SIT105 - Critical Thinking and Problem Solving for IT
Class 09

Communication Between Modules!

Part 1 – Cohesion

Part 2 – Coupling

A/Prof. Xiao Liu

xiao.liu@deakin.edu.au

Quote

Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defence against complexity.

Quote by: David Gelernter
Professor of computer science at Yale University

Developing Good Modules

- A module is defined as a section of an algorithm that is dedicated to the performance of a **single function**.
- It should contain a single **entry** and a single **exit**
- The **name** of the chosen module should accurately describe its function
- When we are making modules we run into common problems, such as:
  - How **big** should it be?
  - Is it too **small**?
  - Are all the instructions **related** to the module?
  - Are the links between modules too **complex**?
- We want to address this today with **cohesion and coupling**

Communication between modules!

- When it comes to designing algorithms, it is necessary to consider not only:
  - *The division of the problem into modules*
  - *But also the flow of information between modules*
- It is important to have fewer and simpler communication between modules
  - *The key is that it is easier to understand and maintain one module without reference to other modules.*
- When we talk about the flow of information, this is called ***intermodule communication***
  - *Which is tackled by the scope of variables or passing of parameters.*

Communication between modules!

A few quick points to keep in mind moving forward!

- **Global data:** This is a variable which can be used by all the modules in your algorithm.
- **Local data:** This variable is defined within a sub-module, they are limited in scope to that sub-module.
- **Passing parameters:** at some stage we will have to communicate between sub-modules, one way we can do this is with passing parameters (this is simply data transferred from a calling module to its subordinate module).

***Side effects:** problems can occur with cross communication, for example a sub-module trying to alter a global variable inside of a module. Which is what we want to try and avoid through today's topic.*

Quick look: parameters?

This is when we are calling a module with parameters

```
MyModule1():
MyModule2(a):
MyModule3(a, b):
MyModule4(a, b, c):
MyModule5(a, b, c, d):
```

No parameters

1 parameter

Multiple parameters

When we call the MyModule we have to call it into action with the same number of parameters, it is defined with.

E.g. of the module

```
MyModule2(a)
        SET x = 1
        z = x + a
        return(z)
END
```

Part 1 Content

1. Module Cohesion
2. Kinds of Cohesion

Cohesion?

- When we are talking about module cohesion, this relates to the internal strength of the module!
- It actually indicates how closely the elements or statements of a module are associated with each other.
- The more closely linked or related the statements are within a module, the higher the cohesion.
- High cohesion is good!

Cohesion – What is it ?

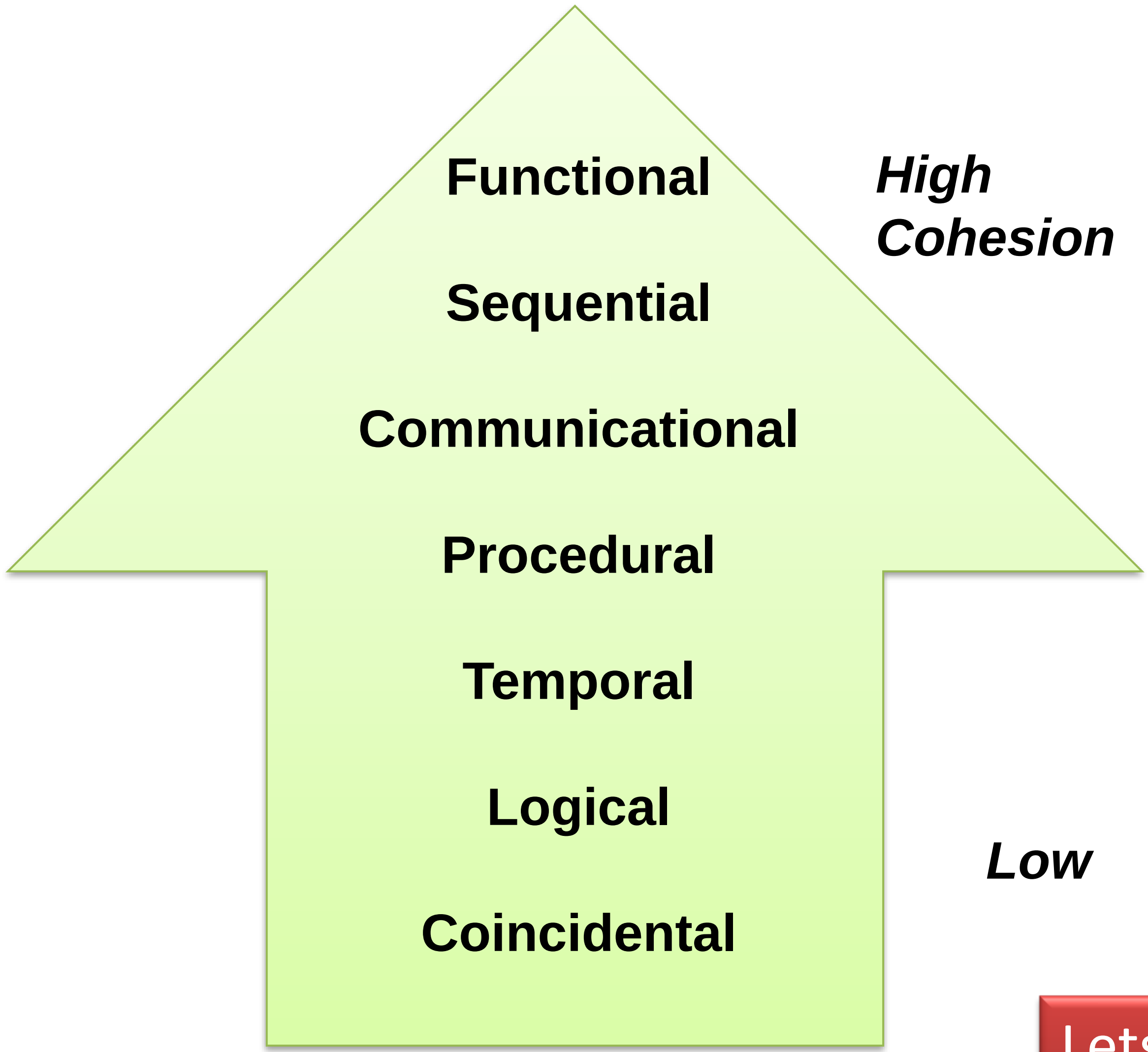
- **Cohesion**: the elements grouped together inside a module should have strong relationships and should be together.
- It is essentially the **internal glue** with which a module is constructed.

Definition:

- The degree to which all elements of a module are directed towards a single task.
- The degree to which all elements directed towards a task are contained in a single module.
- The degree to which all responsibilities of a single module are related.

- Goal: design your modules with high cohesion!

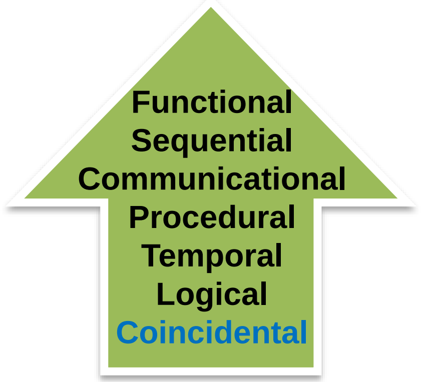
Module Cohesion



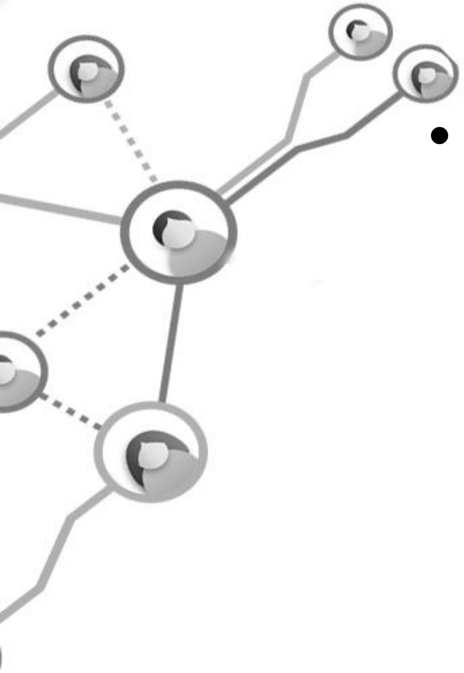
Lets discuss these

1. Coincidental Cohesion (low cohesion)

- No obvious relationship between the elements of the module
- Parts of the module are only related by their location in the source code.
- Worst form and to be avoided!



