Software Design
- Modularity -

Fall, 2024

jehong@chungbuk.ac.kr

Topics Covered

Modularity

Coupling

- Message Coupling
- Data Coupling
- Stamp Coupling

  :

- Content Coupling

Cohesion

- Functional Cohesion
- Sequential Cohesion
- Communicational Cohesion

  :

- Coincidental Cohesion

Software Design

A process through which requirements are translated into a representation of software.

Software design methodology relatively lacks the depth, flexibility and quantitative nature in comparison to classical engineering fields.

Q: What is importance of software design?

- Design is the place where _____ is fostered.
- Design provides us with representations of SW that can be assessed for _____ .

Characteristics of Good Design

A design should exhibit a hierarchical structure.

A design should be modular.

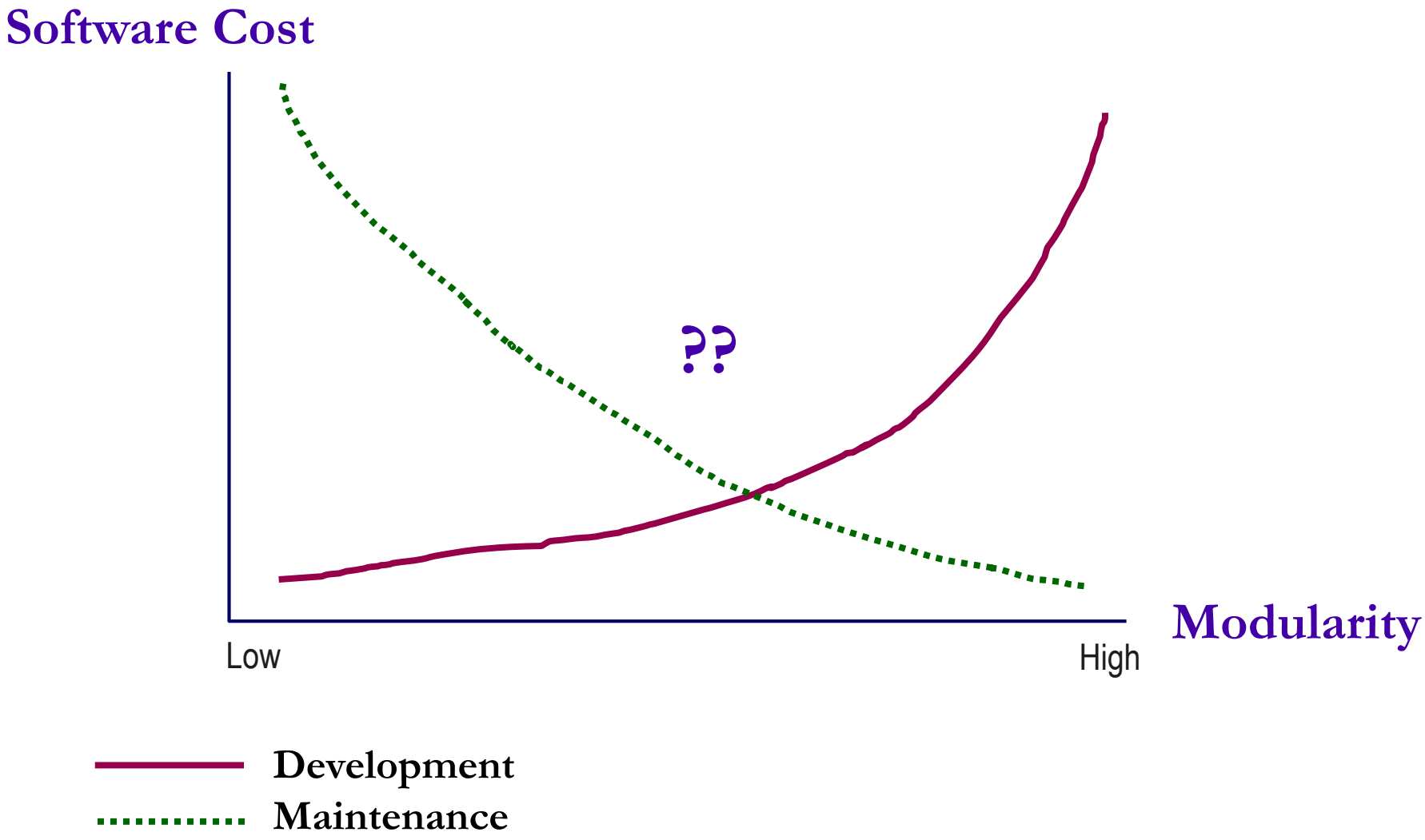
A design should contain distinct and separable representation of data and procedure.

A design should lead to modules that exhibit independent functional characteristics.

A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.

A design should easy to accommodate change.

