



 ĐẠI HỌC BÁCH KHOA HÀ NỘI
 VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Bài 10: Các khái niệm trong thiết kế

1

Nội dung


1. Cách thức thiết kế
2. Móc nối linh hoạt (low coupling)
3. Tính kết dính cao (high cohesion)


 VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

2

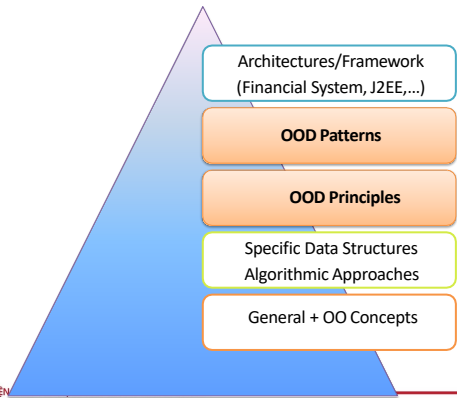
Nội dung

1. Cách thức thiết kế
2. Móc nối linh hoạt (low coupling)
3. Tính kết dính cao (high cohesion)


 VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

3

Các mức thiết kế




Architectures/Framework (Financial System, J2EE,...)

OOD Patterns

OOD Principles

Specific Data Structures
Algorithmic Approaches

General + OO Concepts


 VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

4

Quá trình thiết kế

❖ Định nghĩa:

- Thiết kế là quá trình tìm và mô tả giải pháp
 - Thực thi các yêu cầu chức năng
 - Đảm bảo các yêu cầu phi chức năng
 - Bao gồm cả yêu cầu về ngân sách
 - Tuân thủ các nguyên lý đảm bảo chất lượng thiết kế

Thiết kế là phải ra các quyết định

❖ Người thiết kế phải đối mặt với rất nhiều vấn đề

- Mỗi vấn đề sẽ có một vài giải pháp
 - Các lựa chọn thiết kế
- Người thiết kế phải ra quyết định, chọn giải pháp cho từng vấn đề
 - Có nhiều lựa chọn
 - Giải pháp vấn đề này ảnh hưởng tới giải pháp của vấn đề khác

Ra quyết định

❖ Khi ra quyết định, người kiến trúc sư phần mềm phải sử dụng

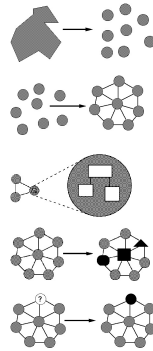
- Các yêu cầu phần mềm
- Các thiết kế đến hiện tại
- Các công nghệ sẵn có
- Các nguyên lý và hướng dẫn thiết kế phần mềm
- Kinh nghiệm trong quá khứ

Modules

- ❖ Phần mềm đều được thiết kế theo hướng module hóa
- ❖ Module là khái niệm tổng quát. Nó có thể là một phương thức, lớp, package hoặc là một đơn vị thiết kế bất kỳ nào
- ❖ Thiết kế module hóa tập trung vào việc định nghĩa các module, vào các đặc tả của chúng, cách chúng liên kết với nhau, không tập trung vào việc thực thi các module

Các tính chất cần đảm bảo khi thiết kế module hóa

- ❖ Tính tách rời được: hệ thống có thể được phân chia thành các module để giảm độ phức tạp, cho phép nhiều người cùng đồng thời làm việc, mỗi người phụ trách một module
- ❖ Tính tổ hợp được: Sau khi chia module, yêu cầu là có thể tích hợp các module lại thành một hệ thống
- ❖ Tính hiểu được: Mỗi module có thể được phát triển, kiểm tra, đánh giá một cách độc lập
- ❖ Tính liên tục theo thời gian: một thay đổi nhỏ trong yêu cầu chỉ ảnh hưởng đến một số lượng nhỏ các module
- ❖ Tính tách biệt: Lỗi trong một module phải bị cô lập theo cách thức tốt nhất có thể, không lan sang module khác



9

Thiết kế Top-down và bottom-up

❖ Thiết kế Top-down

- Trước tiên thiết kế cấu trúc hệ thống ở mức cao
- Sau đó đi chi tiết dần xuống các mức dưới
- Đi đến các quyết định chi tiết như
 - CTDL, thuật toán như thế nào

10

Thiết kế Top-down và bottom-up

❖ Thiết kế Bottom-up

- Xác định các thành phần mức thấp trước
- Sau đó tổ hợp lại dần dần thành các thành phần mức cao

❖ Nên kết hợp thiết kế top-down và bottom-up:

- Thiết kế Top-down giúp hệ thống có cấu trúc rõ ràng, tối ưu
- Thiết kế Bottom-up giúp tạo ra các thành phần có tính tái sử dụng cao