2. Logical Cohesion



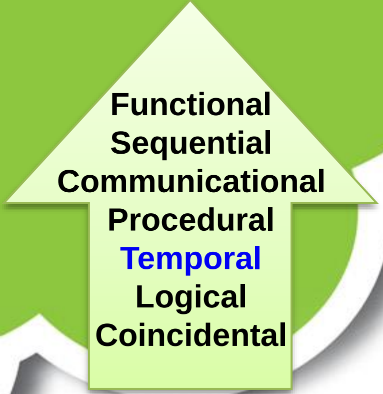
- Elements are grouped together because they do <u>the same kind of thing!</u>
- Elements of a module are related **logically** and **not functionally**.
- <u>E.g.</u>
  - A module reads inputs from tape, disk, and network.
  - All the code for these functions are in the same module.
  - Operations are related, but the functions are significantly different.

Functional
Sequential
Communicational
Procedural
Temporal
Logical
Coincidental

3. Temporal Cohesion

- Elements in a module related only because they are executed at the same time such as *start of program, end of program, and other events.*
  - Elements are grouped by when they are processed.
- These tasks may not be interrelated in any way!
  - May have a module, with several things to do at the start of the program but are not related.

Functional
Sequential
Communicational
Procedural
Temporal
Logical
Coincidental



3. Temporal Cohesion (example)

1. An <u>initialisation</u> module may contain all of the following:
   1. initialising variables (e.g. set X = 0)
   2. creating data structures and/or objects (data structure can be like say a student record)
      - An object can be a variable, a data structure, or a function.
   3. initialising data structures and/or objects (putting data into these objects)
   4. opening files (getting access to data files)
   5. connecting to other local processes (operating system manages the processes)
   6. connecting to other remote processes (processes on another machine)

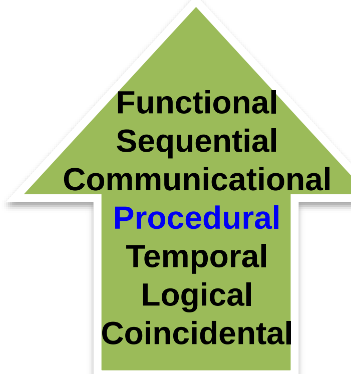
Lots of different activities occur, all at initialization time!
Are they all related to the same task though?

4. Procedural Cohesion

- Elements of a module are related only to ensure a particular order of execution.
  - This first, then that and then something else.
- Actions can still be weakly connected though.

E.g. In your module your first 10 lines do something, your next 20 do something else.

- You move the 20 lines of code to the very top so it occurs first instead of the 10

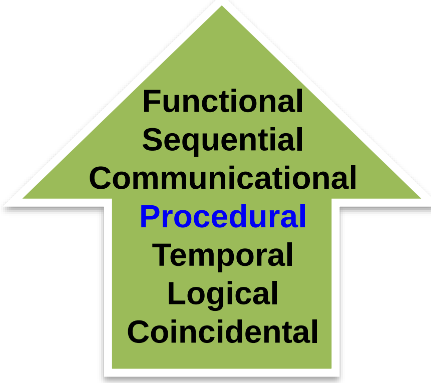


4. Procedural Cohesion (example)

3. A module that sends email performs the following:
   - Get the sender's address
   - Get the receiver's address
   - Get the subject text
   - Get the body text
   - Get the attachments
   - Etc.

When you hit the send button that's the important part.

The order in which we gather this data is not important

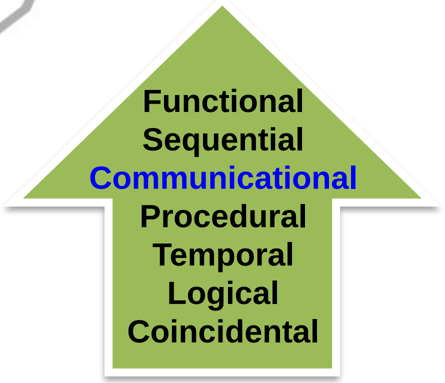


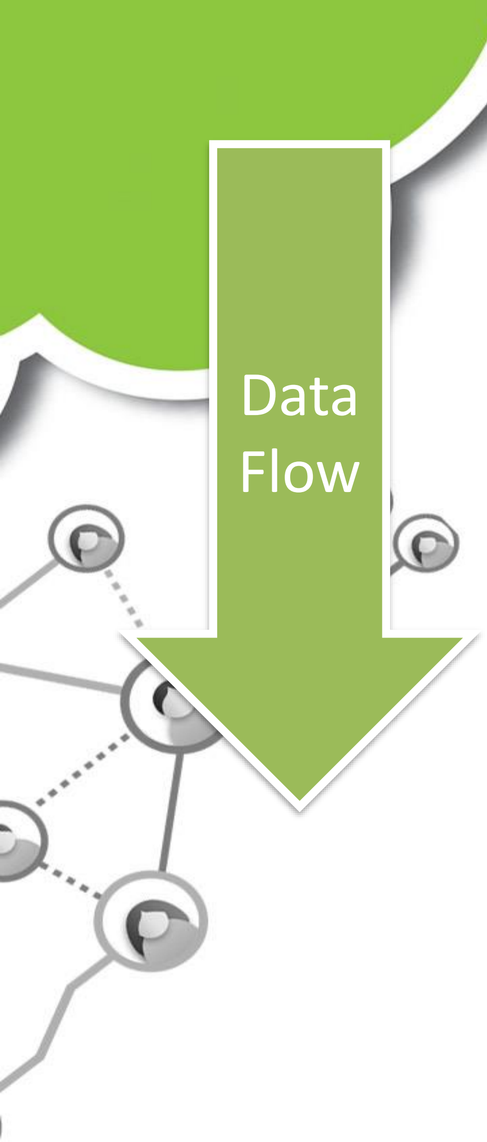
5. Communicational Cohesion

- Functions performed on the same data or to produce the same data.
- Many chunks of code that can access or shared among the same data.

Examples

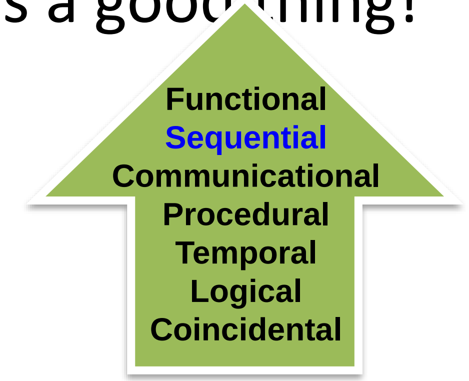
1. Excel formula, functions and macros operate on data within the workbook (a set of worksheets)
2. DBMS (data base management system) functions operate on the data within the database



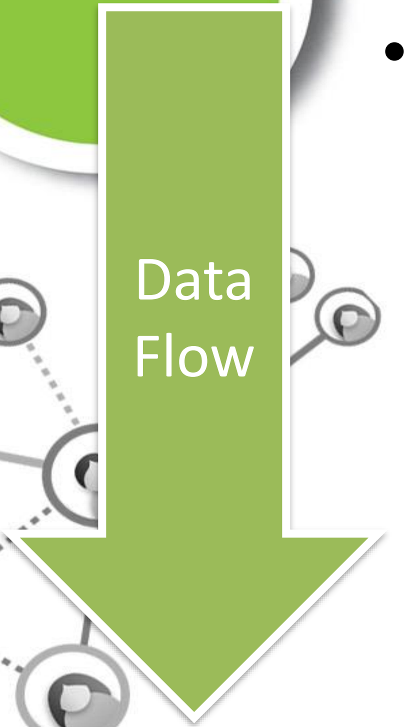


6. Sequential Cohesion

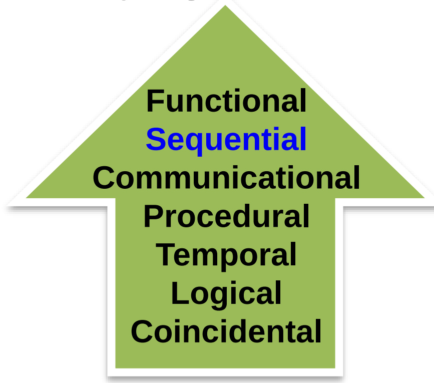
- This cohesion occurs when a module contains elements that depend on the processing of previous elements.
- **Much like an assembly line** – series of sequential steps that perform transformations of data.
- The output of **one part** is the **input to another**.
  - What happens before is needed for what happens afterwards.
  - Order is important !
- *Data flow* between your modules and is a good thing!
- Do this more often!