Modularity and Software Cost



Effective Modular Design

Functional independence

Based on the concepts of abstraction and information hiding

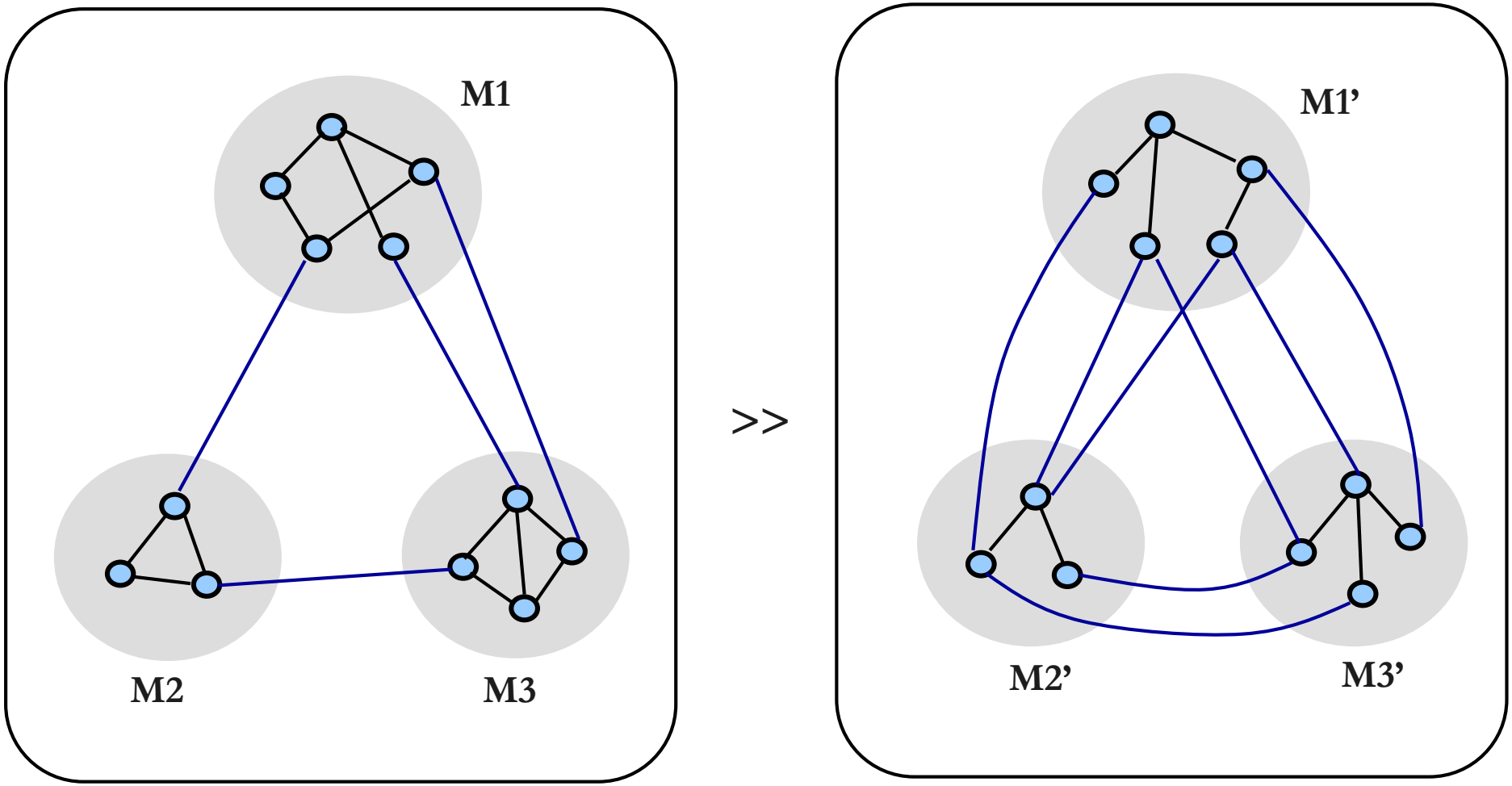
Each module has

- a specific functionality of requirements
- a simple interface when viewed from outside

Measured by two quantitative criteria

- **Coupling**
- **Cohesion**

Modularity



Modularity – Coupling

jehong@chungbuk.ac.kr

Coupling

Degree of interdependence between two modules.

Low coupled system

- Prevent ripple effect
- Enhance understandability

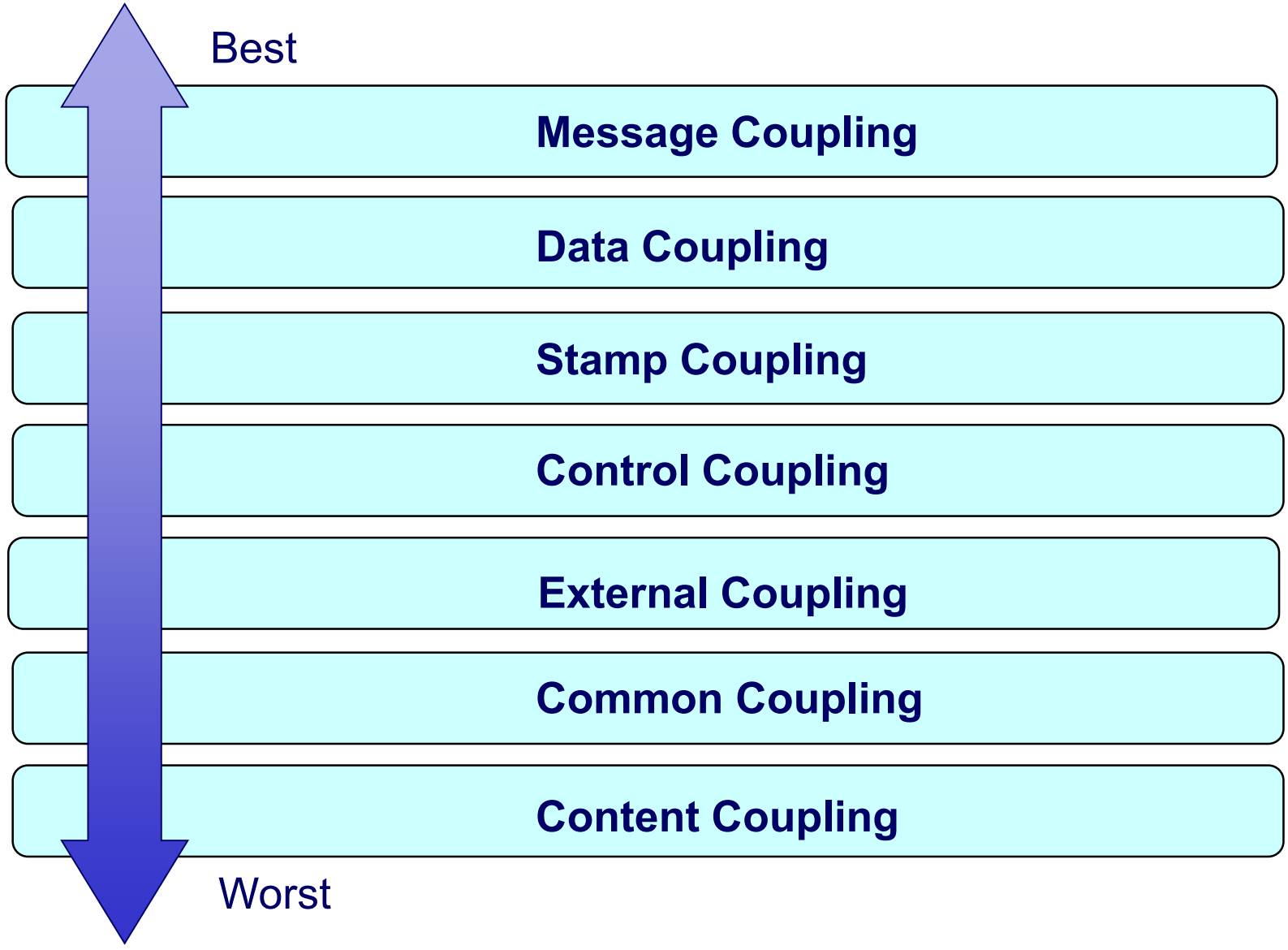
Objective: Minimize coupling

- Make modules as independent as possible.
- Indicates a well-partitioned system.

