

Công nghệ phần mềm

Cài đặt và tích hợp

Nội dung chính

- Tổng quan
- Môi trường và Phương pháp cài đặt
- Phong cách lập trình
- Lập trình hướng hiệu quả
- Kiến trúc mô hình 1 lớp, 2 lớp, 3 lớp

Tổng quan

- **Cài đặt:** Là quá trình chuyển đổi từ **thiết kế** chi tiết sang **mã lệnh**.
- Triển khai (Implementation) # lập trình (programming)
- Triển khai = triển khai **thiết kế** chi tiết thành **chương trình**
 - Sử dụng lại mã nguồn/các thành phần
 - Lập trình nếu không/khó sử dụng lại
 - Tích hợp các thành phần.
- Mục tiêu: SẢN PHẨM PM TỐT

Tổng quan

Thế nào là ngôn ngữ lập trình tốt?

- Tập trung vào nhu cầu xác định dự án phát triển của từng phần mềm riêng.
- Có thể thiết lập được một tập hợp tổng quát các yêu cầu:
 - Dễ dịch thiết kế sang chương trình
 - Có trình biên dịch hiệu quả,
 - Khả chuyển chương trình gốc
 - Có sẵn công cụ phát triển
- Dễ bảo trì

Ngôn ngữ lập trình tốt (1)

- Dễ dịch thiết kế sang chương trình
 - Về lý thuyết, việc sinh chương trình gốc từ một đặc tả chi tiết nên là trực tiếp.
 - Cho phép một ngôn ngữ cài đặt trực tiếp cho các kết cấu có cấu trúc, các CTDL phức tạp, các vào/ra đặc biệt, khả năng thao tác bit và các đối tượng.
 - Làm cho việc dịch từ thiết kế sang chương trình gốc dễ hơn nhiều.

Ngôn ngữ lập trình tốt (2)

- Có trình biên dịch hiệu quả
 - Sự phát triển mạnh về phần cứng đã bắt đầu làm giảm nhẹ nhu cầu chương trình siêu hiệu quả, nhiều ứng dụng vẫn còn đòi hỏi các chương trình chạy nhanh, gọn (yêu cầu bộ nhớ thấp).
 - Các ngôn ngữ với trình biên dịch tối ưu có thể là hấp dẫn nếu hiệu năng PM là yêu cầu chủ chốt.

Khái niệm lập trình hiệu quả

- Sản phẩm PM tốt khi:
 - Phân tích tốt
 - Thiết kết tốt
 - Lập trình tốt
 - Kiểm thử chặt chẽ
- Kỹ thuật lập trình tốt:
 - Chuyên nghiệp (tuân theo các chuẩn)
 - Ổn định
 - Hiệu quả

Khái niệm lập trình hiệu quả

- Chương trình phải dễ hiểu
 - Tốc độ phát triển cao hơn
 - Năng lực biểu diễn cao hơn
 - Khả năng sử dụng lại cao hơn
 - Dễ bảo trì hơn
 - Dễ hiểu, dễ sửa đổi, dễ thích nghi
 - Chất lượng cao hơn
 - Sử dụng các cấu trúc an toàn hơn
- Lập trình hiệu quả hơn, sản phẩm rẻ tiền hơn**

Tốc độ viết mã nguồn

- Tốc độ phát triển cao # làm ngắn chương trình nguồn
 - Tốc độ không ~ số dòng lệnh
 - Câu lệnh phức tạp làm giảm độ dễ hiểu
- Ngôn ngữ mức cao (4GL)
 - Năng lực biểu diễn cao
 - Tốc độ phát triển nhanh

Tiến hóa của kỹ thuật lập trình

- Lập trình tuyến tính (tuần tự)
 - Khó sửa, dễ sinh lỗi
- Lập trình có cấu trúc (thủ tục)
 - Dễ hiểu hơn, an toàn hơn
- Lập trình hướng chức năng
 - Trao đổi DL bằng tham số
 - Loại bỏ hoàn toàn DL dùng chung
- OOP
 - Tái sử dụng, thuận tiện với các ứng dụng lớn
- Kỹ thuật lập trình thế hệ thứ 4
 - Mô tả tri thức

Lập trình tuyến tính

- Khi các PM còn *rất đơn giản*:
 - Chương trình được viết tuần tự với các câu lệnh thực hiện từ đầu đến cuối
- Ngày nay, khoa học máy tính ngày càng phát triển
 - Các PM đòi hỏi ngày càng phức tạp và lớn hơn rất nhiều.
 - Phương pháp lập trình tuyến tính kém hiệu quả.

Lập trình tuyến tính

- Không có/ thiếu các lệnh có cấu trúc
 - (for, while, do...while)
- Lạm dụng các lệnh **GOTO**
- Thiếu khả năng khai báo biến cục bộ

- 
- ☐ Độ ghép nối cao
 - ☐ Chương trình khó hiểu, khó sửa, dễ sinh lỗi

Các ngôn ngữ: thế hệ 1, 2: assembly, basic...

Lập trình hướng cấu trúc

- Chương trình được tổ chức thành các chương trình con, ra đời biến cục bộ
- Hệ thống chia các chức năng (hàm) thành các chức năng nhỏ hơn.
- Chương trình = CTDL + Giải thuật

 Dễ hiểu hơn, an toàn hơn


Các ngôn ngữ: thế hệ 2, 3: Fortran, Pascal, C...

Lập trình hướng chức năng

- Dựa trên nguyên tắc ghép nối DL
 - Trao đổi DL bằng tham số và giá trị trả về
 - Loại bỏ hoàn toàn DL dùng chung
- Loại bỏ các hiệu ứng phụ khi sửa đổi các module chương trình
- Nâng cao tính tái sử dụng

Ngôn ngữ: Lisp...