6. Sequential Cohesion (examples)

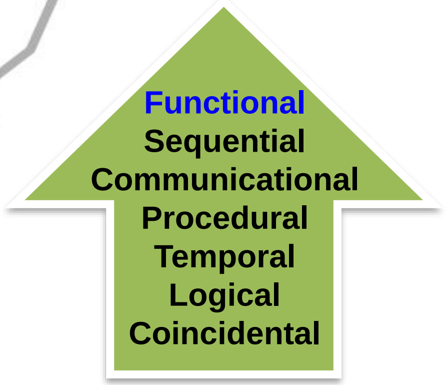


- **Updating a record of a file contains:**
  1. Open file (e.g. MS Excel file)
  2. Enter new values to be stored
  3. Validate or check the new values (are they OK? E.g. are you trying to put a number into text field)
  4. Update record values (once OK then change what you need too)
  5. Write record to file (updated record in the file)



7. Functional Cohesion (high cohesion)

- All processing elements contribute to the performance of a single task!
  - Every essential processing element is contained in the module.
- IDEAL!
- What is a functionally cohesive component?
  - One that not only **performs the task** for which it was designed but
  - It performs **only that function** and nothing else.



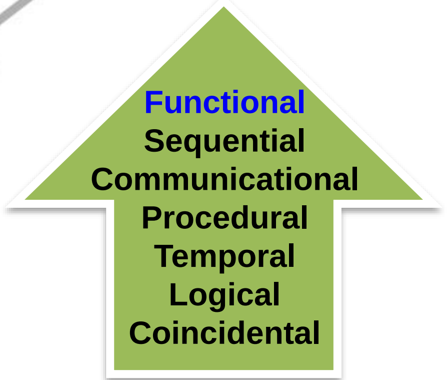
NOTE: in the following examples,
the processing does not include, say,
printing values because this is irrelevant to the task

7. Functional Cohesion (example 1)

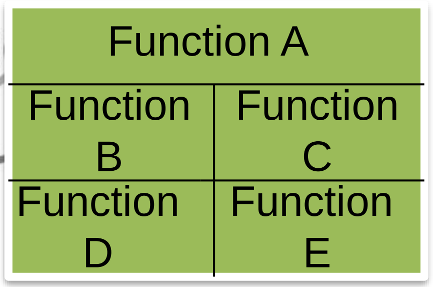
Calculate the surface area of a box

Inputs	Processing	Outputs
box dimensions	calculate area	surface area

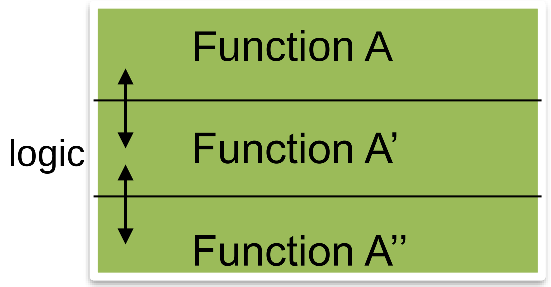
```
surfaceAreaBox(width, height, depth)
    top = width * depth
    front = width * height
    side = depth * height

    surfaceArea = 2 * (top + front + side)

    return(surfaceArea)
END
```



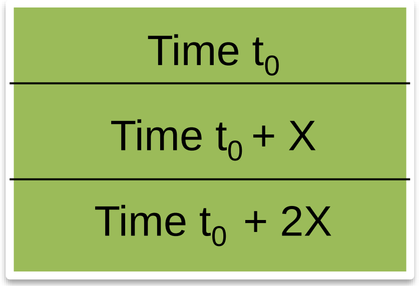
Summary: Cohesion



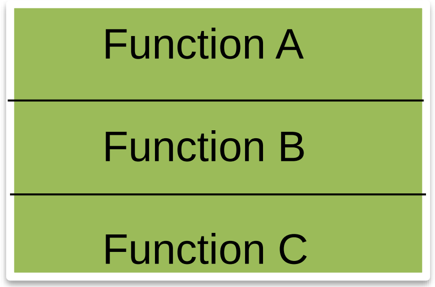
Coincidental
Parts unrelated /
no relationship



Logical
Similar functions



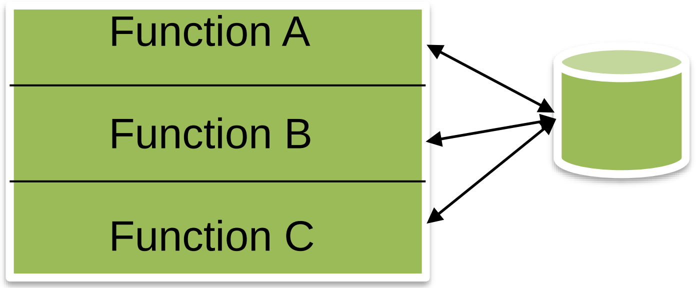
Temporal
Executed at the same time



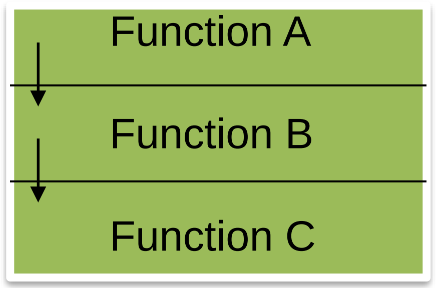
Procedural
Related by order of functions

Summary: Cohesion (Cont.)

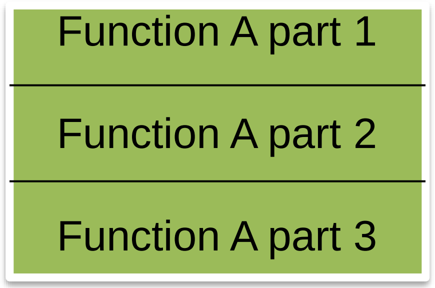
Sequential and Functional are best !



