## 

## 

Truc Huynh

ACS 560: Software Engineering

HW2

## Functional Cohesion

### DEFINITION:

Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module (e.g., [Lexical analysis](https://en.wikipedia.org/wiki/Lexical_analysis) of an XML string) (1).

### EXAMPLE CODE:

Example code using pseudocode

*/\**

*Groups: The function definitions*

*Parts: The terms on each function*

*\*/*

Module A {

*/\**

*Implementation of arithmetic operations*

*This module is said to have functional cohesion because*

*there is an intention to group simple arithmetic operations*

*on it.*

*\*/*

a(x, y) = x + y

b(x, y) = x \* y

}

Module B {

*/\**

*Module B: Implements r(x) = 5x + 3*

*This module can be said to have atomic cohesion. The whole*

*system (with Modules A and B as parts) can also be said to have functional*

*cohesion, because its parts both have specific separate purposes.*

*\*/*

r(x) = [Module A].a([Module A].b(5, x), 3)

}

### PROS/CONS:

Pro: Reducing the complexity of the module. All the functional cohesion is strongly related. Therefore, it helps build reliable and reusable software. Have fewer modules compare to other cohesion (it is a single-problem-related task).

Cons: Hard to implement (need more time, good design, and skill).

## Communicational Cohesion

### DEFINITION:

Communicational or informational cohesion is when parts of a module are grouped because they operate on the same data (e.g., a module that operates on the same record of information) (1).

### EXAMPLE CODE:

Example code using R

*/\**

*All Module is called from Client-side*

*GetConnection establish the connection between the*

*\*/*

getConnection <- function(){

connection = dbConnect(MySQL(),

user = 'root',

password = 'mypassword',

dbname = 'mydata',

host = 'localhost');

return(connection);

};

*/\*Create the connection\*/*

Connection<-getConnection()

*/\**

*ModuleGetUserTransactions: get the transaction\_id, transaction\_store, amount from the user\_transaction\_user\_id\_ user\_id \*/*

ModuleGetUserTransactions(connection,user\_id){

*/\**

*Query to get the transaction\_id, transaction\_store, amount from the user\_transaction\_user\_id\_ user\_id*

*\*/*

query<-sprintf("SELECT transaction\_id, transaction\_store, amount FROM mydata.user\_transaction\_user\_id\_%s;",user\_id)

*/\*Send the query to the database and get result \*/*

rs <- dbSendQuery(connection, query)

GetUserTransactions<-dbFetch(rs)

dbClearResult(rs) */\*Clear the query\*/*

dbDisconnect(connection) */\*Disconnect the database\*/*

return GetUserTransactions

}

*/\**

*ModuleGetUserStoreTransactions: get the transaction\_id, transaction\_category, store\_name, store\_location from the user\_transaction\_user\_id\_ user\_id \*/*

ModuleGetUserStoreTransactions(connection,user\_id){

*/\**

*Query to get the transaction\_id, transaction\_store, amount from the user\_transaction\_user\_id\_ user\_id*

*\*/*

query<-sprintf("SELECT transaction\_id, transaction\_category, store\_name, store\_location FROM mydata.user\_transaction\_user\_id\_%s;",user\_id)

*/\*Send the query to the database and get result \*/*

rs <- dbSendQuery(connection, query)

GetUserStoreTransactions<-dbFetch(rs)

dbClearResult(rs) */\*Clear the query\*/*

dbDisconnect(connection) */\*Disconnect the database\*/*

return GetUserStoreTransactions

}

*/\**

*The 2 functions access to the same table in the same database by calling getConnection() and pass it as a parameter.*

*\*/*

### 3. PROS/CONS:

Cons: Security issues: Since multiple modules work on the same data there is complicated to implement data security. Data modification (or even injection) may execute without noticing. This can make the database admin job harder.

Pros: Centralizing data make communication easier to make changes across all the applications.