How to achieve low coupling

- Eliminating unnecessary relationships
- Reducing the number of necessary relationships

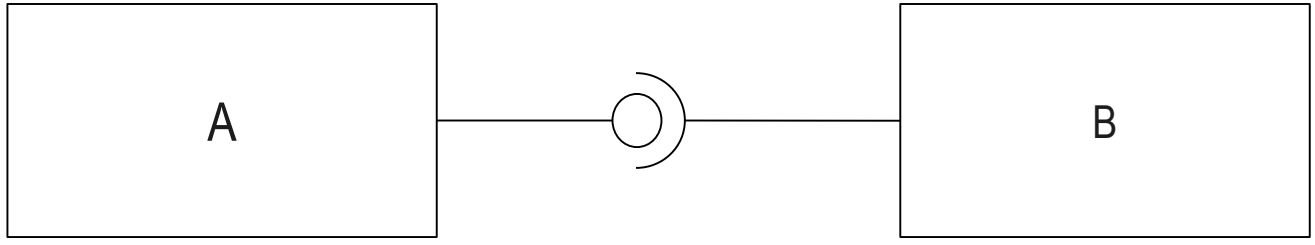
Scale of Coupling



Message Coupling

This is the loosest type of coupling. It can be achieved by state decentralization (as in objects) and component communication is done via parameters or message passing.

Modules are not dependent on each other, instead they use a public interface to exchange parameterless messages (or events).

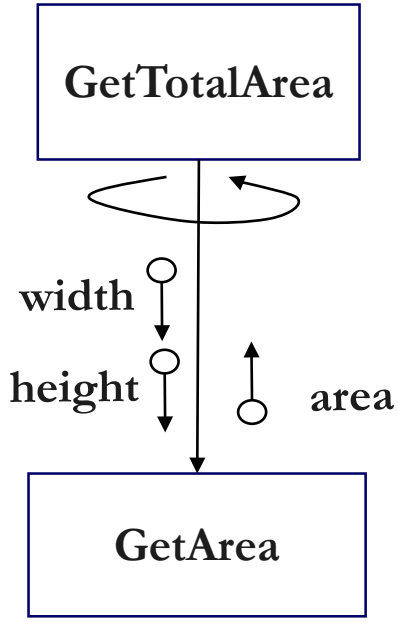


Data Coupling

Two modules are data coupled if they communicate by parameters, each parameter being an elementary piece of data.

Since modules must communicate, data coupling is unavoidable and is quite harmless as long as it's kept to a minimum

```
int GetTotalArea()
{
    int totalArea = 0 ;
    for ( i  = 0 ; i < count ; i ++ ) {
        totalArea += GetArea(shapes[i].width, shapes[i].height) ;
    }
    return  totalArea ;
}
```



Warnings on Data Coupling

Avoid wide interface; keep the interface as narrow as possible

```
int ReadCustomerInfo(char* ID, char* name, char* address, char* phone, char* SSN) {

}
int StoreCustomerInfo(char* ID, char* name, char* address, char* phone, char* SSN) {

}
```

Recommendation: Define and use composite data

```
typedef struct _customerInfo {
    char *name, *address, *phone, *SSN ;
} CUSTOMER_INFO ;

int ReadCustomerInfo(char* ID, CUSTOMER_INFO* info) {

}
int StoreCustomerInfo(char* ID, CUSTOMER_INFO info) {

}
```

Warnings on Data Coupling

Avoid TRAMP DATA

- Tramp data is a piece of information that shuffles aimlessly around a system, unwanted by and meaningless to most of the modules through which it passes
- The more modules a piece of tramp data travels through, the more likely that it will be accidentally altered, or that it will spread a defect

Recommendation

- Restructure the module structure

Stamp Coupling

Two modules are stamp coupled if one passes to the other a composite piece of data

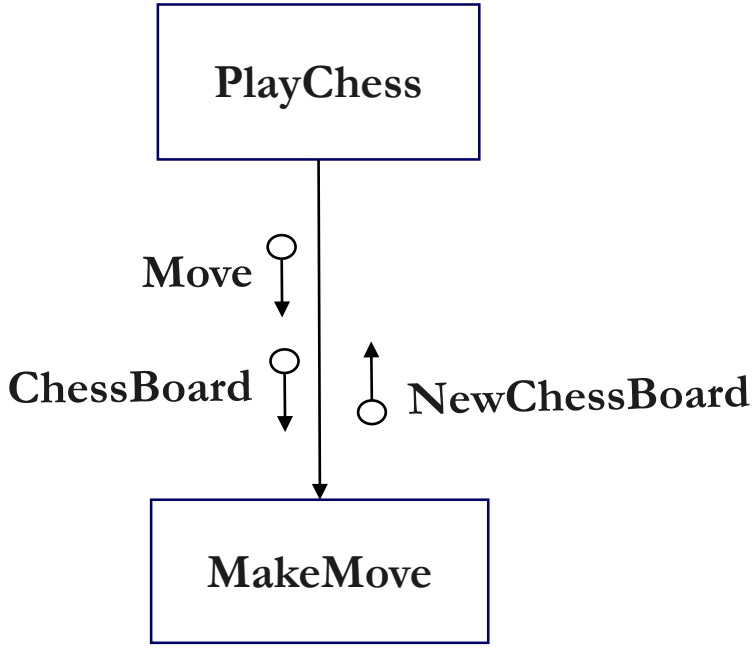
Stamp coupled module is fine if and only if the data structure is natural to the application and there is no gratuitous obscurity.

```
typedef  struct _customerInfo {
    char *name, *address, *phone, *SSN ;
} CUSTOMER_INFO ;

int ReadCustomerInfo(char* ID, CUSTOMER_INFO* info) {


}

int StoreCustomerInfo(char* ID, CUSTOMER_INFO info) {


}
```