11

Các vấn đề cần thiết kế

- Thiết kế kiến trúc:
 - Chia thành các phần tử, các hệ thống con
 - Cách chúng kết nối với nhau
 - Cách chúng tương tác với nhau
 - Giao diện của chúng
- Thiết kế lớp
- Thiết kế giao diện
- Thiết kế thuật toán
- Thiết kế giao thức

12

Thiết kế tốt

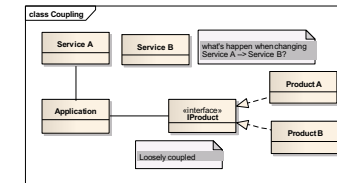
- ❖ Thiết kế tốt phải giúp giảm chi phí phát triển, giúp tăng lợi nhuận. Phải đảm bảo tuân thủ các yêu cầu phần mềm đặt ra, tăng tính tái sử dụng, để bảo trì, tăng hiệu năng, ...
- ❖ Để đánh giá thiết kế: có 2 nguyên lý cơ bản và hữu hiệu nhất về cohesion và coupling

13

Cohesion và Coupling

□ Tính móc nối/phụ thuộc - Coupling

Coupling/Dependency là mức độ 1 module phụ thuộc vào các module khác.



□ Tính kết dính – Cohesion

Cohesion là mức độ liên quan giữa các phần tử trong một module.

14

Các tính chất cần đạt được cho các modules

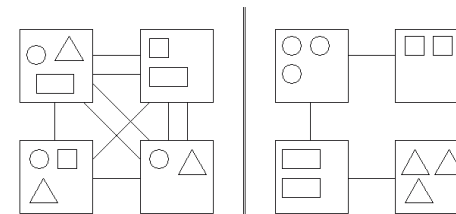
- ❖ Các modules nên độc lập với nhau
 - Tránh một module làm nhiều tác vụ
- ==> high COHESION (tính kết dính cao)
- Một module nên tương tác ít nhất có thể với các module khác
 - Giao diện đơn giản
- ==> low COUPLING (tính phụ thuộc thấp/tính móc nối linh hoạt)

[Steven, Myers, Constantine]

15

Cohesion và Coupling

- ❖ Thiết kế tốt nhất là phải có tính kết dính cao (high cohesion hoặc strong cohesion) trong một module và móc nối linh hoạt lỏng lẻo (low coupling hoặc weak coupling) giữa các module.



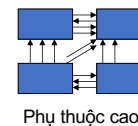
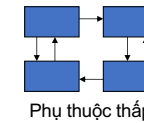
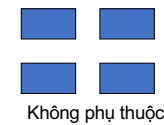
16

Nội dung

1. Cách thức thiết kế
2. **Mức nối linh hoạt (low coupling)**
3. Tính kết dính cao (high cohesion)

17

Coupling: Mức độ phụ thuộc giữa các phần tử



High coupling khiến việc sửa đổi các thành phần trong hệ thống rất khó khăn. Sửa một thành phần sẽ ảnh hưởng tới các thành phần khác kết nối tới nó

18

Đánh giá mức độ Coupling

- ❖ Degree
 - Degree: số lượng kết nối giữa 1 module với các module khác. Càng nhỏ càng tốt
- ❖ Ease
 - Ease: Mức độ dễ dàng khi kết nối 1 module với các module khác. Càng dễ hiểu, dễ làm càng tốt
- ❖ Flexibility
 - Flexibility: các module có thể dễ dàng được thay thế hay không

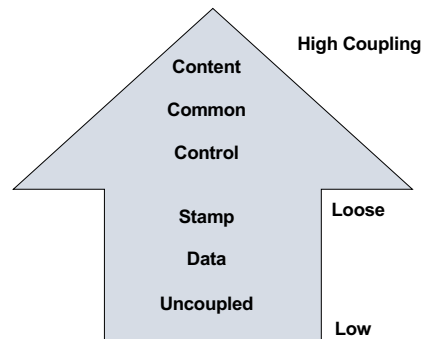
19

Tác hại của tight coupling

- ❖ Thay đổi một module, dẫn đến các thay đổi lan truyền ở các module khác
- ❖ Kết nối các module lại với nhau rất khó khăn
- ❖ Khó tái sử dụng, test các module

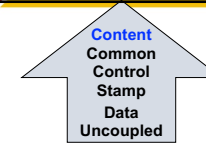
20

Các loại/mức độ Coupling



21

2.1. Content coupling (Móc nối nội dung)



❖ Định nghĩa: Một component tham chiếu đến nội dung của component khác

❖ Ví dụ:

- Component sửa dữ liệu của component khác
- Component chỉnh sửa code của component khác (NNLT bậc thấp)

22

Content coupling:

- ❖ Để tránh 1 component sửa dữ liệu nội bộ trong component khác, trong OOP:
 - Cần thực hiện nguyên lý đóng gói và che giấu dữ liệu, đảm bảo dữ liệu là private, cung cấp getter, setter hợp lý
- ❖ Lợi ích: nếu f thay đổi biến cục bộ của g, khi sửa f ta phải sửa g và ngược lại

23

Ví dụ 1

❖ Hàm bạn trong C++

```
class A {
private:
    int a;
public:
    A() { a = 0; }
    friend class B; // Friend Class
};

class B {
private:
    int b;
public:
    void showA(A& x)
    {
        // Since B is friend of A, it can access
        // private members of A
        std::cout << "A::a=" << x.a;
    }
};
```

24

Ví dụ 2

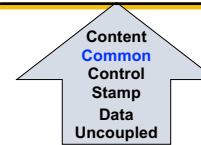
- ❖ "getters" và "setters" có thể vẫn chưa đảm bảo ngăn ngừa content coupling

```
public int sumValues(Calculator c){ // tight coupling
    int result = c.getFirstNumber() + c.getSecondNumber();
    c.setResult(result);
    return c.getResult();
}

public int sumValues(Calculator c){ // loose coupling
    c.sumAndUpdateResult();
    return c.getResult();
}
```

25

2.2. Common Coupling (Móc nối thành phần chung)



- ❖ Định nghĩa: Các module cùng chia sẻ chung định dạng dữ liệu, giao thức truyền thông, ...

- ❖ Không tốt vì

- khó hiểu
- khó xác định component nào ảnh hưởng tới dữ liệu nào
- khó tái sử dụng component
- khó điều khiển truy cập tới dữ liệu, ...

26

26

Ví dụ

- ❖ Tất cả thành phần cùng truy cập biến toàn cục, 1 số thiết lập giá trị, 1 số đọc giá trị
- ❖ OOP: biến static để là public
- ❖ Bất lợi khi debug
 - Biến toàn cục nhận giá trị khó hiểu
 - Xử lý đồng bộ
 - Khó theo dõi

27

27

Ví dụ

```
while( global_variable > 0 ) {
    switch( global_variable ) {
        case 1: function_a(); break;
        case 2: function_b(); break;
        ...
        case n:...
    }
    global_variable++;
}
```

28

28

2.3. Control Coupling (Móc nối điều khiển)

❖ Định nghĩa: Component truyền tham số điều khiển tới component khác

- Ví dụ, một method làm nhiều nhiệm vụ, tùy theo tham số flag truyền vào.

❖ Có thể tốt/xấu, tùy tình huống

- Xấu nếu component cần biết cấu trúc bên trong và cách xử lý của component kia.
- Tốt nếu tham số giúp ích cho việc factoring và tái sử dụng các chức năng

❖ Ví dụ tốt: hàm sắp xếp mảng, nhận tham số là hàm so sánh 2 phần tử.

Ví dụ 1

❖ Phương thức

- updateCustomer(int whatKind, Customer customer) trong đó
 - whatKind nhận giá trị ADD, EDIT hoặc DELETE, và
 - customer được dùng cho EDIT, nhưng không dùng gì trong ADD, chỉ trường id dùng cho DELETE.

Ví dụ 1 – giải pháp

❖ Cần tránh, vì module gọi phải biết logic của module được gọi

❖ Phương thức sẽ khó hiểu, khó bảo trì

❖ Để tránh: cần tạo các phương thức riêng cho từng chức năng. Cần áp dụng kế thừa, đa hình.

Ví dụ 2 – tight coupling

```
public void run() {  
    takeAction(1);  
}  
  
public void takeAction(int key) {  
    switch (key) {  
        case 1:  
            System.out.println("ONE RECEIVED");  
            break;  
        case 2:  
            System.out.println("TWO RECEIVED");  
            break;  
    }  
}
```


Ví dụ 2 – loose coupling

```
public void run() {
    Printable printable = new PrinterOne();
    takeAction(printable);
}
public void takeAction(Printable printable) {
    printable.print();
}

public interface Printable {
    void print();
}

public class PrinterOne implements Printable {
    @Override
    public void print() {
        System.out.println("ONE RECEIVED");
    }
}

public class PrinterTwo implements Printable {
    @Override
    public void print() {
        System.out.println("TWO RECEIVED");
    }
}
```

33

Ví dụ 3 – Vấn đề?

```
int testStack(int kind, Stack S) {
    if (((kind == 1) && (S.num == 0)) || ((kind == 2) && (S.num == MAX)))
        return 1;
    else
        return 0;
}
```