Communicational
Access same data



Sequential
Output of one is input to another

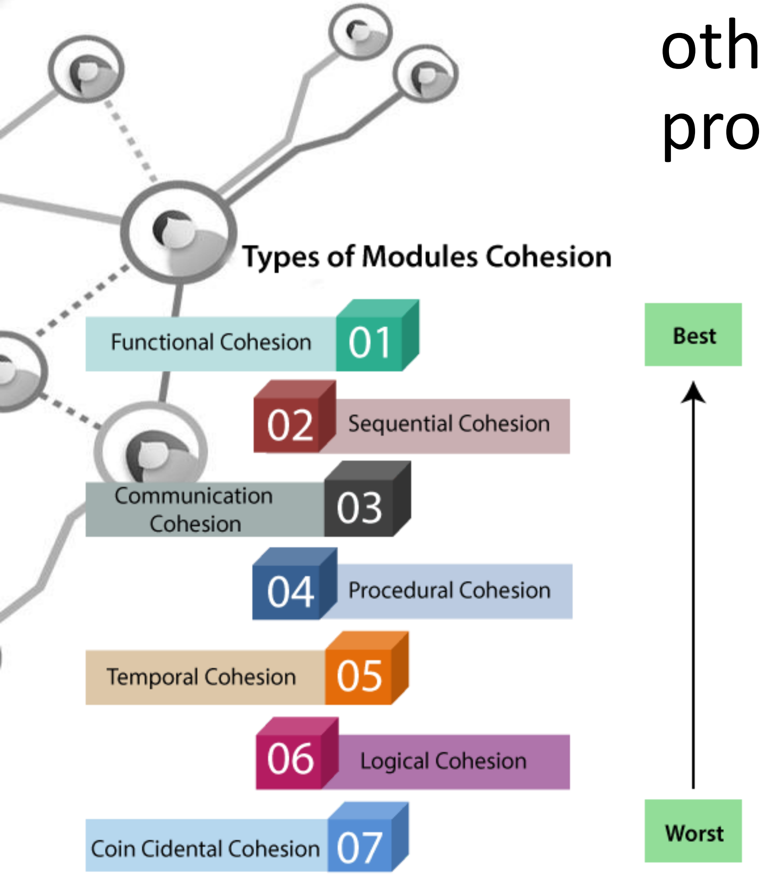


Functional
Sequential with complete, related functions

Part 1 Summary

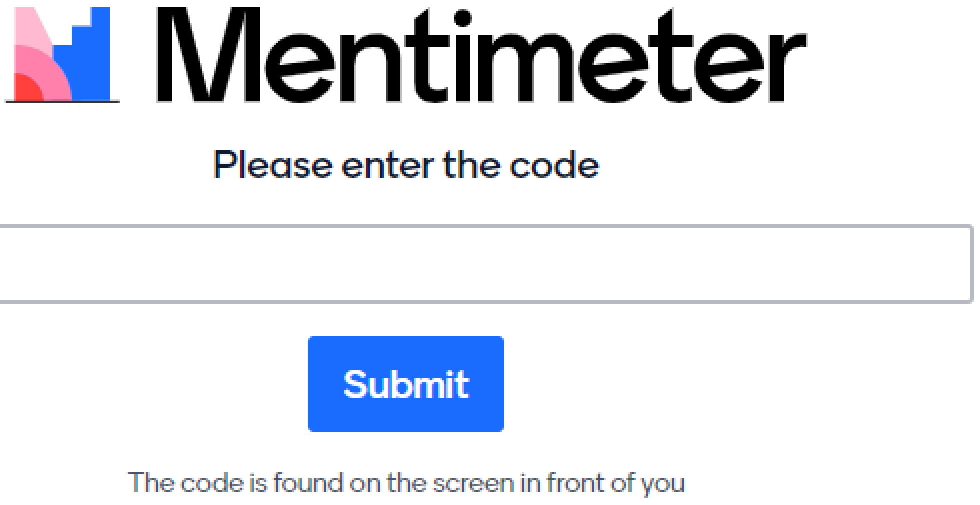
- <u>Coincidental cohesion should be avoided.</u>
- The contents of a module should be based on the other 6 types of cohesion; each has a place in programming.

2. Logical
3. Temporal
4. Procedural
5. Communicational
6. Sequential
7. **Functional (IDEAL)**



Question Time!

- What happens when you type a URL in browser?


Mentimeter

Please enter the code

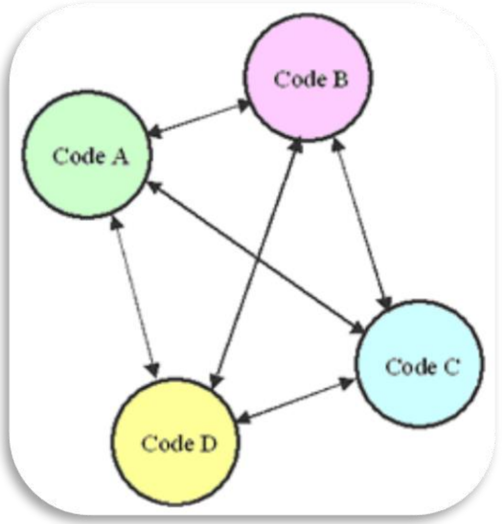
1234 5678

The code is found on the screen in front of you

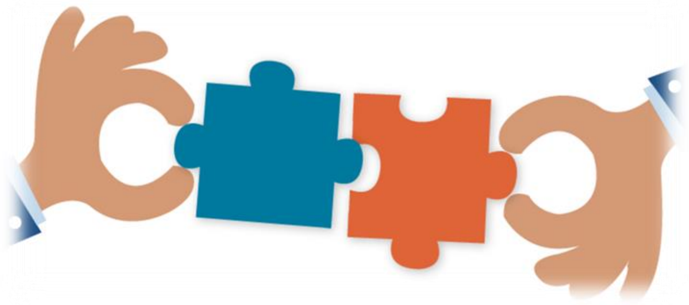
What happens when you type a URL in browser?

Part 2 Content

1. Input and Output Parameters
2. Inter-module Communications
3. Module Coupling
4. Kinds of Coupling

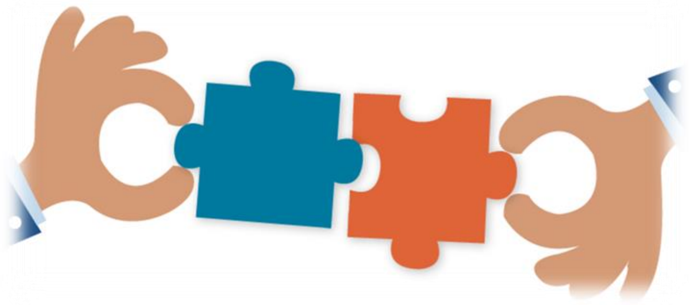


Coupling?

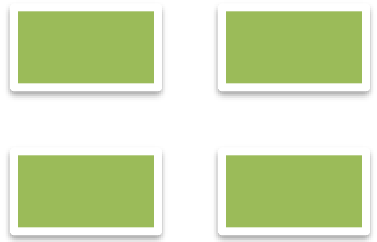


- In order to get the best module we have to manage not only the cohesion but another concept called coupling.
- This is all about the **flow of information** between modules.
- You really want to aim to achieve **total independence** with your modules.
- For this we need fewer and simple connections with other modules.
- Coupling measures the extent of information interchange between modules.
  - **Tight coupling** = large dependence
  - **Loose coupling** = independent and easier to maintain!

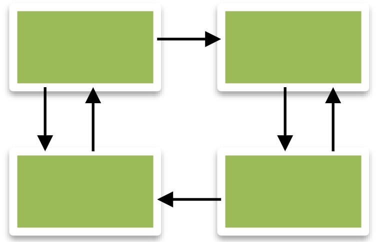
Coupling



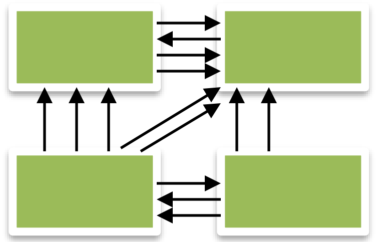
The degree of dependence, such as the amount of *interactions* among modules.