Stamp Coupling

The data structure can introduce some indirectness

However, it is a small price to pay for avoiding an unconscionably broad interface



Warnings on Stamp Coupling

Never pass records containing many fields to modules that need part of those fields

```
typedef struct _customerInfo {
    char *name, *address, *phone, *SSN ;
} CUSTOMER_INFO ;

int ValidatePhoneNumber(CUSTOMER_INFO* info) {

    /* use only phone field of CUSTOMER_INFO */
}
```

It creates dependencies between unrelated modules

It introduces extra obscurity, reduces flexibility and involves superfluous data

Warnings on Stamp Coupling

Recommendation: Narrow the interface of the module so that the part of the data structure that is actually used is passed to the called module.

```
int ValidatePhoneNumber(char* phone)
{


}
```

Warnings on Stamp Coupling

Bundling means collecting unrelated data items into an artificial data structure

```
int CalcTototalPurchaseCost(int pricePerItem, int dicountType, int itemCount, int tax) {

}
```

```
typedef  struct _stuff {
     int pricePerItem, dicountType,  itemCount, tax ;
} STUFF ;

int CalcTototalPurchaseCost(STUFF stuff) {

}
```

Bundling offers you nothing but needless obscurity

Warnings on Stamp Coupling

Recommendation: Never collect unrelated data items into an artificial data structure

Should define a data structure that is meaningful to the application domain; that is, adopt the bundling if the domain expert can understand its meaning.

Signs of bundling

A bundled data structure tends to have vague or meaningless name. e.g) info, stuff, …

Recommendation: Untie the bundling and Define the interface using the elementary data and simple composite type.

Control Coupling

Two modules are control coupled if one passes to the other a piece of information intended to control the internal logic of the other

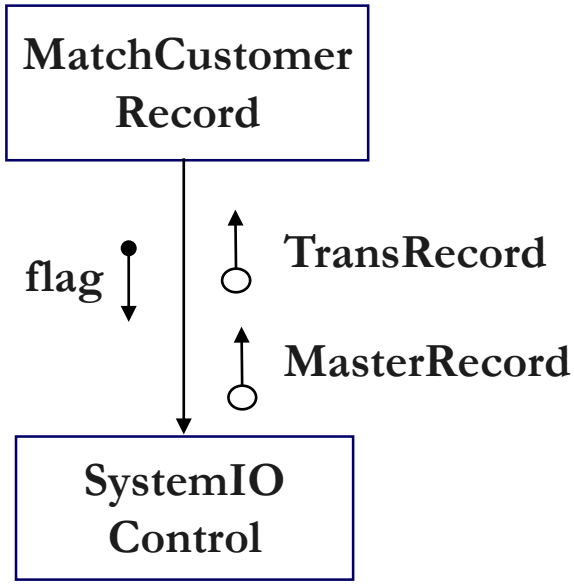
Direction of Control

Forward: The caller controls the internal flow of the callee

Backward: The callee controls the internal flow of the caller

Control Coupling

Forward Control



- The value of flag indicates to SystemIOControl module which record(s) to read
- MatchCustomerRecords explicitly decides which part of the logic of SystemIOControl module to use
- In order for the caller to make such a decision, the caller must know how the logic of the called module is organized

Control Coupling

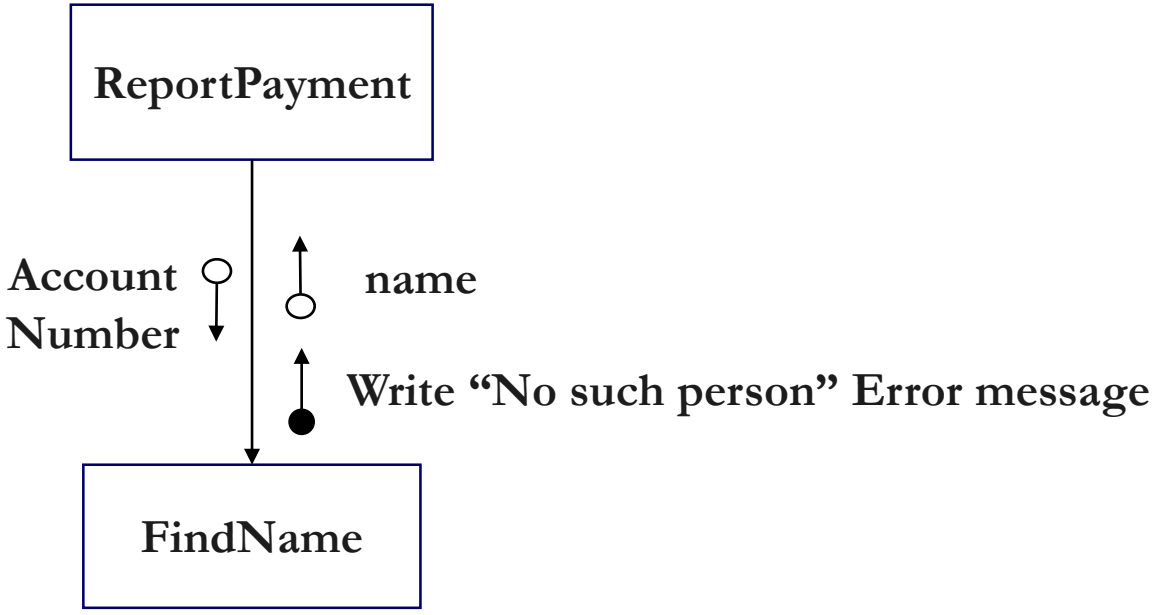
Recommendation: Split the called module by defining new module for each exclusively executed logic

```
int GeneralIORoutine(int flag, void* buffer, int size)  {

    if ( flag == 0 )
        /* READ data and store them into buffer */
    else
        /* WRITE data in buffer */
}
```

```
int Read(void* buffer, int size) {
        /* read data and store them into buffer */
}

int Write(void* buffer, int size) {
        /* write data in buffer */
}
```