34

Ví dụ 3 – Giải pháp

❖ Nên có 2 hàm rõ ràng

```
int emptyStack(Stack S) {
    if (S.num == 0)
        return 1;
    else
        return 0;
}

int fullStack(Stack S) {
    if (S.num == MAXSTACK)
        return 1;
    else
        return 0;
}
```

35

2.4. Stamp Coupling (Móc nối dữ liệu phức hợp)

- ❖ Định nghĩa: Tham số dữ liệu được truyền trong cấu trúc dữ liệu có nhiều trường thông tin, nhưng chỉ dùng một số trường trong đó
- ❖ Có thể làm như vậy, nhưng cần có lý giải rõ ràng, thay vì lý do là "lười"
- ❖ Hai giải pháp: 1-dùng interface, 2-truyền tham số là dữ liệu nguyên thủy

36

Ví dụ 1

- ❖ Tính thuế phải trả của nhân viên

```
int incomeTaxPayable(Person p) {  
    /* code which refers to only p.salary */  
}
```

- ❖ Hàm incomeTaxPayable này làm LTV khác nghĩ nó dùng tất cả các trường đối tượng person → debug khó hiểu.

Ví dụ 1 – giải pháp

```
int incomeTaxPayable(int salary) {  
    /* code which refers salary */  
}
```

Ví dụ 2

```
public class Emailer {  
    public void sendEmail(Employee e, String text) {  
        ...  
    }  
    ...  
}
```

- ❖ Giải pháp Sử dụng dữ liệu nguyên thủy:

```
public class Emailer {  
    public void sendEmail(String name, String email, String text) {  
        ...  
    }  
    ...  
}
```

Ví dụ 2

- ❖ Giải pháp sử dụng interface

```
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) {  
        ...  
    }  
    ...  
}
```

2.5. Data coupling (Móc nối dữ liệu nguyên thủy)

❖ Định nghĩa: Module này gọi module kia, chỉ dùng tham số nguyên thủy thay vì đối tượng

- Càng nhiều tham số, tính coupling càng cao
- Nên bỏ bớt các tham số nếu được

41

Data coupling

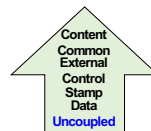
- ❖ Điểm mạnh
 - là loại coupling thấp nhất → tốt nhất.
 - 1 module chỉ được cung cấp các phần tử dữ liệu nó cần.
- ❖ Điểm yếu
 - module sẽ có nhiều tham số truyền vào
- ❖ Có mâu thuẫn (trade-off) giữa data coupling và stamp coupling

42

42

Uncoupled

❖ Các thành phần không liên quan tới nhau



43

43

Nội dung

1. Cách thức thiết kế
2. Móc nối linh hoạt (low coupling)
3. Tính kết dính cao (high cohesion)

44

44

3. Cohesion

- ❖ Định nghĩa 1: Mức độ cùng hướng tới mục tiêu chung giữa các phần tử trong 1 component
- ❖ Định nghĩa 2: Cohesion: Mỗi phần tử có 1 chức năng rõ ràng
- ❖ Là “chất keo” tạo nên 1 component
- ❖ Cao là tốt

Cohesion

- ❖ Mỗi module một nhiệm vụ, không nhiều hơn một, nhiệm vụ rõ ràng → high cohesion
- ❖ Một module nhiều nhiệm vụ → không có nhiệm vụ rõ ràng, khó đặt tên → low cohesion
- ❖ Nguyên lý **Single Responsibility Principle** (SOLID) hướng tới các lớp có tính cohesion cao
- ❖ Cohesion tang nếu
 - Các tính năng trong lớp cung cấp qua các phương thức có nhiều điểm chung going nhau
 - Mỗi phương thức thực thi các hoạt động liên quan tới nhau, thay vì các hoạt động, dữ liệu không liên quan

Lợi ích của tính kết dính cao

- ❖ Giảm độ phức tạp
- ❖ Module ít chức năng → đơn giản, dễ hiểu, dễ bảo trì, ít ảnh hưởng tới nhau
- ❖ Mỗi module 1 chức năng → Tăng tính tái sử dụng module.