## Logical Cohesion

### DEFINITION:

Logical cohesion is when parts of a module are grouped because they are logically categorized to do the same thing even though they are different by nature (e.g., grouping all mouse and keyboard input handling routines). (1)

### EXAMPLE CODE:

Example code use pseudocode:

*/\**

*Class Human has 3 function jump, read and walk*

*\*/*

Class Human {

*@param jump*

*@param move*

*@param read*

*/\**

*function jump*

*\*/*

jump <- function(times){

return jump\*times

};

*/\**

*function walk*

*\*/*

walk <- function(steps){

return move\*steps

};

*/\**

*function walk*

*\*/*

read <- function(minutes){

return read\*minutes

};

};

*/\**

*Class Dog has 2 function jump and walk*

*\*/*

Class Dogs {

*@param jump*

*@param move*

*/\**

*function jump*

*\*/*

jump <- function(times){

return jump\*times

};

*/\**

*function walk*

*\*/*

walk <- function(steps){

return move\*steps

};

};

### 3. PROS/CONS:

**Pro:** Put all attributes in one object, easier to deploy the object (Humans can read, walk, jump/ Dogs can walk and jump).

**Con**: Maintainability issues: May repeat the function when all the objects perform the same action (jump or walk in this case). Change in one function may not be in another. Have to go to each class to change. Ex: I want to add a feature to the walk function, then I need to change it in both classes. Solution: Can design a parent class Mammals(contain walk and jump function) then the Human and Dogs Class can be the child of Mammals Class.

## Content Coupling

### DEFINITION:

Content coupling is said to occur when one module uses the code of another module, for instance, a branch. This violates information hiding (basic design concept).

Content coupling 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. (3)

### EXAMPLE CODE:

Example code using java(4)

*/\**

*Class Line contains the Point Object private*

*\*/*

public class Line

{

private Point start, end;

...

public Point getStart() { return start; }

public Point getEnd() { return end; }

}

*/\**

*The slant() access to the end inside the Line and change it.*

*\*/*

public class Arch

{

private Line baseline;

...

void slant(int newY)

{

Point theEnd = baseline.getEnd();

theEnd.setLocation(theEnd.getX(),newY);

}

}

### PROS/CONS:

**Cons**: Content coupling is a design fault since it violates the Object-Oriented Design concept (Information hiding)

## Common Coupling

### DEFINITION:

Common coupling happens when multiple modules have access to the same global data. However, changes are made are unforeseen and error is created.

### EXAMPLE CODE:

Example code using java

*/\**

*Pi is a public variable and set to private*

*\*/*

Public double pi=3.14

*/\**

*Class Line return a pi =0, and accident set pi =0*

*\*/*

public class Line

{

private Point start, end;

...

public Point getStart() { return start; }

public Point getEnd() { return end; }

public getPi(){return pi=0;}

}

*/\**

*Class circle used pi to calculate the square*

*\*/*

public class Circle

{

private Line radius;

...

Double square (){

return radius\*pi;

}

}

### PROS/CONS:

**Cons**: Make sure to control the changes of data. Limited changing, and make global variable const as much as possible. Otherwise, can create Logic problems and hard to fix (unpredicted changes in a variable).

**Pro**: global variable is sometimes needed and convenient. It can help in some situations where designers need access to the same data.

## Stamp Coupling

### DEFINITION:

Stamp Coupling is also called data-structured coupling. When a module passes a non-global data structure or entire structure to another module they said to be stamp coupled

### EXAMPLE CODE:

*/\**

*For example, User A has 20 functions and 10 variable*

*\*/*

Public class user

{

Private int id;

Private int address;

Private int name;

Private int SSN;

. . .

Int GetUserID(){return id}

}

*/\**

*Pass the whole user into transaction Object to retrieve the ID*

*\*/*

public class userTransaction(user UserA)

{

userID = UserA.GetUserID()

...

*/\**

*Use the User ID to search for transaction data or do something*

*\*/*

}

### 3. PROS/CONS:

**Cons**: This will make the program run slow since put the whole object into a new object just to get part of the object

**Pros**: This will ensure data is secure, but we have many ways to work around it.

## Mutation Testing

### DEFINITION:

Mutation Testing means modifying a program in a small way. Each mutated version of the program is called a mutant. Test detect and reject mutants based on the difference in the behavior compared to the original version. One program is modified, a sequence of the unit test are executed against the mutant. Acceptable unit-test must detect the errors, otherwise, it will be rewritten. This is called killing the mutant.