No dependencies



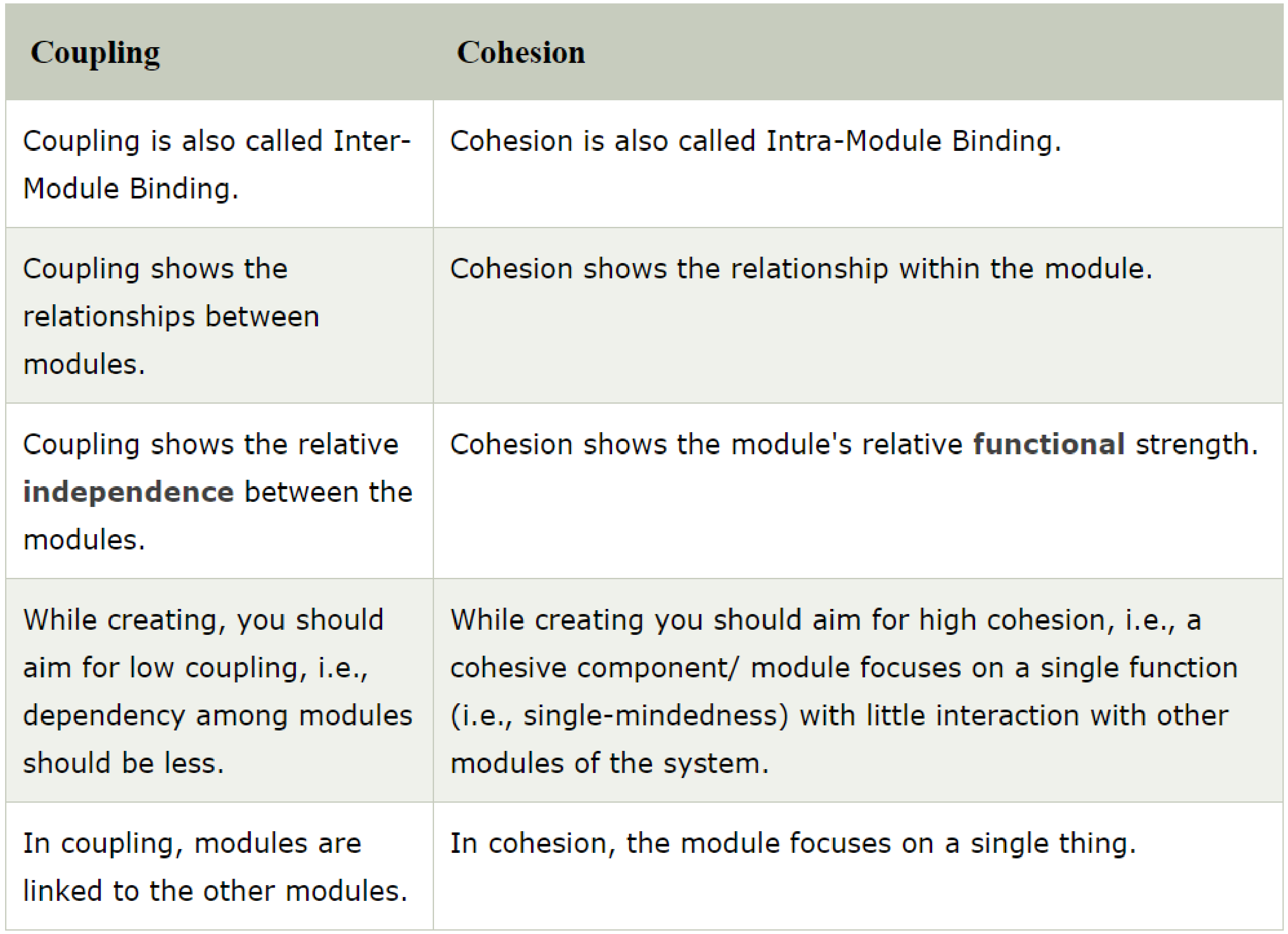
Loosely coupled
some dependencies



Highly coupled
many dependencies

Good modules have strong cohesion and weak coupling

Differentiate between Coupling and Cohesion



Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative **independence** between the modules.	Cohesion shows the module's relative **functional** strength.
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

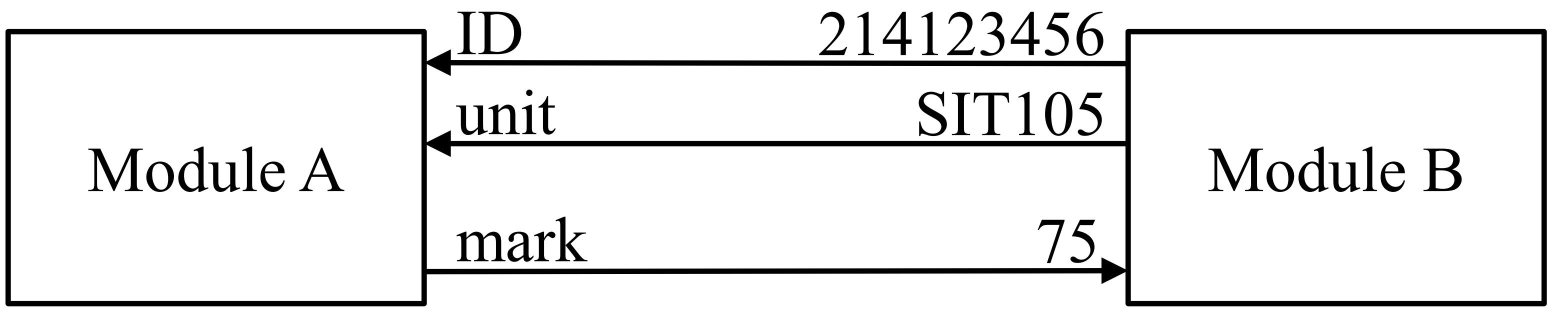
Source: https://www.javatpoint.com/software-engineering-coupling-and-cohesion

Input and Output Parameters

1. In IT, parameters *are also known as* arguments.
2. Input parameters, say **ID** and **unit**, of module A:
   - Contain data, say 214123456 and SIT105, that are provided by another module B, and
   - Are needed by module A to perform its task.
3. Output parameters of module A contain data:
   - Which module A provides to module B

Parameters are the variables!

Are they dependant on eachother?



Return Parameters

Some real programming languages use “return” values, consider these as an output parameter.

Data returned from **one module can be sent to another module**.

The return statement ends function execution and specifies a value to be returned to whatever **called it.**

Input Parameters and Return Value

```
Main
    area = surfaceAreaBox(2, 5, 6)
    PRINT area
END
```

```
surfaceAreaBox(width, height, depth)
    top = width * depth
    front = width * height
    side = depth * height

    surfaceAreaBox = 2 * (top + front + side)

    return(surfaceAreaBox)
END
```

Parameter values

Input parameters

Return value

1

Input and Output Parameters

```
Main
    surfaceAreaBox(area, 2, 5, 6)
    PRINT area
END
```

```
surfaceAreaBox(surfaceArea, width, height, depth)
    top = width * depth
    front = width * height
    side = depth * height

    surfaceArea = 2 * (top + front + side)
END
```

Parameter values

Input parameters

Output Parameter

Input and Output Parameters

```
Main
    surfaceAreaBox(area, 2, 5, 6)
    PRINT area
END
```

```
surfaceAreaBox(surfaceArea, width, height, depth)
    top = width * depth
    front = width * height
    side = depth * height

    surfaceArea = 2 * (top + front + side)
END
```

3

Inter-module Communications

Communication between modules!

Inter-module Communication relates to:

- global data, or (poor programming)
- passing parameters (good programming)

The **kinds** of inter-module communications are:

1. Global data
2. Local data
3. Passing parameters (we just looked at this)
4. Sharing data

Lets look at this

1. Global Data

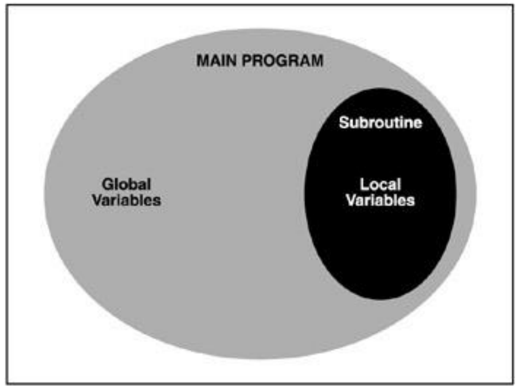
- **Global data** is accessible to every module in a program, therefore modules can communicate using **global data**
- E.g. We have say variable X, any module can modify it if they want.
- For example, consider a spreadsheet.
  - Values on one worksheet can accessible in another.

Always try to reduce the scope of your data / variables!

2. Local Data

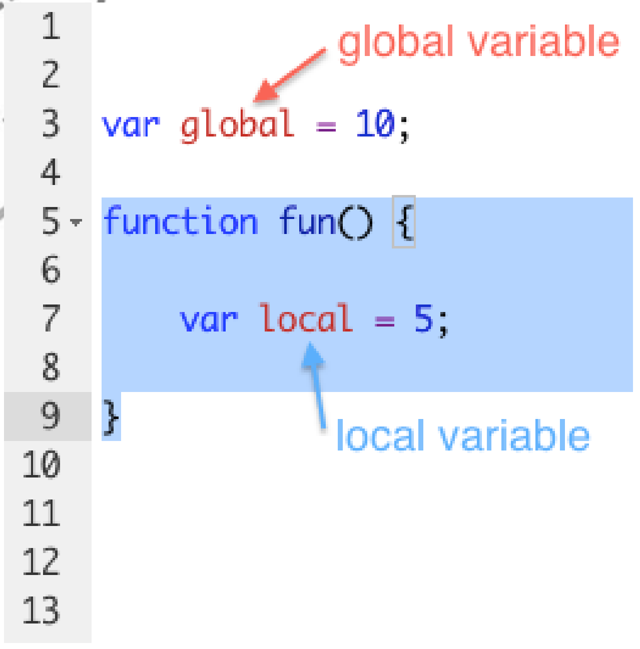
- Local data minimises inter-module communication
- A variable declared inside a module
  - – is accessible by code **within** this module
  - – is **not (directly)** accessible by other modules

NOTE – it is good practice to declare local variables as much as possible!