Ví dụ

```
class A
checkEmail()
validateEmail()
sendEmail()
printLetter()
printAddress()
```

Fig: Low cohesion

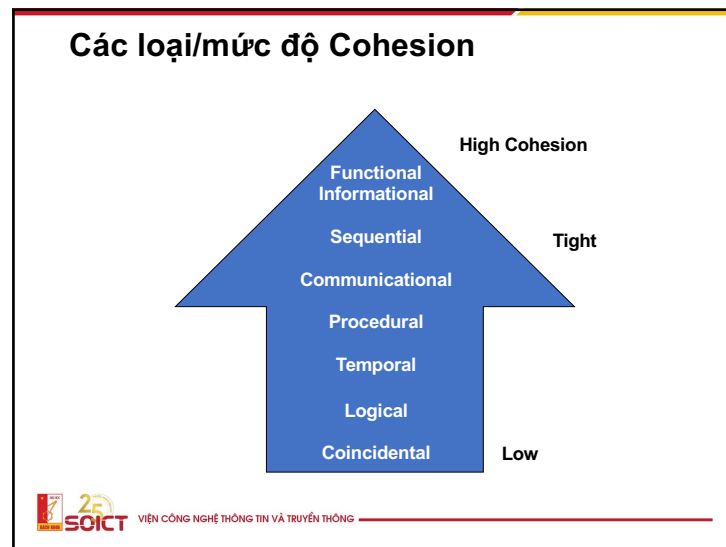
```
class A
checkEmail()
```

```
class B
validateEmail()
```

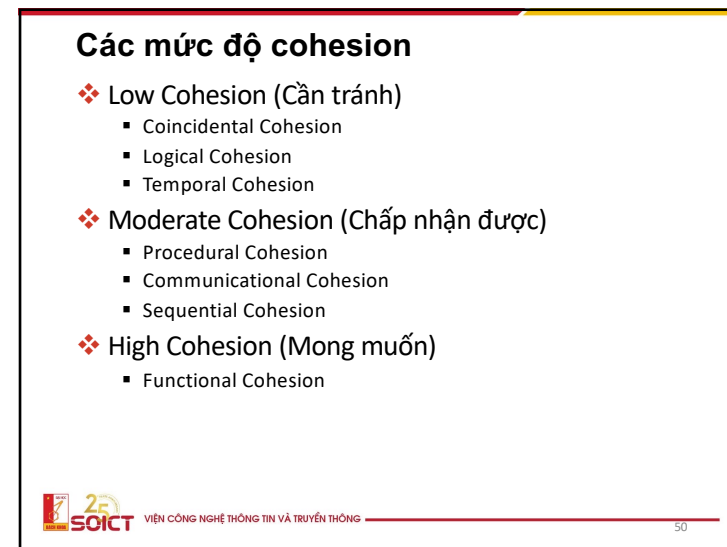
```
class C
sendEmail()
```

```
class D
printLetter()
```