Lập trình hướng đối tượng

- Bao gói & che dấu thông tin
- Thao tác với DL thông qua các giao diện xác định
- Kế thừa
 -  ☐ Cục bộ hơn
 - ☐ Dễ tái sử dụng hơn
 - ☐ Thuận tiện cho các ứng dụng lớn

Các ngôn ngữ: C++, Java, C#...

Lập trình logic

- Tách tri thức về bài toán khỏi kỹ thuật lập trình
- Mô tả tri thức:
 - Các quy tắc
 - Các sự kiện
 - Mục tiêu
- Hệ thống tự chứng minh
 - Tìm đường đi đến mục tiêu

Ngôn ngữ: Prolog...

Phương pháp cài đặt

- Cài đặt từ dưới lên
 - Lần lượt cài đặt tất cả module từ dưới lên
 - Thao tác kiểm định được thực hiện sau khi hoàn tất việc cài đặt tất cả các module
- Cài đặt từ trên xuống và tăng trưởng
 - Cài đặt từng module theo thứ tự từ trên xuống
 - Cài đặt đến đâu, kiểm định đến đó
 - Sử dụng các module giả lập để thay vào các module chưa cài đặt
 - Hệ thống tăng trưởng dần qua các lần lặp cho đến khi trở thành hệ thống hoàn chỉnh

Ưu điểm của phương pháp cài đặt từ trên xuống

- **Giảm thiểu được các thay đổi** có thể dẫn đến đổ vỡ toàn bộ hệ thống.
- Cho phép thực hiện nhiều **phương án trung gian** một cách mềm dẻo để có được hệ thống mới.
- **Sớm nhận được các thông tin phản hồi** để điều chỉnh các phương án vì người cài đặt và người dùng thấy được sự hoạt động của hệ thống sớm.
- Phần lớn các giao diện được cài đặt và kiểm định sớm, tạo ra ngay được bộ mặt của hệ thống, do đó có thể **thẩm định tính dùng được của sản phẩm sớm**.


Môi trường cài đặt

- Tùy **kích cỡ** và **đặc trưng xử lý** của hệ thống mà chọn lựa một môi trường phần cứng và phần mềm thích hợp.


Lựa chọn ngôn ngữ lập trình

- Đặc trưng của ngôn ngữ
 - Năng lực (kiểu biến, các cấu trúc)
 - Tính khả chuyển
 - Mức độ hỗ trợ của các công cụ
- Miền ứng dụng của ngôn ngữ
 - Lập trình hệ thống
 - Nghiệp vụ, kinh doanh
 - Khoa học kỹ thuật
 - Trí tuệ nhân tạo
- Năng lực, kinh nghiệm của nhóm phát triển
- Yêu cầu của KH

Lựa chọn ngôn ngữ lập trình

- Ngôn ngữ bậc cao:
 - Có cấu trúc, câu lệnh phong phú
 - Hỗ trợ nhiều kiểu DL
 - Hỗ trợ con trỏ, đệ quy
 - Hỗ trợ HĐT
 - Thư viện đa dạng
-  Nên dùng ngôn ngữ bậc cao (hơn)

Lựa chọn ngôn ngữ lập trình

- Tính khả chuyển là yếu tố quan trọng của ngôn ngữ, cần khi
 - Thay đổi OS
 - Thay đổi phần cứng
 - Java khả chuyển
 - Các ngôn ngữ thông dịch (script) khả chuyển)
- 
- Sử dụng các tính năng chuẩn của ngôn ngữ
 - Sử dụng script khi có thể

Lựa chọn ngôn ngữ lập trình

- Có công cụ hiệu quả
 - Trình biên dịch hiệu quả
 - Biên dịch tốc độ cao
 - Khả năng tối ưu cao
 - Khai thác các tập lệnh, kiến trúc phần cứng mới
 - Có công cụ trợ giúp hiệu quả
 - Editor, debugger, linker, maker...
 - IDE (Integrated Develop Environment)
 - Môi trường UNIX thường không dùng IDE

Lựa chọn ngôn ngữ lập trình

- Miền ứng dụng và ngôn ngữ
 - Phần mềm hệ thống
 - Hiệu quả, vận năng, dễ mở rộng
 - C/C++
 - Hệ thời gian thực: C, C++, Ada, Assembly
 - PM nhúng: C++, Java
 - PM khoa học kỹ thuật
 - Tính toán chính xác, thư viện toán học mạnh, dễ dàng song song hóa
 - Fortran, Matlab, Maple

Lựa chọn ngôn ngữ lập trình

- Miền ứng dụng và ngôn ngữ
 - PM nghiệp vụ:
 - CSDL: Oracle, DB2, SQL server, MySQL...
 - Ngôn ngữ: FoxPro, Cobol, VB, C++...
 - Trí tuệ nhân tạo:
 - Lisp, Prolog, C++, OPS5...
 - Lập trình web/CGI:
 - Perl, ASP, PHP, Java, Python, Java script...

Nội dung chính

- Tổng quan
- Môi trường và Phương pháp cài đặt
- **Phong cách lập trình**
- Lập trình hướng hiệu quả
- Kiến trúc mô hình 1 lớp, 2 lớp, 3 lớp

Phong cách lập trình

- Bao gồm các yếu tố:
 - Cách đặt tên biến, hàm
 - Cách xây dựng câu lệnh, cấu trúc chương trình
 - Cách viết chú thích
- Hướng tới phong cách làm cho mã nguồn:
 - Dễ hiểu, dễ sửa lỗi
 - An toàn (ít lỗi)

Người khác có thể hiểu được, bảo trì được

Tại sao cần dễ hiểu?