Control Coupling

Backward Control; inversion of authority



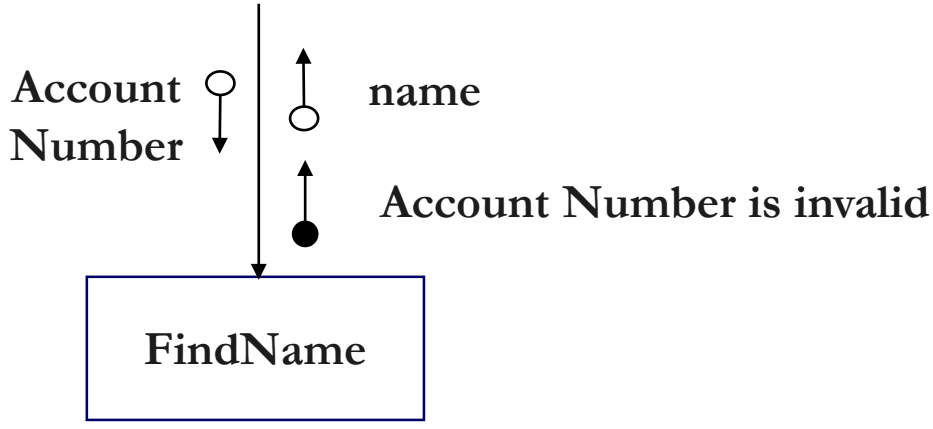
The called module assume some capabilities of the caller module

This reduces the flexibility of the called module

Control Coupling

Recommendation: Use descriptive flag instead of control flag

Type	Type of name	Examples
Control flag	Verb	Read next record Reject this customer Rewind master file
Descriptive flag	Adjective	Egg is rotten Zip code is numeric Transaction file is at end

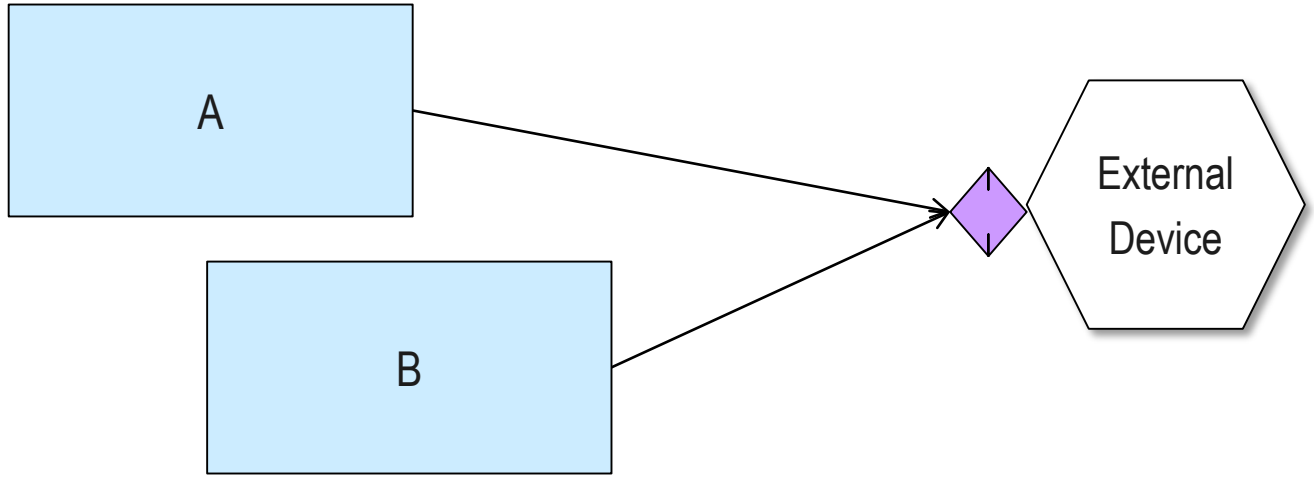


External Coupling

Two components share something externally imposed, e.g.,

- External file
- Device interface
- Protocol
- Data format

Basically related to the communication to external tools and devices



Common Coupling

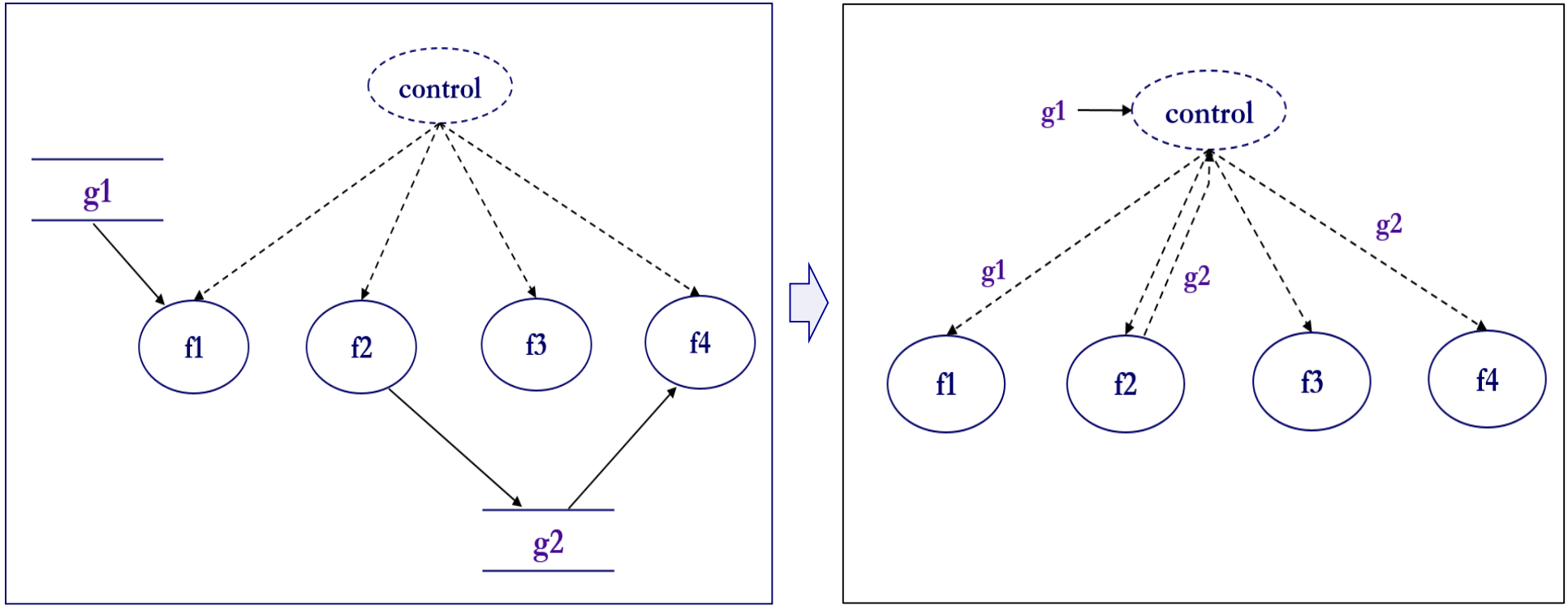
Two modules are common coupled if they refer to the same global data area