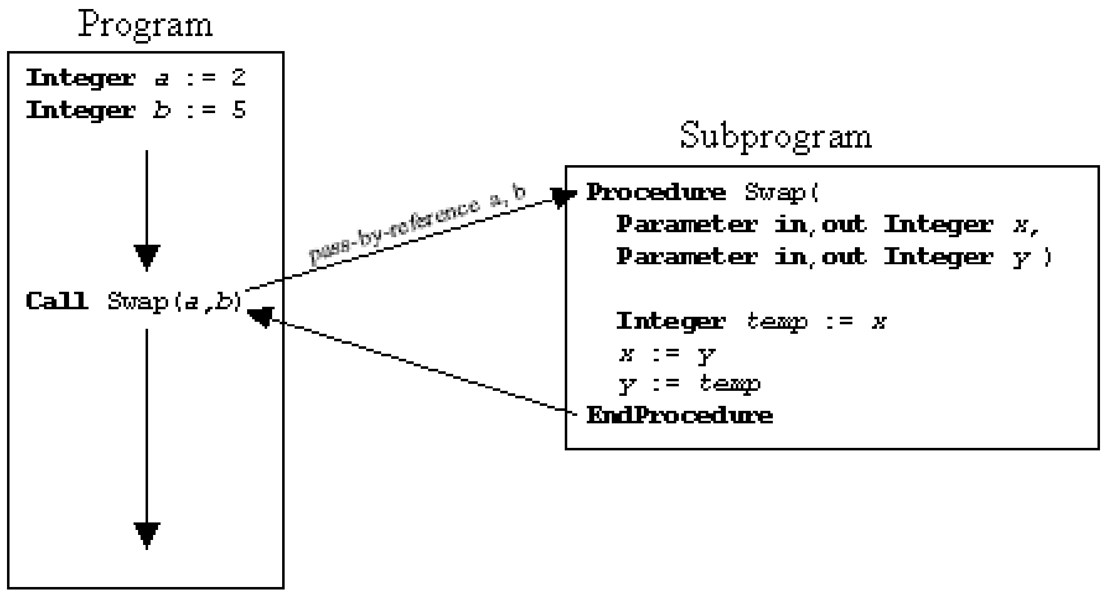


3. Passing Parameters

- Modules communicate by passing data to each other using:
  - parameters (arguments), or
  - global variables



```

var global = 10;

function fun() {

    var local = 5;

}
```



```
Integer a := 2
Integer b := 5

Call Swap(a,b)
```

```
Procedure Swap(
  Parameter in,out Integer x,
  Parameter in,out Integer y )

  Integer temp := x
  x := y
  y := temp
EndProcedure
```

3. Passing Parameters

<u>**Passing data may be done for 3 reasons:**</u>

1. A calling module (say MAIN) passes data to the called module (PRINT)

   e.g., PRINT “Final mark is”, mark

   PRINT is taking 2 input parameters

2. A calling module (say MAIN) expects to receive data from the called module (READ)

   e.g., READ studentID, unitcode

   READ is sending back 2 output parameters

3. A calling module (say MAIN) passes data which the called module (SORT) alters and sends back to the calling module

   e.g., SORT studentNames

   Rearrange the data and send it back

4. Sharing Data

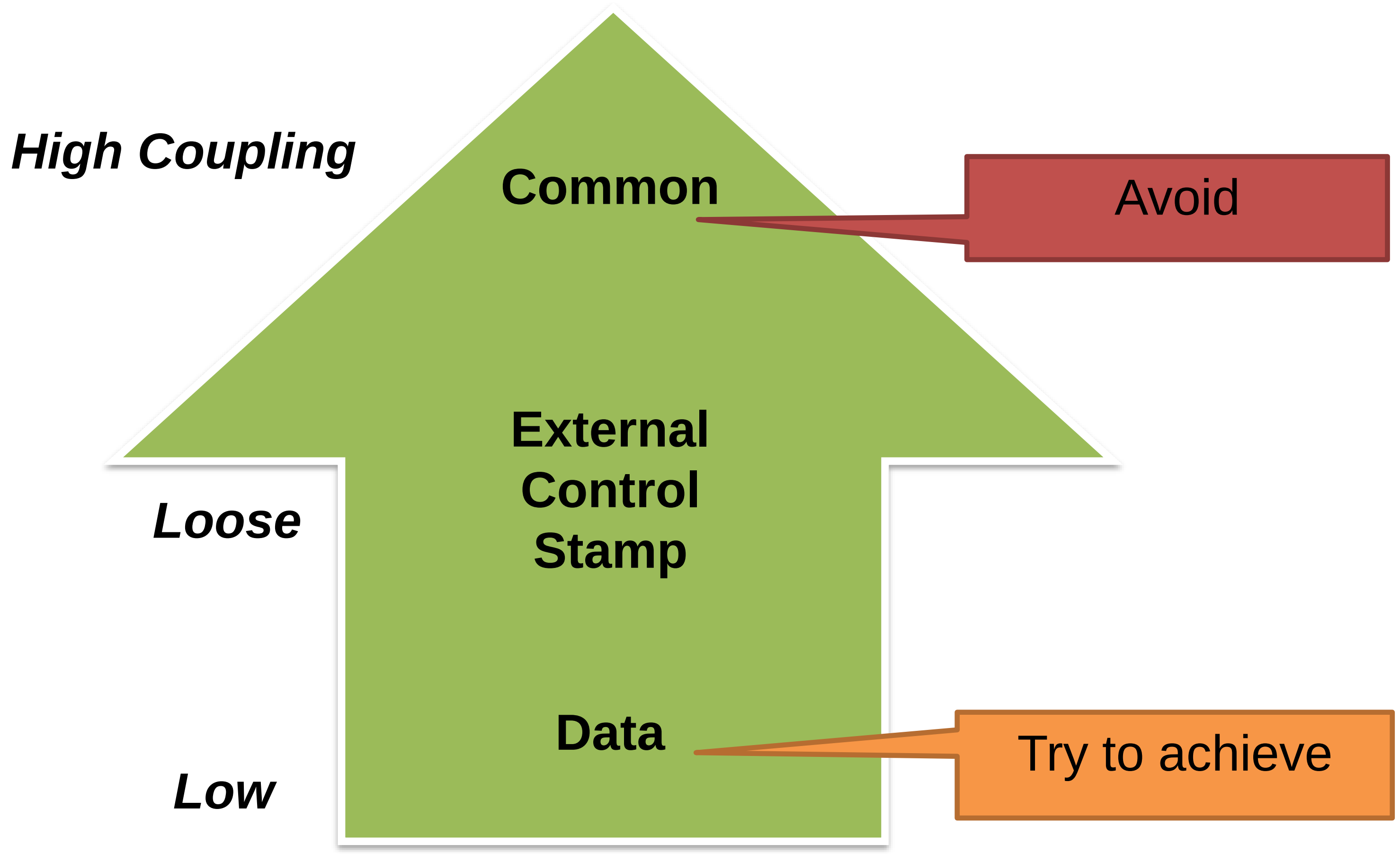
- The less data shared by a module with other modules, the more **easier** it is to alter that module.
- When a module shares many data, it is said to be **tightly coupled** (many data connections).
- The more **tightly coupled**, the **less independent** a module is.

Module Coupling

- This outlines the dependency between modules with respect to ***exchanging data.***
  - Does Module A depend on Module B ? What is their dependence?