- Phần mềm luôn cần sửa đổi
 - Sửa lỗi
 - Nâng cấp
- ➔ Kéo dài tuổi thọ, nâng cao hiệu quả kinh tế
- Nếu không dễ hiểu
 - Bảo trì tốn thời gian, chi phí cao
 - Tác giả phải bảo trì suốt vòng đời của PM
 - Bản thân tác giả cũng không hiểu

Chú thích

- Mọi điều được chú thích trong chương trình:
 - Mục đích sử dụng của các biến
 - Chức năng của khối lệnh, câu lệnh
 - Các lệnh điều khiển
 - Các lệnh phức tạp
 - Chú thích các module
 - Mục đích, chức năng của module
 - Tham số, giá trị trả về (giao diện)
 - Các module thuộc cấp
 - Cấu trúc, thuật toán
 - Nhiệm vụ của các biến cục bộ
 - Tác giả, người kiểm tra, thời gian

Đặt tên

- Đặt tên biến, tên hàm có nghĩa, gợi nhớ
- Sử dụng các ký hiệu, từ Tiếng Anh có nghĩa
- Làm cho dễ đọc
 - Dùng DateOfBirth/ date_of_birth/Date_Of_Birth
 - Không viết dateofbirth
- Tránh đặt tên quá dài
- Thống nhất cách dùng
 - Tên lớp bắt đầu bằng chữ hoa, tên hằng toàn chữ hoa
 - Tên biến bắt đầu bằng chữ thường
 - i cho biến vòng lặp, tmp cho các giá trị tạm

Câu lệnh

- Các câu lệnh phải mô tả cấu trúc
 - Thụt lề, dễ đọc, dễ hiểu
- Làm đơn giản các lệnh
 - Mỗi lệnh trên 1 dòng
 - Triển khai các biểu thức phức tạp
 - Hạn chế truyền tham số là kết quả của hàm, biểu thức
- Tránh các cấu trúc phức tạp
 - Các lệnh if lồng nhau
 - Điều kiện phủ định

Hàm và biến cục bộ

- Chương trình cần được chia thành nhiều module
- Không viết hàm quá dài
 - Không quá 2 trang màn hình
 - Tạo ra các hàm thứ cấp để giảm độ dài từng hàm
- Không dùng quá nhiều biến cục bộ
 - Không thể theo dõi đồng thời hoạt động của nhiều biến
 - Ví dụ: 7 biến

Xử lý lỗi

- Có thể phát hiện lỗi trong khi thực hiện
 - Lỗi chia 0
 - Lỗi input/output
- Xử lý lỗi
 - Nhất quán trong xử lý: phân loại lỗi; thống nhất định dạng thông báo
 - Phân biệt output và thông báo lỗi
 - Các hàm thư viện nên tránh việc tự xử lý, tự đưa ra thông báo lỗi

Output và thông báo lỗi

- Output là DL, còn được dùng làm input cho PM khác
- Thông báo lỗi là các thông tin nhất thời, trạng thái hệ thống, lỗi và cách khắc phục
- Cần tách output và thông báo lỗi
- OS thường cung cấp 3 dòng DL chuẩn
 - Stdin (cin)
 - Stdout (cout)
 - Stderr (cerr)

Ngoại lệ

- Là cách thức xử lý lỗi tiên tiến trong các OOL
 - Module xử lý ném ra 1 ngoại lệ (đối tượng chứa thông tin lỗi)
 - Module điều khiển bắt ngoại lệ (nếu có)
- Tách phần xử lý lỗi khỏi phần cài đặt thuật toán thông thường, làm cho chương trình dễ đọc hơn
- Dễ dùng hơn, an toàn hơn

Ném ngoại lệ

```
double MyDivide(double num, double denom)
{
    if (denom == 0.0) {
        throw invalid_argument("The denom cannot be 0.");
    }
    else {
        return num / denom;
    }
}
```

Bắt ngoại lệ

```
try {  
    result = MyDivide(x, y);  
}  
catch (invalid_argument& e) {  
    cerr << e.what() << endl;  
    ...    // mã xử lý với ngoại lệ  
};
```

Giao diện của module

- Thống nhất định dạng
 - Thứ tự truyền tham số
 - Strcpy(des, src)
- Kiểm tra tính hợp lệ của DL
 - Chỉ thực hiện xử lý với DL hợp lệ
- Làm đơn giản giao diện (giảm mức ghép nối)
 - Không truyền thừa tham số
 - Tìm cách kết hợp các khoản mục liên quan

Phong cách lập trình tốt

- Tuân theo các chuẩn thông dụng
- Chuẩn được chấp nhận rộng rãi hơn dễ hiểu hơn
- Chú thích đầy đủ mỗi khi không tuân theo chuẩn

“Người khác có hiểu được không?”