Bad reasons for common coupling

- Ripple effect
- Less flexibility of globally coupled module
- Remoteness in time is introduced
- Abuse of the same global area for multiple purpose
- Difficulty in understanding and tracking the use of global area

Common Coupling

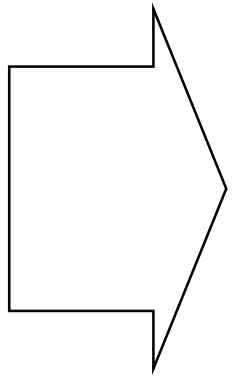
Recommendation: *convert into parameters* the global variables that are meaningful to some modules



Common Coupling

Recommendation: limit the access to the global area by declaring them as *module scoped variables*

```
int gA, gB, gC, gD, gE ;
void f1( )  {
    /* access gA, gB, gC */
}

void f2( ) {
   /* access gA, gB, gC */
}

void f3( ) {
   /* access gD, gE */
}

void f4( ) {
   /* access gD, gE */
}
```



```
static int gA, gB, gC ;
void f1( ) {
     /* access gA, gB, gC */
}

void f2( ) {
     /* access gA, gB, gC */
}
```

```
static int gD, gE ;
void f3( ) {
     /* access gD, gE */
}

void f4( )  {
     /* access gD, gE */
}
```

Common Coupling

Recommendation: Define the *access functions* for each global variable

```
/* A.c */
static int gA;
void setA(int a) {
    gA = a ;
}

int getA( ) {

    return a ;
}
```

```
/* B.c */
static int gB;
void setB(int b) {
    gB = b ;
}

int getB( ) {

    return b ;
}
```

```
void f1( )  {
    int x = getA( ) ;
    setA(x++) ;
}

void f2( ) {
    int y = getB( ) ;
    setB(y+getA( )) ;
}

void f3( ) {
    int z = getD( ) ;
    setE(z+getE( )) ;
}

void f4( ) {
    int u = getD( ) + getE( ) ;
    setE(u) ;
}
```

Content Coupling

Two modules are content coupled if one refers to the inside of the other in any way

- On module branches or falls through into another.
- One module refers to (changes) data within another
- Example:
  - Assembly program
  - 'goto' statement

Data Coupling -> **Stamp Coupling** -> Control Coupling -> **Common Coupling** -> Contents Coupling

(Message Coupling) (External Coupling)

Modularity – Cohesion

jehong@chungbuk.ac.kr

Cohesion

The measure of the strength of functional relatedness of elements within a module

- Elements : instruction, data definition, call to another module, ...