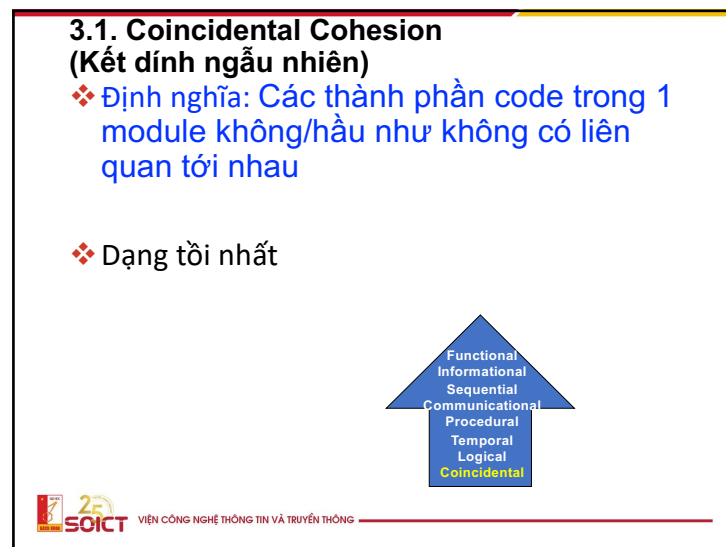
Fig: High cohesion



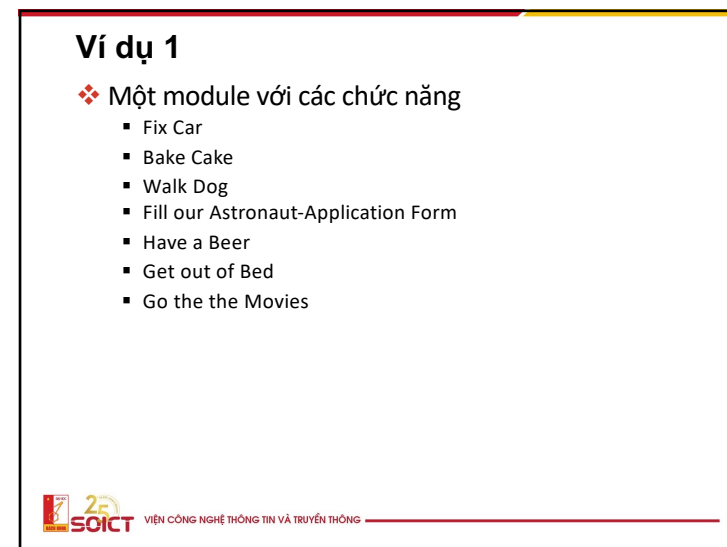
49



50



51



52

Ví dụ 2

```
class Joe {  
    // converts a path in windows to one in linux  
    public String win2lin(String);  
  
    // number of days since the beginning of time  
    public int days(String);  
  
    // outputs a financial report  
    public void outputReport(FinanceData);  
}
```

53

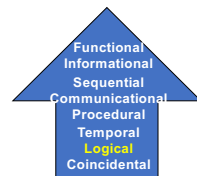
Ví dụ 3

```
class Output {  
    // outputs a financial report  
    public void outputReport(FinanceData);  
  
    // outputs the current weather  
    public void outputWeather(WeatherData);  
  
    // output a number in a nice formatted way  
    public void outputInt(int);  
}
```

54

3.2. Logical Cohesion (Kết dính logic)

- ❖ Định nghĩa: Các thành phần trong 1 component có liên quan tới nhau về mặt logic, không phải về mặt chức năng
- ❖ Phần tử sử dụng các component này sẽ lựa chọn một component nào đó để sử dụng.



55

Ví dụ 1

- ❖ lên kế hoạch đi du lịch, chọn 1 trong các hình thức sau đến đến điểm cần đến
 - Go by Car
 - Go by Train
 - Go by Boat
 - Go by Plane
- ❖ Các hoạt động này liên quan gì tới nhau mà đặt trong cùng 1 module? Đó là các hình thức đi lại. Nhưng khi đi du lịch, ta chỉ chọn 1 hình thức, không chọn nhiều

56

Ví dụ 2

- ❖ Hàm in báo cáo, tùy tham số flag truyền vào, hàm sẽ xử lý hình thức báo cáo tương ứng.
 - Printing a local sales report.
 - Printing a regional sales report.
 - Printing a national sales report.
- ❖ Ta chỉ chọn 1 trong các hình thức báo cáo này → không nên cài đặt tất cả các hình thức báo cáo trong hàm đó

57

Ví dụ 3

- ❖ Cần đọc dữ liệu vào, từ nhiều nguồn khác nhau.
- ❖ Cho tất cả code vào 1 component. Truyền vào tham số flag (0: tape, 1: disk, 2: network).
- ❖ Component ở đây có thể là method/class

```
public class Example {  
    public void readDataFromTape(Tape t) {  
    }  
    public void readDataFromDisk(Disk d) {  
    }  
    public void readDataFromNetwork(Network n) {  
    }  
}
```

58

58

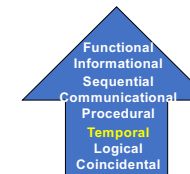
Ví dụ 3 – giải pháp

```
public abstract class DataReader<T> {  
    public void read(T t)  
}  
  
public class TapeDataReader extends DataReader<Tape> {  
    @Override  
    public void read(Tape t) {  
    }  
}  
  
public class DiskDataReader extends DataReader<Disk> {  
    @Override  
    public void read(Disk d) {  
    }  
}  
  
public class NetWorkDataReader extends DataReader<Network> {  
    @Override  
    public void read(Network n) {  
    }  
}
```

59

3.3. Temporal Cohesion (Kết dính hướng thời gian)

- ❖ Định nghĩa: Các phần tử trong một component liên kết tới nhau về mặt thời gian: cùng được thực hiện ở một thời điểm nào đó
- Khó sửa đổi, vì phức tạp. Đồng thời Khó tái sử dụng



60

Ví dụ 1

❖ Các hoạt động sau được đặt trong cùng 1 module, vì đều được thực hiện vào buổi tối

- Put out Milk Bottles
- Put out Cat
- Turn off TV
- Brush Teeth

61

Ví dụ 2

❖ Nhiều công việc khởi tạo các thành phần hệ thống được đưa hết vào lớp Init

```
class Init {  
    // initializes financial report  
    public void initReport(FinanceData);  
  
    // initializes current weather  
    public void initWeather(WeatherData);  
  
    // initializes master count  
    public void initCount();  
}
```

62

Ví dụ 2 – giải pháp

❖ Giải pháp: Dùng đa hình, interface/abstract class

❖ Lưu ý: Việc khởi tạo của từng lớp cần được tách độc lập với nhau, do lớp đó đảm nhận

Remember: The goal is to produce procedures which can do their single function independently of other procedures.