Nội dung chính

- Tổng quan
- Môi trường và Phương pháp cài đặt
- Phong cách lập trình
- Lập trình hướng hiệu quả
- Kiến trúc mô hình 1 lớp, 2 lớp, 3 lớp

Kỹ thuật lập trình tránh lỗi

Dựa trên các yếu tố

- Lập trình có cấu trúc
 - Dùng các lệnh có cấu trúc
 - Module hóa
 - Hạn chế dùng các cấu trúc nguy hiểm
- Đóng gói/ che dấu thông tin
 - Xây dựng kiểu DL trừu tượng
 - Hạn chế thao tác trực tiếp lên thuộc tính

Tránh các cấu trúc nguy hiểm

- Số thực
 - Các phép toán được làm tròn, kết quả không chính xác tuyệt đối
 - So sánh ($=$) hai số thực là không khả thi
 - Phép chia lấy dư ($\%$) là không khả thi
- Con trỏ
 - Khái niệm mức thấp
 - Có khả năng gây lỗi nghiêm trọng
 - Dễ nhầm

Tránh các cấu trúc nguy hiểm

- Cấp phát bộ nhớ động
 - Quên cấp phát
 - Quên giải phóng
- Độ quy
 - Dễ nhầm điều kiện dừng

Lập trình phòng thủ

Defensive programming

- Nhiều lệnh có khả năng sinh lỗi
 - Lệnh vào/ra
 - Các phép toán
 - Thao tác với bộ nhớ
 - Truyền tham số sai kiểu
- Dự đoán khả năng xuất hiện lỗi
- Khắc phục lỗi
 - Lưu trạng thái an toàn
 - Quay lại trạng thái an toàn gần nhất

Lập trình phòng thủ

Defensive programming

- Lệnh vào/ra
 - DL không hợp lệ
 - Tràn bộ đệm (kiểu ký tự)
 - Lỗi thao tác file (sai tên, chưa được mở)
- Các phép toán
 - Lỗi chia cho 0
 - Tràn số
 - So sánh số thực

Lập trình phòng thủ

Defensive programming

- Thao tác bộ nhớ
 - Quên cấp phát, quên giải phóng bộ nhớ
 - Thiếu bộ nhớ
 - Sai địa chỉ, tràn bộ nhớ

Lập trình phòng thủ

Defensive programming

```
FILE *in;  
in = fopen("dulieu.txt", "r");
```



```
FILE *in;  
if ((in = fopen("dulieu.txt", "r")) == NULL) {  
    fprintf(stderr, "Không thể mở tệp dulieu.txt");  
    perror("\n Lý do: ");  
    exit(1);  
}
```

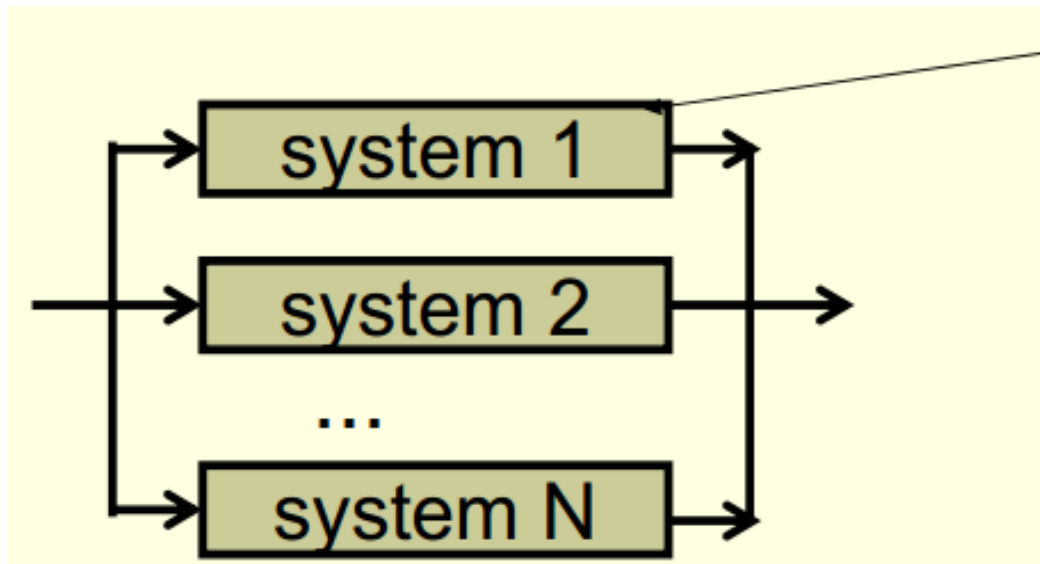
Lập trình thứ lỗi

Fault tolerant programming

- Không thể loại trừ hoàn toàn lỗi
- Cần có các hệ thống có độ tin cậy đặc biệt
- Dung thứ lỗi: chấp nhận sự xuất hiện lỗi lập trình
- Phát hiện, khắc phục lỗi
- Khởi nguyên từ thứ lỗi phần cứng
- Dựa trên nguyên tắc song song hóa chức năng.

Nguyên lý thứ lỗi

- Song song hóa thiết bị



xác suất có lỗi $\alpha < 1$

xác suất cả hệ thống
ngừng hoạt động
 α^N

Thứ lỗi DL

- Phục hồi lùi
 - Kiểm tra tính hợp lệ của DL mỗi khi biến đổi
 - Chỉ chấp nhận các DL hợp lệ
- Phục hồi tiến
 - Dùng DL dư thừa
 - Kiểm tra và khôi phục DL

Hiệu quả thực hiện

- PM ngày càng phức tạp, đa dạng
 - Mô phỏng
 - Ứng dụng thời gian thực
 - PM nhúng
 - Trò chơi
- Hiệu quả thực hiện luôn cần được xem xét
 - Thuật toán hiệu quả
 - Kỹ thuật lập trình hiệu quả
 - Ngôn ngữ lập trình hiệu quả

Cải thiện tốc độ

- Tối ưu hóa chu trình
 - Đưa phép toán bất biến ra ngoài chu trình
 - Tính sẵn giá trị được sử dụng nhiều lần
- Tránh dùng mảng nhiều chiều
- Tránh dùng biến con trỏ
- Dùng các KDL đơn giản
 - double → float