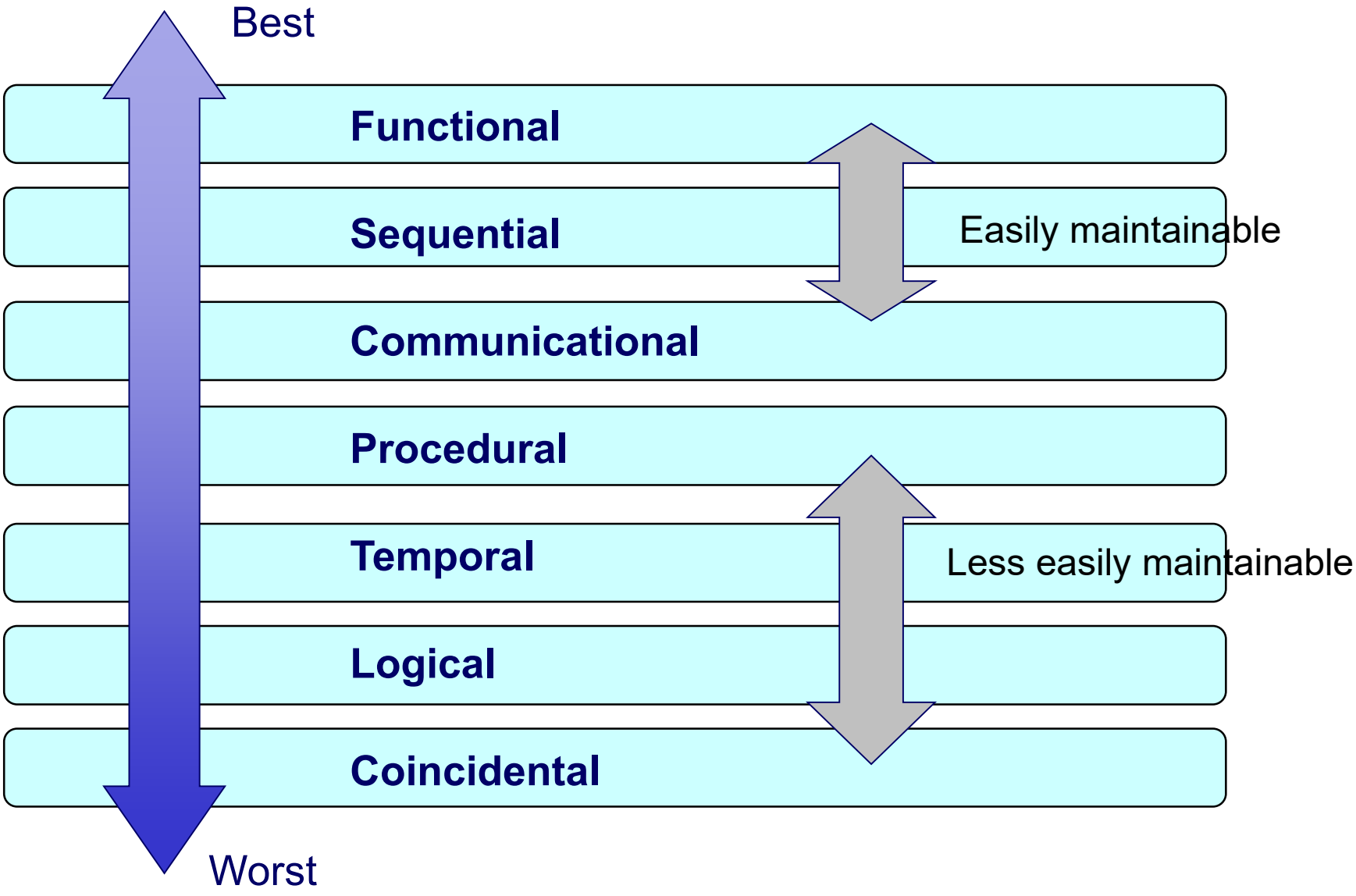
Objective: Maximize Cohesion

- Make modules whose elements are strongly and genuinely related to one another

Relationship between Coupling and Cohesion

- 2 ways to evaluate the partitioning of a system
- The cohesion of a module often determines how tightly it will be coupled to other modules in a system.

Scale of Cohesion



Functional Cohesion

A functionally cohesive module contains elements that all contribute to the execution of one and only one problem-related task

A module of function cohesion has a strong, single-minded purpose

Example

- Compute Cosine of Angle
- Verify Alphabetic Syntax
- Read Transaction Record
- Compute Point of Impact of Missile
- Assign Seat to Airline Customer

It carries out just one job to completion without getting involved in any extracurricular activity

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

```
Clean Car Body ;
Fill In Holes In Car ;
Sand Car Body ;
Apply Primer ;
```

A Sequentially cohesive module usually has good coupling and is easily maintained.

Sequential Cohesion

Recommendation: Try to convert it into functionally cohesive module

```
Module Repaint Car {
    Clean Car Body ;
    Fill In Holes In Car ;
    Sand Car Body ;
    Apply Primer ;
    Put on Final Coat ;
} End Module
```

Sequential cohesion v.s. functional cohesion

- Functionally cohesive module can correspond to some concept in application domain, but sequentially cohesive module cannot.
- Sequential cohesion shows less reusability than functional cohesion

Communicational Cohesion

A communicationally cohesive module is one whose elements contribute to activities that use the same input or output data

```
Module FindInfoOfBook {
    Use book no ;
    Find Title of Book ;
    Find Publisher of Book ;
    Find Author of Book ;
    return title, publisher, author ;
} End Module
```

```
Module Determine Customer Details {
    Use customer account no ;
    Find customer name ;
    Find customer loan balance ;
    return name, loan balance ;
} End Module
```

Communicational cohesion v.s. sequential cohesion

- Have acceptable cohesion and acceptable coupling
- The activities of sequential cohesion must be carried out in a specific order. But, in a communicational cohesive module, order of execution is unimportant

Warnings on Communicational Cohesion

When the caller has interested in the part of the return value : Some module need to find out customer's name but wouldn't be interested in his loan balance

- The module would either have to discard the loan balance(dirty and redundant coupling), or
- would need its own code for finding the customer name(duplication of function)

Warnings on Communicational Cohesion

Recommendation: Define functionally cohesive module for each elementary return value

```
Module Find Customer Name {
    Use customer account no ;
    return name ;
} End Module
Module Find Customer Loan Balance {
    Use customer account no ;
    return loan balance ;
} End Module
```

```
Module Determine Customer Details {
    Use customer account no ;
    name = Call Find Customer Name ;
    loan balance = Call Find Customer Loan Balance  ;
    return name, loan balance ;
} End Module
```

Warnings on Communicational Cohesion

Temptation to share code among the activities that use the same data

```
Module Produce Employee Salary Report and Calculate Average Salary {
    Use Employee Salary Table ;
    sum = 0 ;
    For each e in Employee Salary Table {
        Write Employee Information of e ;
        sum += salary of e ;
    } End For
    Average Salary = sum / Employee count ;
    return Average Salary ;
} End Module
```

What if you want to produce a salary report on only the first 20 employees ?

Warnings on Communicational Cohesion

Recommendation: Define functionally cohesive module

```
Module Produce Employee Salary Report {
      Use Employee Salary Table ;
      For each e in Employee Salary Table
            Write Employee Information of e ;
      End For
} End Module
```

```
Module Calculate Average Salary {
      Use Employee Salary Table ;
      sum = 0 ;
      For each e in Employee Salary Table
            sum += salary of e ;
      End For
      Average Salary = sum / Employee count ;
      return Average Salary ;
} End Module
```

Procedural Cohesion

A procedurally cohesive module is one whose elements are involved in different and possible unrelated activities in which control flows from each activity to the next

```
Module Prepare Supper {
        Clean Utensils From Previous Meal ;
        Prepare Turkey For Roasting ;
        Make Phone Call ;
        Take Shower ;
        Chop Vegetables ;
        Set Table ;
} End Module
```

They are related by order of execution rather than by any single problem-related function

Procedural Cohesion

Recommendation: convert it into sequential/functional cohesion by removing some unrelated activities

```
Module Prepare Supper {
    Clean Utensils From Previous Meal ;
    Prepare Turkey For Roasting ;
    Chop Vegetables ;
    Set Table ;
} End Module
```

Procedural Cohesion

Use a single loop to do two distinct activities

```
Module compute table-A-sum and table-B-sum {
      use table-A, table-B ;
      table-A-sum = 0 ;
      table-B-sum = 0 ;
      for i = 0 to 100 {
            table-A-sum  += table-A(i) ;
            table-B-sum  += table-B(i) ;
      } end for
      return table-A-sum, table-B-sum ;
} End Module
```

- table-A and table-B are completely unrelated tables where they just happen to have 100 elements each
- What if the length of table-B changes to 120 or if the elements in table-B become in some way dependent on table-A-sum

Recommendation: Split it into two functions

Temporal Cohesion

A temporally cohesive module is one whose elements are involved in activities that are related in time

Elements are grouped by when they are processed.

```
Module Late Evening Scene {
    Put Out Milk Bottles ;
    Put Out Cat ;
    Turn Off TV ;
    Brush Teeth ;
} End Module
```

The programmer is tempted to share code among activities related only by time, and the module is difficult to reuse, either in this system or in others

Temporal Cohesion

Examples of temporal cohesive module

- Initialize all variables prior to store data
- Closes all open files
- Creates an error log
- Process all messages for user notification

Recommendation: In the case of initialization and termination, define separate initialization and termination module for each functionally-related data instead of mighty module.

