**

public **interface Addressee** {

public abstract String getName();

public abstract String getEmail();

}

public class **Employee implements Addressee** {...}

public class Emailer{

public void sendEmail (**Addressee** e, String text) {...}

}

**Why the refactored code does not contain stamp coupling**

In the refactored code, we have introduced an interface, *Addressee*, which only contains the name and email address of the employee. The *Employee* class now implements *Addressee* which is passed to the sendEmail() function. The receiver now has access to the only data it requires to send email. Thus the piece of code does not contain any stamp coupling.

**Answer:**

|  |
| --- |
| Solution |
| Task 1  Cohesions:  Temporal Cohesions:  Violating Code:  class closeAllConnections(){  public void turnoff(){  os.closeDatabaseConncetion()  os.turnOffNetworkConnection()  os.saveProgramLastState()  os.closeProgram()  }}  Why the code contains temporal coupling  The code above has temporal coupling because after a program finishes there are 4 actions taken. The actions are not related to each other but are together because they must be executed in a particular time frame, in this case, after the program finishes.  Refactored Code:  class closeDatabaseConnection(){  public void closeDatabase(){  os.closeDatabaseConnection()}}  class turnOffNetwork(){  public void turnOffNetwork(){  os.closeAllWifiConnections()}}  class programState(){  public void saveProgram(){  os.saveProgramState()}  public void closeProgram(){  os.close()} }  Why the refactored code does not contain temporal cohesion?  The above code has separated modules for all the functions that are to be executed after the program ends. The ones that are connected are in the same module, for example programSave and programClose are in the same module. This finishes temporal cohesion completely since all the connected functions are in one module, else separated.    Coincidental Cohesion  Violating Code:  class bank () {  Account account  public void process () {  account.verifyCheck()  account.verifyMoney()  account.transferMoney()  account.depositMoney()  }}  Why the code contains coincidental cohesion?  The above code has coincidental coupling since all the functions which are miscellaneous, which are not known about where they belong, are put into the bank class. They are not at all related but just are about one domain, bank.  Refactored Code:  Global Account account  class transfer () {  public transferM(Account account){  account.transferMoney(int cash)}}  class verify () {  public checkV(Account account){  account.verifyCheck()}  public moneyV (Account account) {  account.verifyMoney()}}  Why the refactored code does not contain coincidental cohesion?  The above code has separate modules for all the connected functions. The functions are separated and have their own respective modules thus increasing cohesion. The responsibility of the modules is clear although previously It wasn’t. Since that one module had miscellaneous functions.  Logical Cohesion  Violating Code:  class doAllsystemIO () {  public void systemIO () {  setDefaultAudioOutput()  recieveAudioInput()  displayOutputToMonitor()}}  Why the code contains logical cohesion  The above code contains logical cohesion since the functions being performed in that one module are all related to the system input/Output, but this module will be very difficult to test and reuse. Since this module does not have one responsibility but has multiple responsibilities of the same domain which are logically related but not functionally.  Refactored Code:  class systemInput () {  public void setDefaultAudioInput(){}  public void recieveAudioInput(){}}  class systemOutput(){  public void displayOutputToMonitor(){}  public void setDefaultAudioOutput(){}}  Why the refactored code does not contain logical cohesion?  The above code has separated functions and only the ones that are functionally connected are in one module. There are separate modules for output and input. Unlike before which had all the system i/o in one module. Since connected functions are in one module, the code is easy to maintain and reuse.  Couplings:  Violating Code:  class driveCar(){  private int carSpeed=5;  public static int returnSpeed(){  return &carSpeed  }}  class startCar(){  public void startMainEngine(){  int i\* = driveCar.returnSpeed  i&=10  }}  Why the code contains content coupling?  The code above has content coupling since the class startCar() has access to the pointer carSpeed of the module driveCar(). Thus the private variable can be changed from the class startCar() thus increasing the coupling of this code.    Refactored Code:  class driveCar(){  private int carSpeed=5;  public static setSpeed(int speed){  return carSpeed=speed  }}  class startCar(){  public void startMainEngine(){  int i= driveCar.setSpeed()}}  Why the refactored code does not contain content coupling?  The above code has a proper flow now, the pointer is not being passed between the modules, there is a proper set method for modifying the car speed. Thus, coupling reduced.  Common Coupling  Violating Code:  Global int batteryPower = 60%  Class closeAllPrograms(){  if(batteryPower<15%){  closeAllPrograms()}}  class turnOnBatterySaver(){  If(batteryPower<20%){  TurnOnSaver()}}  Why the code contains common coupling?  The above code contains common coupling since there is one variable which is global being accessed by both the modules. Both modules are dependent on the global variable, they share the data. Due to this maintainability is difficult and reusability is too.  Refactored Code:  Class closeAllPrograms(){  private int batteryPower = 60%  public int getBatteryPower(){  return batteryPower}  if(batteryPower<15%){  closeAllPrograms()}}  class turnOnBatterySaver(){  If((closeAllPrograms).getBatteryPower<20%){  TurnOnSaver()}}  Why the refactored code does not contain common coupling?  The above code does not have common coupling since the global variable is in one class now and is accessed by the other class using the get method. Thus coupling reduced and the proper flow is introduced.  Stamp Coupling  Violating Code:  class salary(){  public void changeSalary(Employee e, String message){}}  Why the code contains stamp coupling?  The above code contains stamp coupling since the salary has to be changed but the method is given the whole object data of the employee, which is irrelevant, since the method requires the salary of the employee.  Refactored Code:  class salary(){  Employee e  public void changeSalary(e.salary,e.name, String message){}}  Why the refactored code does not contain stamp coupling?  The above code does not contain stamp coupling since only the required piece of information is being passed to the method, which is the salary, rather than the whole employee information. |

### Deliverables

Compile a single word document by filling in the solution part and submit this Word file on LMS. This lab grading policy is as follows: The lab is graded between 0 to 10 marks. The submitted solution can get a maximum of 5 marks. At the end of each lab or in the next lab, there will be a viva related to the tasks. The viva has a weightage of 5 marks. Insert the solution/answer in this document. You must show the implementation of the tasks in the designing tool, along with your completed Word document to get your work graded. You must also submit this Word document on the LMS. In case of any problems with submissions on LMS, submit your Lab assignments by emailing it to **Sundas Dawood** <sundas.dawood@seecs.edu.pk>