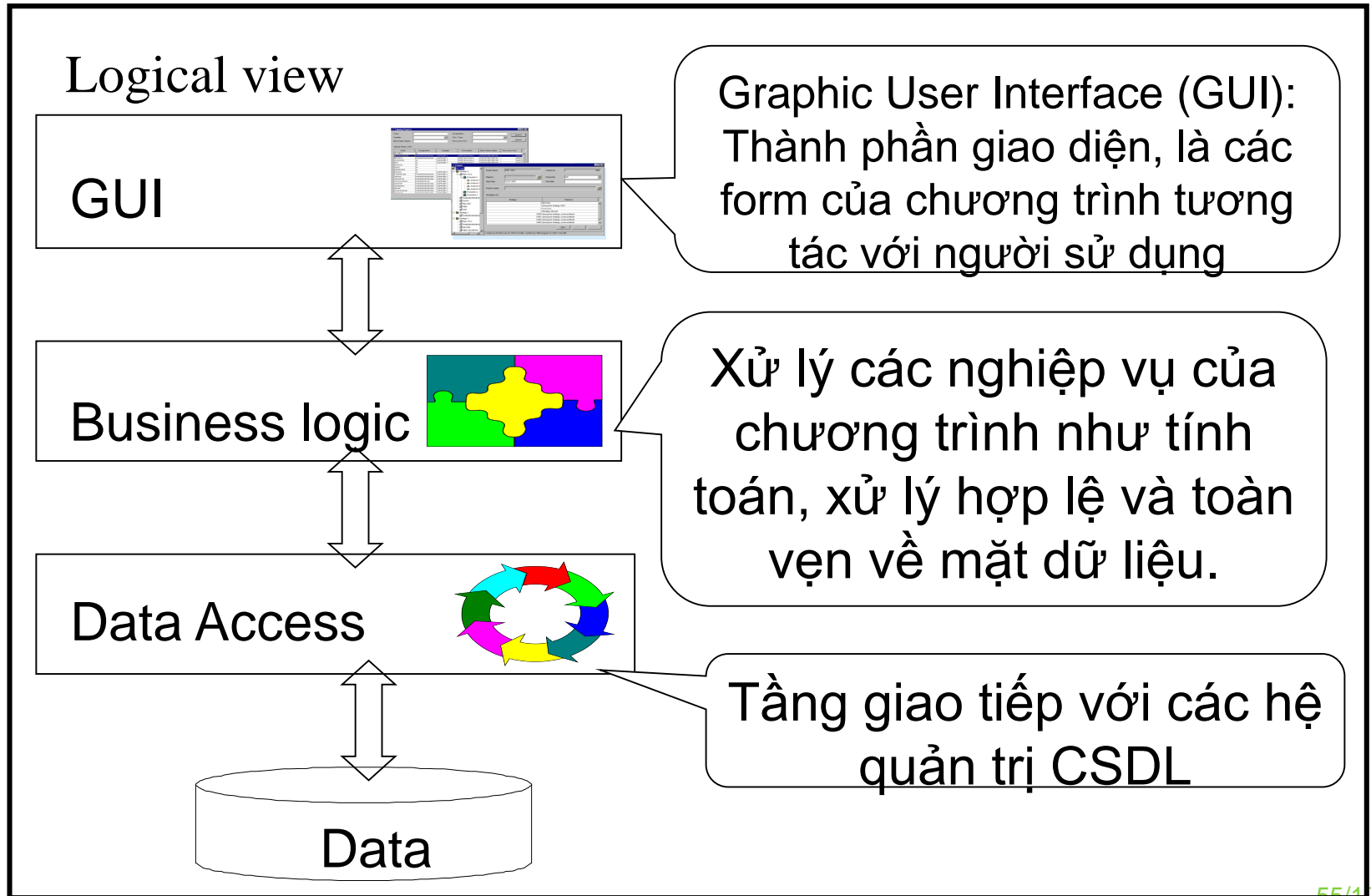
Nội dung chính

- Tổng quan
- Môi trường và Phương pháp cài đặt
- Phong cách lập trình
- Lập trình hướng hiệu quả
- Kiến trúc mô hình 1 lớp, 2 lớp, 3 lớp

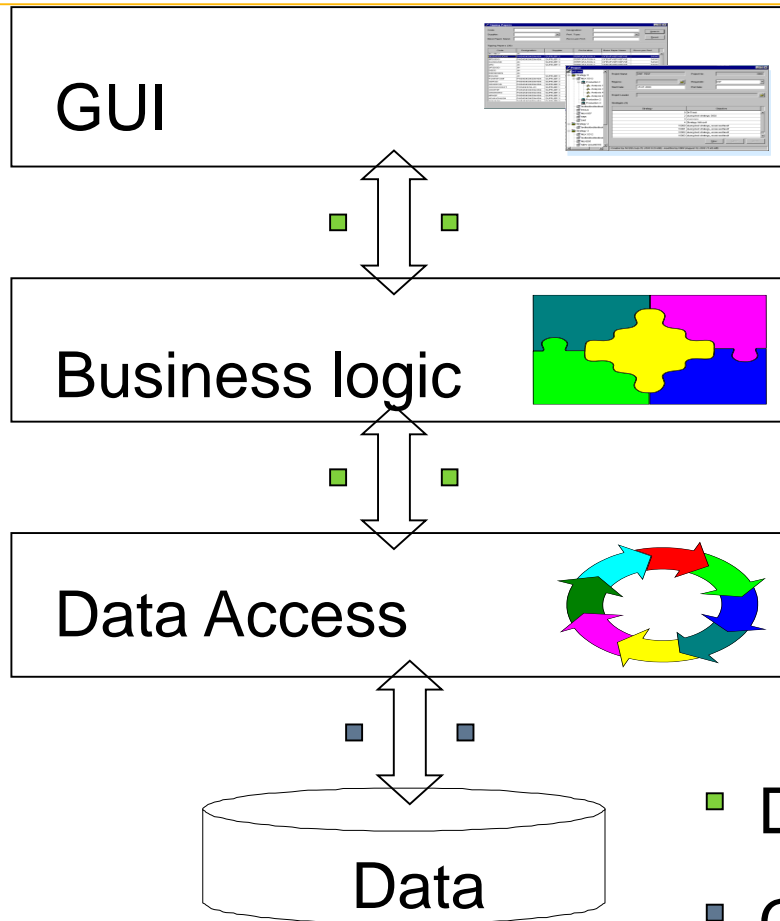
Khái niệm mô hình 3-layer.