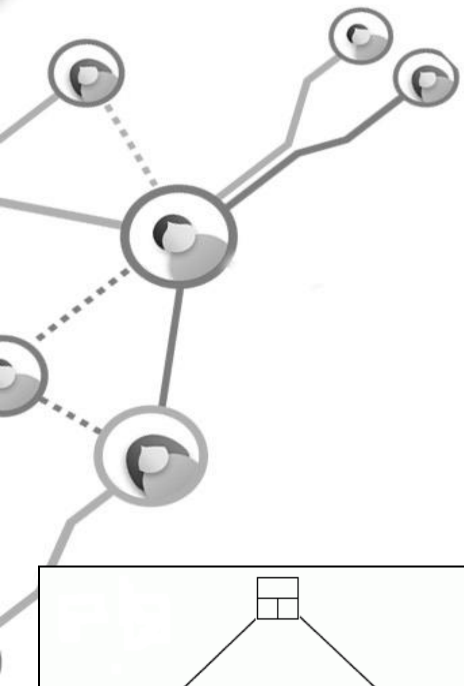
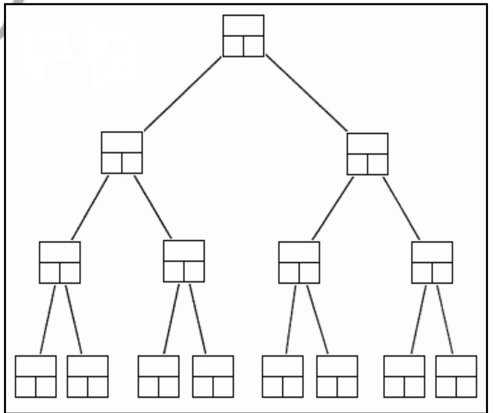
- Loose coupling is best!
  (i.e., low dependence between modules).
  - Much more independence and easier to change.

Type of Coupling



1. Common Coupling: Data structures

- A ***data structure*** is a scheme for organizing data in the memory of a computer.
- Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs.
- The way in which the data is organized affects the performance of a program for different tasks.

CODE: Data Structures: Briefly

A data structure may be selected or designed to store data for the purpose of working on it with various algorithms.

In the below example:

- The structure type is called **product**, and defines it having two members: **weight** and **price**
- This declaration creates a new type (product)
- This is used to declare three objects (variables) of this type: **apple, banana, and melon**.

```
struct product {
int weight;
double price;
} ;
product apple;
product banana, melon;
```

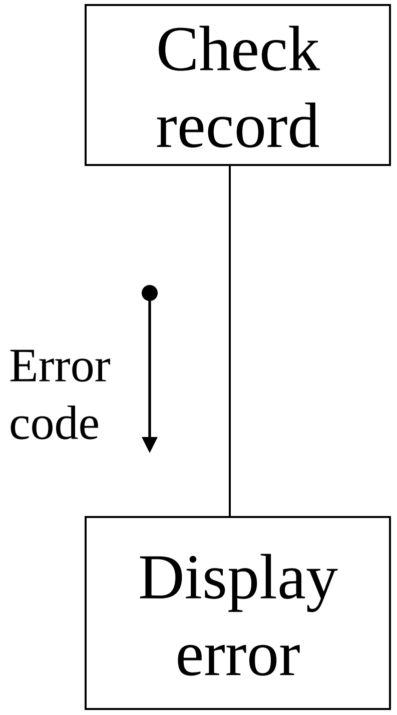
2. External Coupling

- Two modules share something externally imposed. For instance an:
  - External file
  - Device interface
  - Protocol
  - Data format
- Only allows for **variables of fundamental types** such as integers, character, floats to be **globally** accessible.
  - Reduces the chance of common coupling.

3. Control Coupling

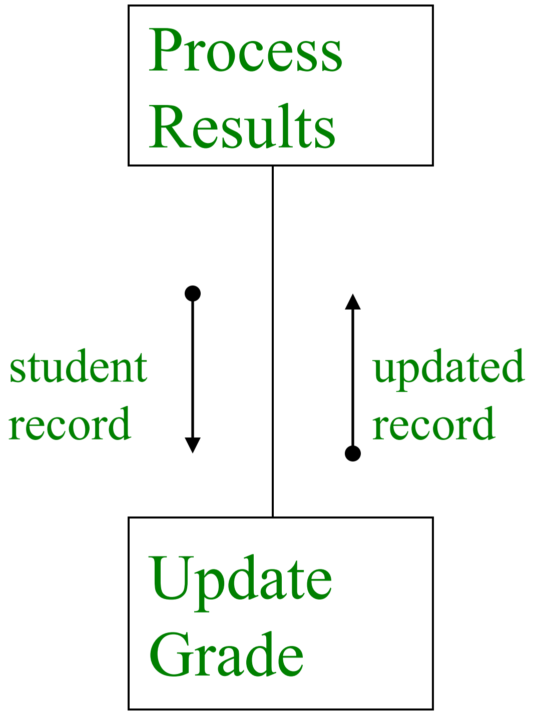
- A module controls the logic of another module through the **parameter** it sends to it.
- Controlling module needs to know how the other module works - ***not flexible!***

- Why is this bad?
  - Modules are not independent
  - Caller must understand some of the details about how the logic of the callee is organized.
  - E.g. what inputs does it take and what do they do



4. Stamp Coupling

- One module passes its own **local data structure** to another module.
- Dependency (again, if you change one module you will have to change the other).
- E.g. Update Grade (student record)
  - Both need to know how a ‘record’ is structured.

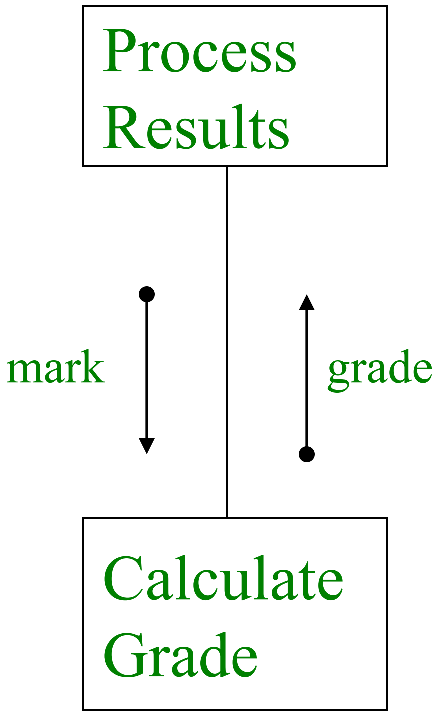


5. Data Coupling

Mark = local to Process Results

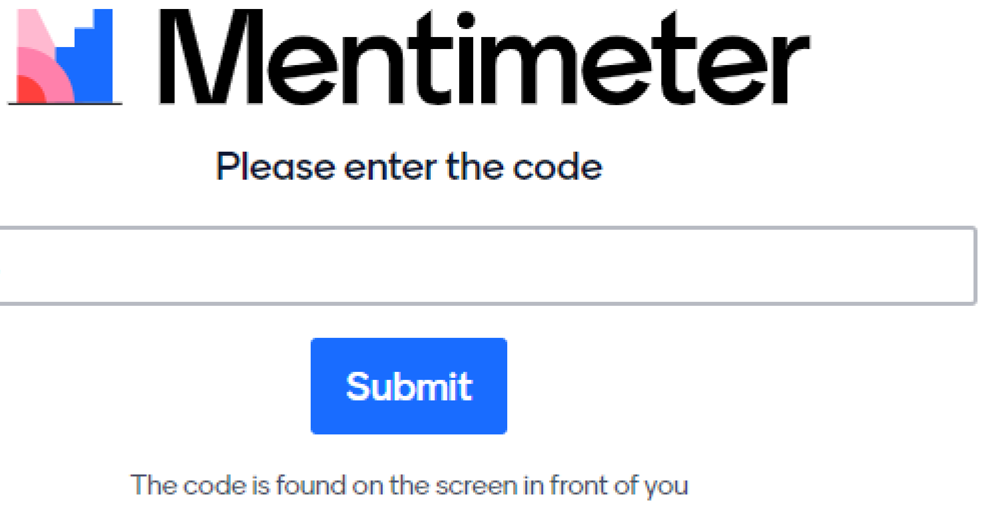
Grade = local to Calculate Grade

- Only **local** data of **fundamental types** are passed.
  - Modules communicate by parameters.
  - Each parameter is necessary to the communication.
- This is the **safest** method.
  - Since all variables are of basic types (fundamental types (e.g. int, string)), it is not possible to change their structure.