### EXAMPLE CODE:

Code example retrieved from (5)

*/\**

*Consider the following C++ code fragment*

*\*/*

if (a && b) {

c = 1;

} else {

c = 0;

}

*/\**

*The condition mutation operator would replace && with || and produce the following mutant:*

*\*/*

if (a || b) {

c = 1;

} else {

c = 0;

}

*/\* Now, for the test to kill this mutant, the following three conditions should be met:*

1. *A test must reach the mutated statement.*
2. *Test input data should infect the program state by causing different program states for the mutant and the original program. For example, a test with a = 1 and b = 0 would do this.*
3. *The incorrect program state (the value of 'c') must propagate to the program's output and be checked by the test.*

*\*/*

### 3. PROS/CONS:

Pros:

* Mutation testing brings a good level of error detection to software developer
* This method uncovers ambiguities in the source code and can detect all the faults in the program.

Cons:

* Extremely costly and time-consuming since many programs need to create
* Can’t be done with automation tools.

## Regression Testing

### DEFINITION:

Regression Testing is to run all the functional and non-functional testing of the software after changes are made.

### EXAMPLE CODE:

*/\**

*Consider the following Code for User Object*

*\*/*

Class User {

Int id,

String Name,

. . .

Int getID(){return id};

String getName(){return Name};

}

*/\**

*After Do all the test for the app*

*And the app works good for a while,*

*The user wants to add the registration() function to the User Class.*

*Now the app looks like this*

*\*/*

Class User {

Int id,

String Name,

. . .

Int getID(){return id};

String getName(){return Name};

Bool Registration(){

*/\*Do some here\*/* }

}

*/\**

*We now need to apply regression methods such as Retest All, Regression test, Test Case Prioritization, or Hybrid … to Test the software again after the Registration() function is created*

*\*/*

### 3. PROS/CONS:

## Iterator pattern

### DEFINITION:

Iterator pattern is a popular design pattern method that is widely used in Java and .Net environments. The idea is to access the elements of a collection object (in a sequential manner) without knowing the underlayment. (6)

### EXAMPLE CODE:

Code example retrieved from (6)

STEP 1: Create Interfaces

*/\**

*Consider the following Java Code for class Iterator.Java*

*\*/*

public interface Iterator {

public boolean hasNext();

public Object next();

}

*/\**

*Consider the following Java Code for class Container.Java*

*\*/*

public interface Container {

public Iterator getIterator();

}

STEP 2: Create a concrete class implementing the Container Interface

*/\**

*Inner Class Iterator implements the Iterator interface*

*\*/*

public class NameRepository implements Container {

public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

*/\**

*getIterator () override the method in Container.java*

*\*/*

@Override

public Iterator getIterator() {

return new NameIterator();

}

private class NameIterator implements Iterator {

int index;

*/\**

*hasNext () override the method in Iterator.Java*

*\*/*

@Override

public boolean hasNext() {

if(index < names.length){

return true;

}

return false;

}

*/\**

*Next function override the method in Iterator.Java*

*\*/*

@Override

public Object next() {

if(this.hasNext()){

return names[index++];

}

return null;

}

}

}

STEP 3:

*/\**

*hasNext next override the method in Iterator.Java*

*Use the NameRepository to get iterator and print names.*

*IteratorPatternDemo.java*

*\*/*

public class IteratorPatternDemo {

public static void main(String[] args) {

NameRepository namesRepository = new NameRepository();

for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){

String name = (String)iter.next();

System.out.println("Name : " + name);

}

}

*}*

*/\**

*OUTPUT*

*\*/*

Name : Robert

Name : John

Name : Julie

Name : Lora

### PROS/CONS:

Pro: Hide Implementation. Easier to implement (adding new interface to existing code without changing a lot). Can iterate to the same collection in parallel since each iterator contain its own state (multithreading programming).

Cons: Take longer time to develop, especially for large application. Required certain knowledge and skill level. For simple application, may not be necessary (time and resource consuming).

## Decorator pattern

### DEFINITION:

Decorator pattern is a design pattern that allow developers to add more functionality to the existing object without altering the object structure.

### CASE STUDY:

In the below diagram:

* Shape: public **interface** has *draw()* function
* Rectangle: public **class** that implement **Shape interface**
* Circle: public **class** that implement **Shape interface**
* ShapeDecorator: is a public abstract class implement **Shape interface:**

public abstract class ShapeDecorator implements Shape {

protected Shape decoratedShape;

public ShapeDecorator(Shape decoratedShape){

this.decoratedShape = decoratedShape;

}

// Instead of override the methods.