- Khái niệm của mô hình 3-layer.
 - 3-Layers có tính logic (mỗi layer có 1 công việc) và là 1 thành phần của 3-Tiers. Gồm 3 lớp chính.

Khái niệm của mô hình 3-layer.

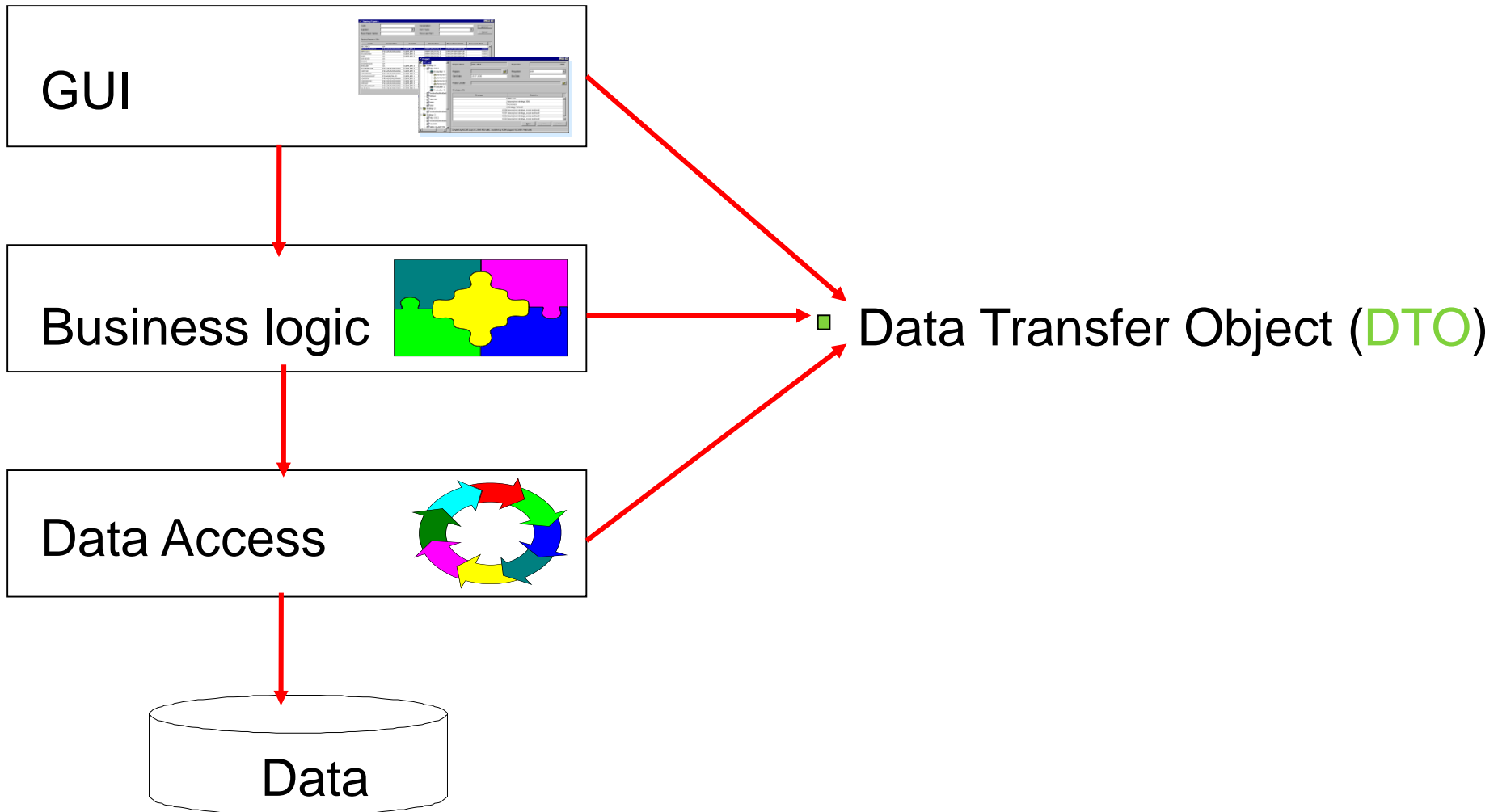


Việc trao đổi liên lạc giữa các layer



- Data Transfer Object (**DTO**)
- Các giá trị, dòng, bảng

Sự phụ thuộc giữa các layer



Tính chất của mô hình 3-layer

- Giảm sự ghép nối giữa các thực thể PM
- Tái sử dụng
- Chia sẻ trách nhiệm

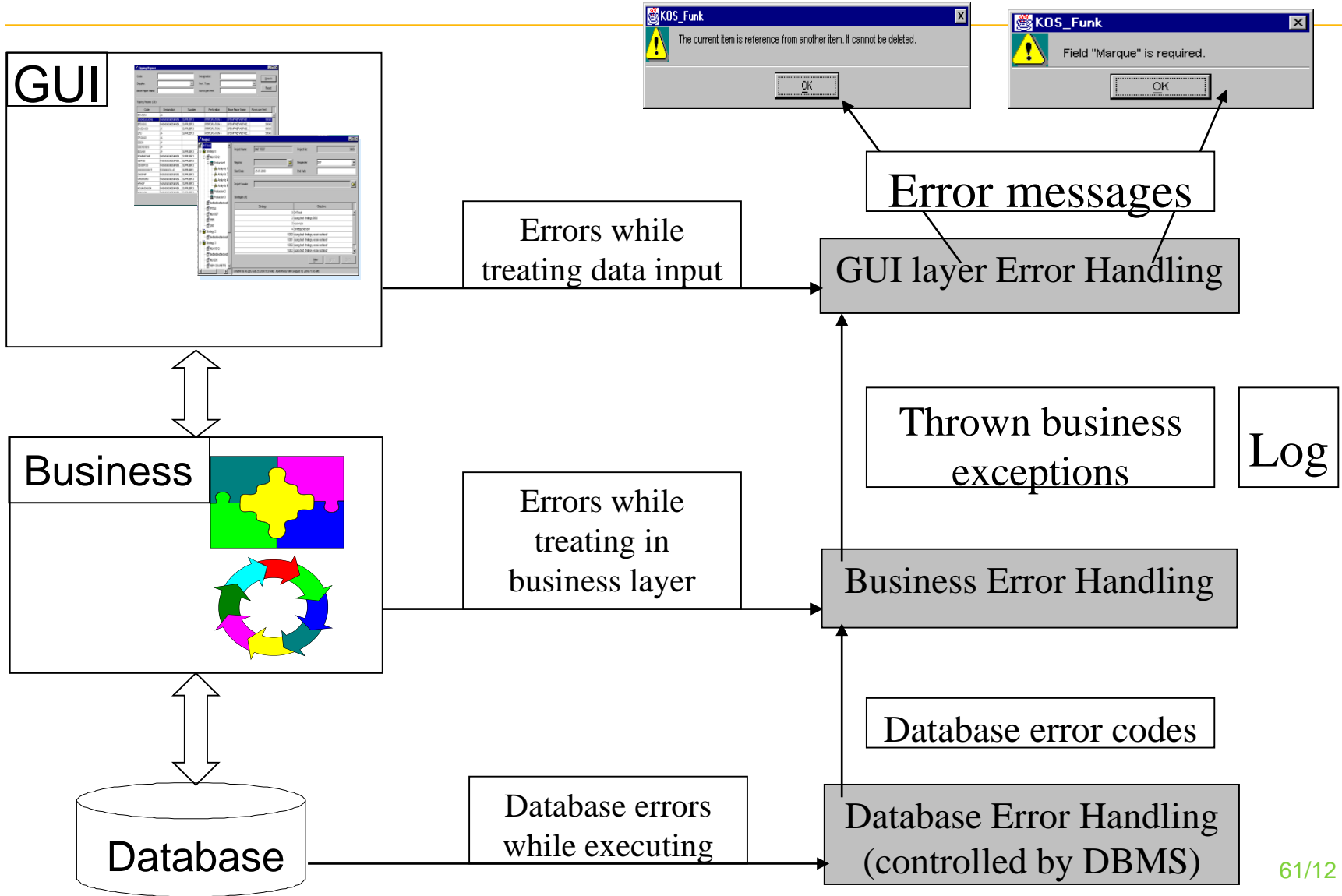
Vai trò của các layer

- **GUI (Presentation) Layer:** Nhập liệu và trình bày dữ liệu, có thể bao gồm các bước kiểm tra dữ liệu trước khi gọi Business Logic Layer.
- **Business Logic Layer:** Kiểm tra các yêu cầu nghiệp vụ trước khi cập nhật dữ liệu, quản lý các **transaction**, quản lý các **concurrent access**.
- **Data Access Layer:** Kết nối CSDL, tìm kiếm, thêm, xóa, sửa,...trên CSDL

Các lưu ý quan trọng