Logical Cohesion

A logically cohesive module is one whose elements contribute to activities of the same general category in which activities to be executed are selected from outside the module

```
Module General IO Routine {
      use input flag ;      /* to choose which function */
      update record-A ;
      if input flag == 1 then
            write record-A to new master file ;
            read tran file1 into record-B ;
      elseif input flag == 2 then
            if record-A == all spaces then
                  read tran file1 into record-B ;
            endif
            read tran file2 into record-C ;
      elseif input flag == 3 then
       …
      return record-B, record-C ;
} End Module
```

Logical Cohesion

Logically cohesive module

- Contain a number of the same general activities
- One of the activities is executed by the given flag parameter
- Some of the parameters will even be left blank

Programmer tend to create logically cohesive module because overlap parts of functions that happen to have the same lines of code or the same buffers – but are not even executed at the same time

Recommendation: Split it into multiple modules of functional cohesion

Coincidental Cohesion

A coincidentally cohesive module is one whose elements contribute to activities with no meaningful relationship to one another

Parts of a module are only related by their location in source code.

Elements needed to achieve some functionality are scattered throughout the system.

Accidental and worst form

Coincidental Cohesion

Example

Module OOOO {
1. Fix car ;
2. Bake cake ;
3. Walk dog ;
4. Fill out astronaut-application form ;
5. Have a beer ;
6. Get out of Bed ;
7. Go to the Movies ;

} End Module

Module OOO {
1. Print next line
2. Reverse string of characters in second argument
3. Add 7 to 5th argument
4. Convert 4th argument to float

} End Module

Comparison of Levels of Cohesion

Cohesion Level	Coupling	Cleanliness of implementation	Modifiability	Understand-ability	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

Refactoring

Definition: a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

External behavior does NOT change and Internal structure is improved

The goals of refactoring are

- NOT to add new functionality
- to make code easier to maintain in the future

Benefits of Refactoring

- Code size is often reduced
- Confusing code is restructured into simpler code
- Both of these greatly improve mainatainability! Which is required because requirements always change!

Refactoring : An Example

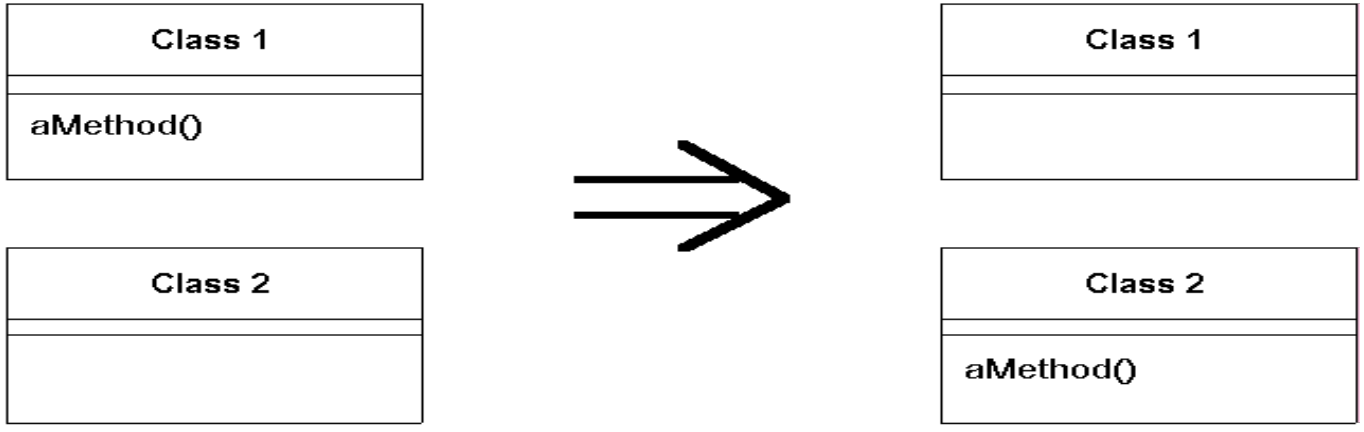
Refactoring Pattern : Move a method

Motivation

- A method is, or will be, using or used by more features of another class than the class on which it is defined.

Technique

- Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.

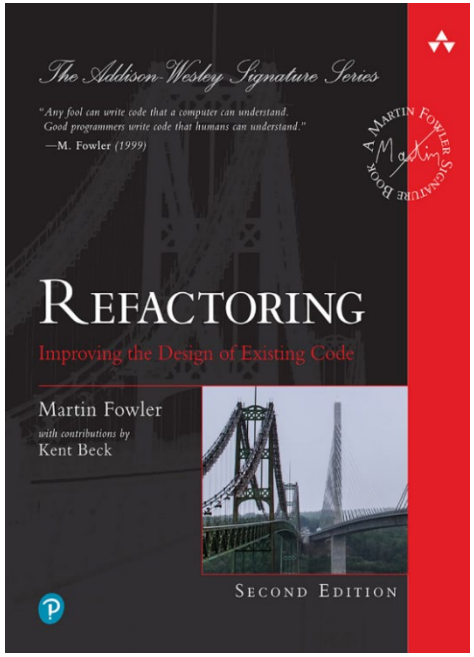


Refactoring by M. Fowler

Bad Smells in Code : *If it stinks, change it.*

Refactoring Techniques

- 68 techniques for code refactoring (sp, object-oriented language)



Refactoring Type : Moving Features Between Objects

R10	**Move Method**
R11	**Move Field**
R12	**Extract Class**
R13	**Inline Class**
R14	**Hide Delegate**
R15	**Remove Middle Man**
R16	**Introduce Foreign Method**
R17	**Introduce Local Extension**

Summary and Discussion

Good Software Design

- Modularity principle is required
- Maintainability quality is fostered

Criteria for Modularity

- **Coupling, Cohesion**

Code Refactoring

Which phase can improve the modularity of software through software life cycle ?

Explain the relationship between Coupling and Cohesion ?