// The abstract class use the Shape as an parameter

// That can be combine with other function when extends the

// class.

public void draw(){

decoratedShape.draw();

}

}

* RedShapeDecorator: is a public class that extend **ShapeDecorator class:**

public class RedShapeDecorator extends ShapeDecorator {

public RedShapeDecorator(Shape decoratedShape) {

super(decoratedShape);

}

// This override the methods in the parent class and add more

// functionality to the child class without modify the code

// in the original Shape interface and the parent

// (Shape Decorator)

@Override

public void draw() {

decoratedShape.draw();

setRedBorder(decoratedShape);

}

// This override the methods in the parent class and add more

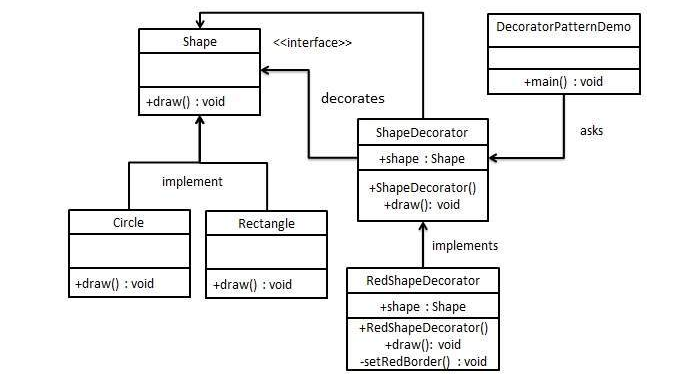
//

private void setRedBorder(Shape decoratedShape){

System.out.println("Border Color: Red");

}

}



public class DecoratorPatternDemo {

public static void main(String[] args) {

Shape circle = new Circle();

// Pass a circle object into the RedShapeDecorator

Shape redCircle = new RedShapeDecorator(new Circle());

// Pass a rectangle object into the RedShapeDecorator

Shape redRectangle = new RedShapeDecorator(new Rectangle());

// This circle is a normal Circle, output:

/\*

Circle with normal border

Shape: Circle

\*/

System.out.println("Circle with normal border");

circle.draw();

// This circle is a Circle with red border, output:

/\*

Circle of Red border

Shape: Circle

Border: Red

\*/

System.out.println("\nCircle of red border");

redCircle.draw();

// This circle is a Circle with red border, output:

/\*

Rectangle of Red border

Shape: Rectangle

Border: Red

\*/

System.out.println("\nRectangle of red border");

redRectangle.draw();

}

}

Summary: Functionality was add without actual modifying the code.

### 3. PROS/CONS:

Pro: Adding more functions to the existing code. This reduce the bug and mistake while we modifying the original code.

Reference:

 Stevens, Wayne P.; Myers, Glenford J.; [Constantine, Larry LeRoy](https://en.wikipedia.org/wiki/Larry_LeRoy_Constantine) (June 1974). "Structured design". IBM Systems Journal. 13 (2): 115–139. [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[10.1147/sj.132.0115](https://doi.org/10.1147%2Fsj.132.0115).(2).

Techopedia (2019) Mutation Testing. Retrieved from [What is Mutation Testing? - Definition from Techopedia](https://www.techopedia.com/definition/20905/mutation-testing#:~:text=Mutation%20testing%20is%20a%20method%20of%20software%20testing,code%20are%20designed%20to%20imitate%20common%20programming%20errors.) (4).

Tutorial Point(n.d.) *Design Patterns - Iterator Pattern*. Retrieved from <https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm> (6)

Tutorial Point(n.d.) *Design Patterns - Decorator Pattern*. Retrieved from [https://www.tutorialspoint.com/design\_pattern/decorator\_ pattern.htm](https://www.tutorialspoint.com/design_pattern/decorator_%20pattern.htm) (7).

Wikipedia (n.d.) *Coupling (computer programming).*  Retrieved from [Coupling (computer programming) - Wikipedia](https://en.wikipedia.org/wiki/Coupling_%28computer_programming%29) (3).

Wikipedia (n.d.) *Mutation Testing (computer programming).*  Retrieved from [Mutation testing - Wikipedia](https://en.wikipedia.org/wiki/Mutation_testing) (5).

Yourdon, Edward; Constantine, Larry LeRoy (1979) [1975]. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press. Bibcode:1979sdfd.book.....Y. ISBN 978-0-13-854471-3. ISBN 0-13-854471-9.(1)