- Phân biệt vai trò Business Layer và khái niệm “xử lý”
- Mỗi Layer vẫn có xử lý riêng, đặc trưng của Layer đó
- Đôi khi việc quyết định 1 xử lý nằm ở layer nào chỉ mang tính chất tương đối

Quản lý ngoại lệ



Quản lý ngoại lệ

- Ngoại lệ có thể xảy ra ở bất kỳ layer nào
- Khi ngoại lệ xảy ra ở một layer thì:
 - Xử lý nội bộ trong layer đó
 - “Ném” ngoại lệ lên layer “cao hơn”
 - Không xử lý
- Khi một layer nhận ngoại lệ từ một layer “thấp hơn”
 - Xử lý nội bộ
 - “Ném” ngoại lệ lên layer “cao hơn”
 - Không xử lý

Ưu điểm của mô hình 3-Layer

- Khi ứng dụng đòi hỏi có sự tách biệt 3 phần: giao diện, xử lý nghiệp vụ, giao tiếp với hệ quản trị CSDL để người viết có thể dễ dàng quản lý ứng dụng của mình khi có lỗi xảy ra với 1 số thành phần xử lý nghiệp vụ không mong muốn.
- Tạo ra 1 không gian làm việc rất tốt để người thiết kế giao diện, lẫn người lập trình có thể làm việc chung với nhau 1 cách dễ dàng.