Question Time!

- What happens when you search in Google/Baidu?



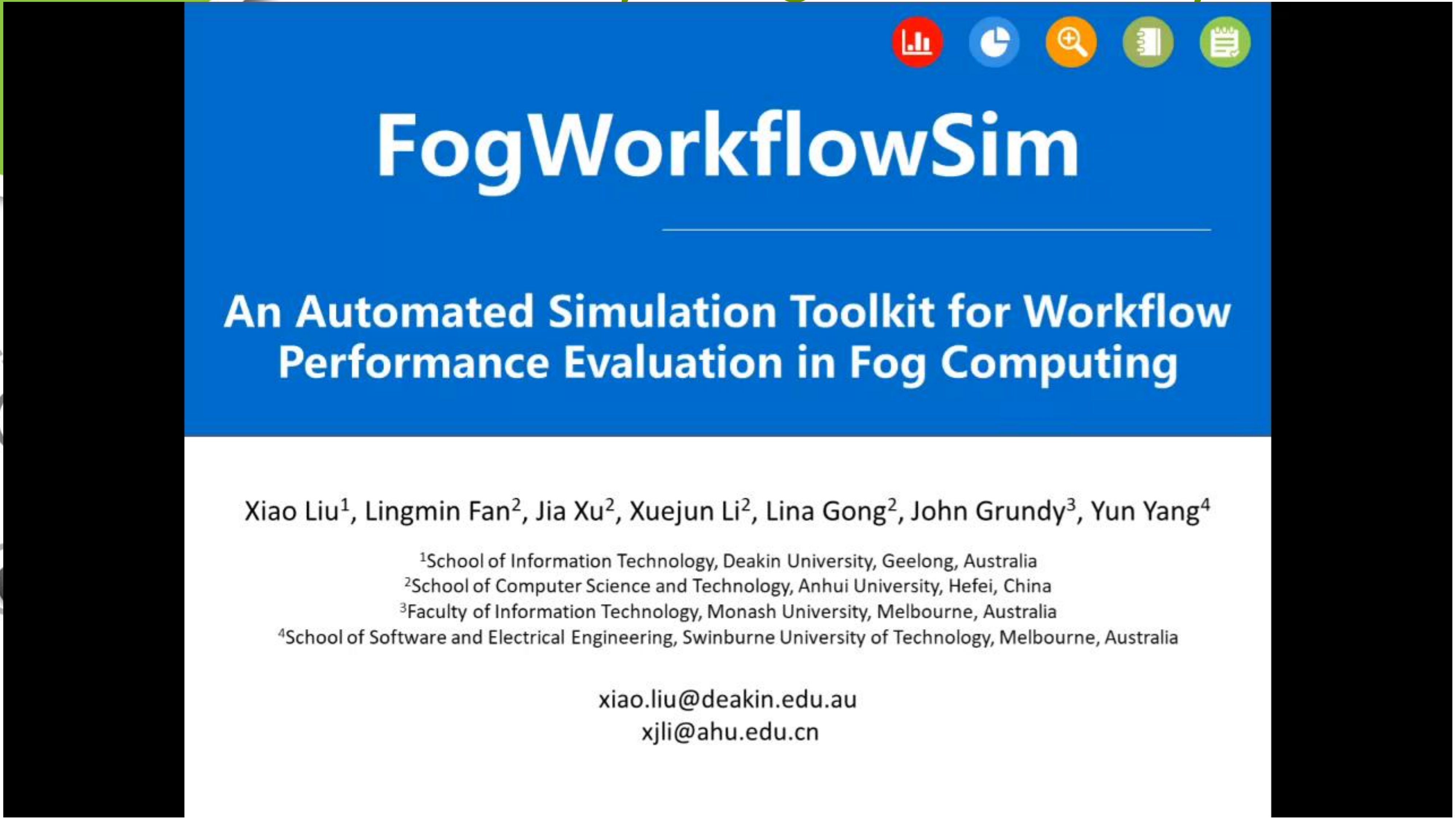
Please enter the code

1234 5678

The code is found on the screen in front of you

What happens when you search in Google?

Case Study – Edge Workflow System



FogWorkflowSim

An Automated Simulation Toolkit for Workflow Performance Evaluation in Fog Computing

Xiao Liu[1], Lingmin Fan[2], Jia Xu[2], Xuejun Li[2], Lina Gong[2], John Grundy[3], Yun Yang[4]

[1]School of Information Technology, Deakin University, Geelong, Australia
[2]School of Computer Science and Technology, Anhui University, Hefei, China
[3]Faculty of Information Technology, Monash University, Melbourne, Australia
[4]School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, Australia

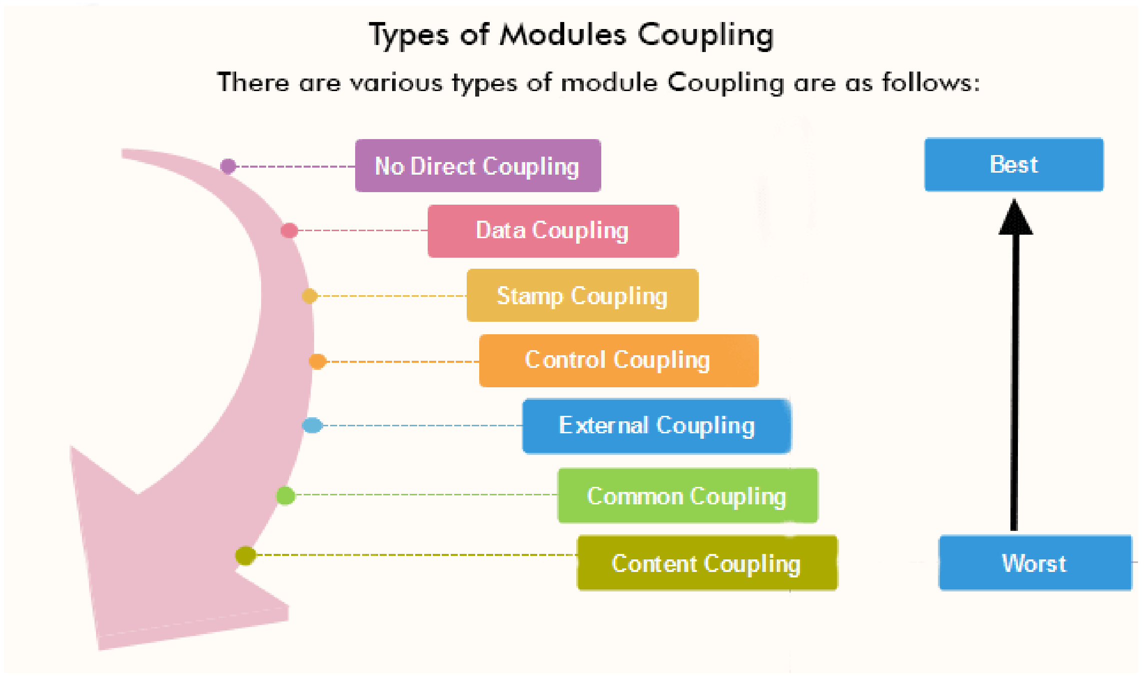
xiao.liu@deakin.edu.au
xjli@ahu.edu.cn

X. Liu et al., "FogWorkflowSim: An Automated Simulation Toolkit for Workflow Performance Evaluation in Fog Computing," 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 2019, pp. 1114-1117,
Jia Xu, Ran Ding, Xiao Liu, Xuejun Li, John Grundy, Yun Yang, EdgeWorkflow: One click to test and deploy your workflow applications to the edge, Journal of Systems and Software, Volume 193, 2022,

Part 2 Summary

- Avoid using global data!
  - Don't let your modules play with global data.
- Use parameter passing – much safer.
- Good modules have strong cohesion and weak coupling

- You cant ever have completely uncoupled modules.
  - If you do, your solution is not a working system.
- Systems are made of interacting components.



Source: https://www.javatpoint.com/software-engineering-coupling-and-cohesion