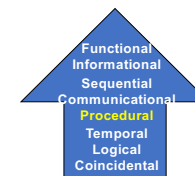
63

3.4. Procedural Cohesion (Kết dính thủ tục)

❖ Định nghĩa: Các thành phần được gom lại vì chúng cần được thực hiện theo thứ tự

❖ Vẫn thiếu tính kết dính và khó tái sử dụng

❖ Lưu ý: Không cần dữ liệu ra của bước này làm dữ liệu vào của bước sau (sequential cohesion)



64

Ví dụ 1

❖ A *Prepare for Holiday Meal* module:

- Clean Utensils from Previous Meal
- Prepare Turkey for Roasting
- Make Phone Call
- Take Shower
- Chop Vegetables
- Set Table

→ Cần tách các component thành các module khác nhau, đặt ở vị trí phù hợp. Không phải vì được thực hiện theo trình tự ở 1 xử lý nào đó mà đặt trong cùng 1 component.

Ví dụ 2

```
public class Example {  
    public void readData() {}  
    public void sendEmail() {}  
}
```



```
public class DataReader {  
    public void readData() {}  
}
```

```
public class Email {  
    public void sendEmail() {}  
}
```

Procedural cohesion

❖ Lợi ích

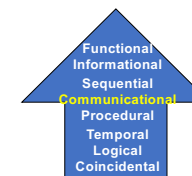
- Các phần tử liên quan tới nhau (về trình tự thực hiện) trong 1 được đặt cạnh nhau

❖ Tác hại

- Chưa tự nhiên. Các phần tử liên quan tới nhau vì trình tự thực hiện, chỉ trong procedure đó, thay vì thực sự có liên quan tới nhau
- Có thể gây ảnh hưởng lan truyền khi sửa 1 phần tử

3.5. Communicational Cohesion (Kết dính giao tiếp)

- ❖ Định nghĩa: Các thành phần con được gom lại trong 1 thành phần lớn, vì các thành phần con cùng thao tác trên một cấu trúc dữ liệu



Ví dụ

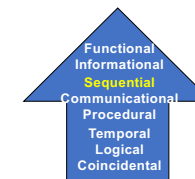
```
public class Example {  
    private Transaction trans;  
  
    public void readTransaction() {}  
    public void sortTransaction() {}  
    public void calculateTransactionMean() {}  
    public void printTransaction() {}  
    public void saveTransaction() {}  
}
```



69

3.6. Sequential Cohesion (Kết dính tuần tự)

- ❖ Tương tự như procedural. Khác biệt: bổ sung thêm yêu cầu input của phần tử này là output của phần tử trước đó
- ❖ Là loại cohesion tốt



70

Ví dụ

- ❖ Module với các hoạt động
 - Clean Car Body
 - Fill in Holes in Car
 - Sand Car Body
 - Apply Primer
- ❖ Tham số car sẽ được truyền từ tác vụ trước sang tác vụ sau



71

Sequential cohesion

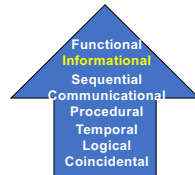
- ❖ Ví dụ: tính toán quỹ đạo và in ra kết quả. Cần phải làm theo thứ tự. Quỹ đạo tính toán được sẽ được in ra trong kết quả
- ❖ Lợi ích
 - tách biệt chức năng rõ ràng hơn so với loại communicational cohesion
- ❖ Bất lợi
 - hạn chế tái sử dụng. VD khi cần tính quỹ đạo rồi ghi kết quả ra file/truyền qua mạng, ..., sẽ phải tạo module mới. Module mới và module cũ bị lặp phần code tính toán quỹ đạo, chỉ khác biệt phần xử lý sau



72

3.7. Informational Cohesion (Kết dính thông tin)

- ❖ **Định nghĩa:** Cũng thao tác trên dữ liệu chung như communicational cohesion. Nhưng mỗi thao tác là độc lập với nhau.



Eví dụ 1

- ❖ 3 phương thức này hoạt động độc lập với nhau, không phụ thuộc vào nhau, cùng thao tác trên 1 CTDL EmpRec

```
void employeeAdd(int status, EmpRec rec);
```

```
void employeeUpdate(int status, EmpRec rec);
```

```
void employeeDelete(int status, EmpRec rec);
```

Ví dụ 2

```
public class Example {  
    public Transaction readTransaction() {}  
    public void sortTransaction(Transaction trans) {}  
    public void calculateTransactionMean(Transaction trans) {}  
    public void printTransaction(Transaction trans) {}  
    public void saveTransaction(Transaction trans) {}  
}
```

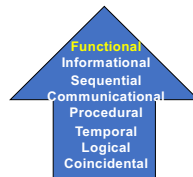
Informational Cohesion

- ❖ **Lợi ích**
 - đóng gói được các chức năng liên quan tới dữ liệu được cung cấp, giúp dễ hiểu, dễ đọc code, giảm chi phí bảo trì
- ❖ **Bất lợi**
 - Nếu ứng dụng nào đó không dùng tất cả các chức năng → lãng phí bộ nhớ

3.8. Functional Cohesion (Kết dính chức năng)

❖ Định nghĩa: Tất cả các thành phần trong 1 module cùng hướng đến 1 chức năng duy nhất mà module cung cấp

❖ Là loại lý tưởng nhất



3.8. Functional Cohesion

❖ Lợi ích

- tăng tính tái sử dụng, tách biệt chức năng rõ ràng, mỗi module 1 chức năng, giúp dễ hiểu, dễ sửa đổi, giảm chi phí bảo trì

❖ Bất lợi

- Tăng số lượng module

Khác biệt giữa Cohesion và Coupling

Cohesion

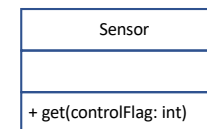
- ❖ Cohesion is the indication of the relationship within module
- ❖ Cohesion shows the module's relative functional strength
- ❖ Cohesion is a degree (quality) to which a component / module focuses on the single thing
- ❖ While designing we should strive for high cohesion. Ex: cohesive component/module focus on a single task with little interaction with other modules of the system
- ❖ Cohesion is the kind of natural extension of data hiding, for example, class having all members visible with a package having default visibility
- ❖ Cohesion is Intra-Module Concept

Coupling

- ❖ Coupling is the indication of the relationships between modules
- ❖ Coupling shows the relative independence among the modules
- ❖ Coupling is a degree to which a component / module is connected to the other modules
- ❖ While designing we should strive for low coupling. Ex: dependency between modules should be less
- ❖ Making private fields, private methods and non public classes provides loose coupling
- ❖ Coupling is Inter-Module Concept

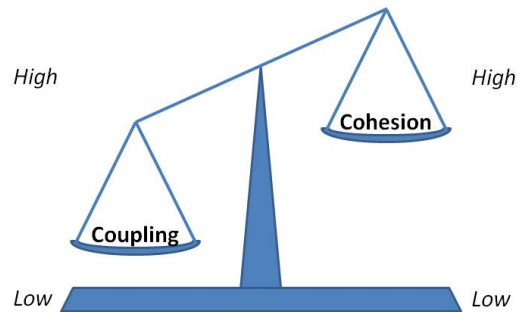
Ví dụ

Phân tích và Cải tiến code???



```
public void get (int controlFlag){
    switch (controlFlag) {
        case 0:
            return this.humidity;
            break;
        case 1:
            return this.temperature;
            break;
        default:
            throw new UnknownFlagException
    }
}
```

Quan hệ giữa Coupling và Cohesion?



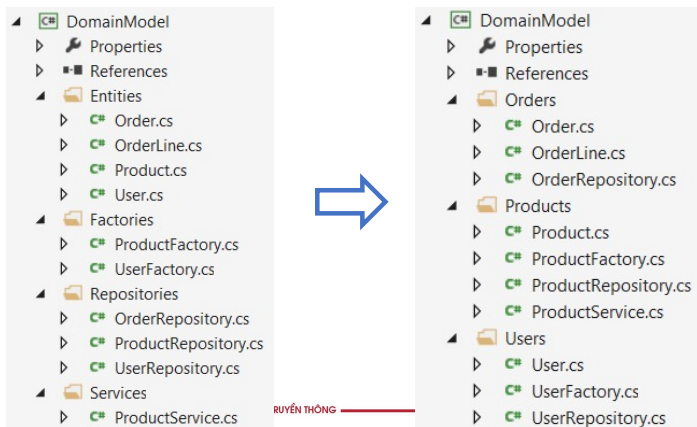
82

Cohesion và Coupling

- ❖ Thường là quan hệ tradeoff. Vì độ phức tạp sẽ hoặc đi vào 1 module, hoặc dàn trải ra giữa các module.
 - 1 module đơn giản sẽ có tính high cohesion. Nhưng khi đó, nó phải phụ thuộc vào nhiều module khác, phải phối hợp với nhiều module khác → tăng coupling
 - Kết nối giữa các module đơn giản để đảm bảo tính low coupling → module phải có nhiều nhiệm vụ hơn → làm giảm tính cohesion

83

Ví dụ



84