Ưu điểm của mô hình 3-Layer

- Khả năng tái tạo cao: Khi ứng dụng bắt chợt yêu cầu thay đổi hệ quản trị CSDL hoặc chuyển ứng dụng từ window application sang web application, việc xây dựng lại ứng dụng từ đầu rất tốn nhiều thời gian và chi phí. Vì vậy với mô hình này ra đời sẽ giải quyết vấn đề này

Ưu điểm của mô hình 3-Layer

- Các thao tác trên control như: kiểm tra nhập hợp lệ, ẩn hiện các control, và các xử lý thông tin trên control thì ta đặt các hàm xử lý ngay trên GUI.
- Các thao tác trên các dữ liệu cơ bản như: List, Array List, Object, DataTable, string, int, long, float,... ta xử lý ngay chính tầng nghiệp vụ BUS, vì bản chất khi thay đổi hệ quản trị hay các platform thì BUS không thay đổi.

Ưu điểm của mô hình 3-Layer

- Các thao tác với CSDL như truy vấn, kết nối, đóng kết nối,... ta xử lý trong DAL.
- Khi có nhu cầu thay đổi hệ quản trị CSDL, ta chỉ cần thay đổi DAL phù hợp với hệ quản trị mới, giữ nguyên BUS, GUI và build lại project.
- Khi có nhu cầu chuyển đổi qua lại giữa ứng dụng web forms hoặc win forms ta chỉ cần thay GUI, giữ nguyên DAL, BUS và build lại project

1-tier, 3-layer

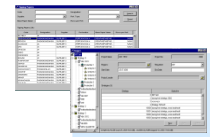
Physical view

Application +



Logical view

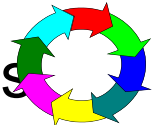
GUI



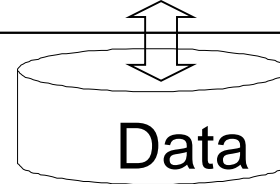
Business logic



Data Access

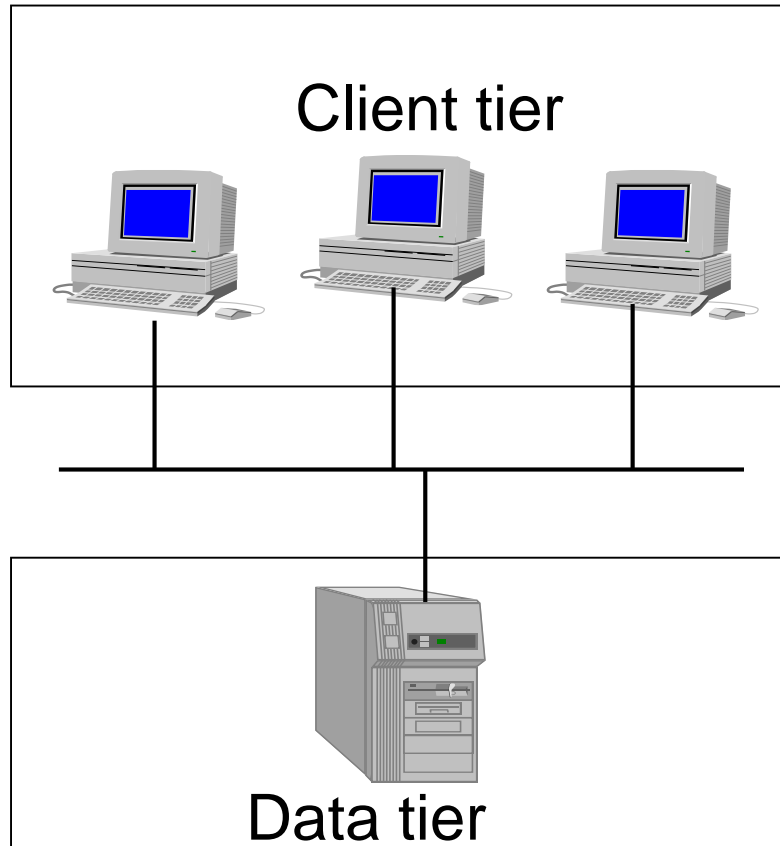


Data

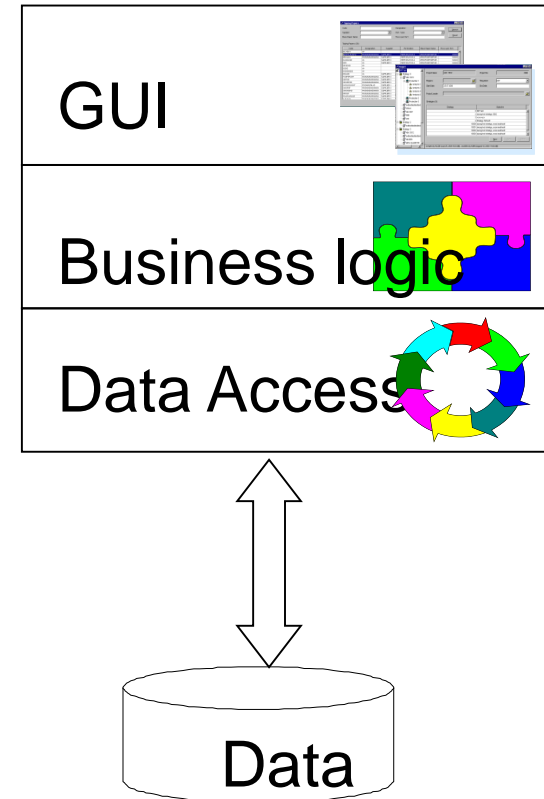


2-tier, 3-layer

Physical view



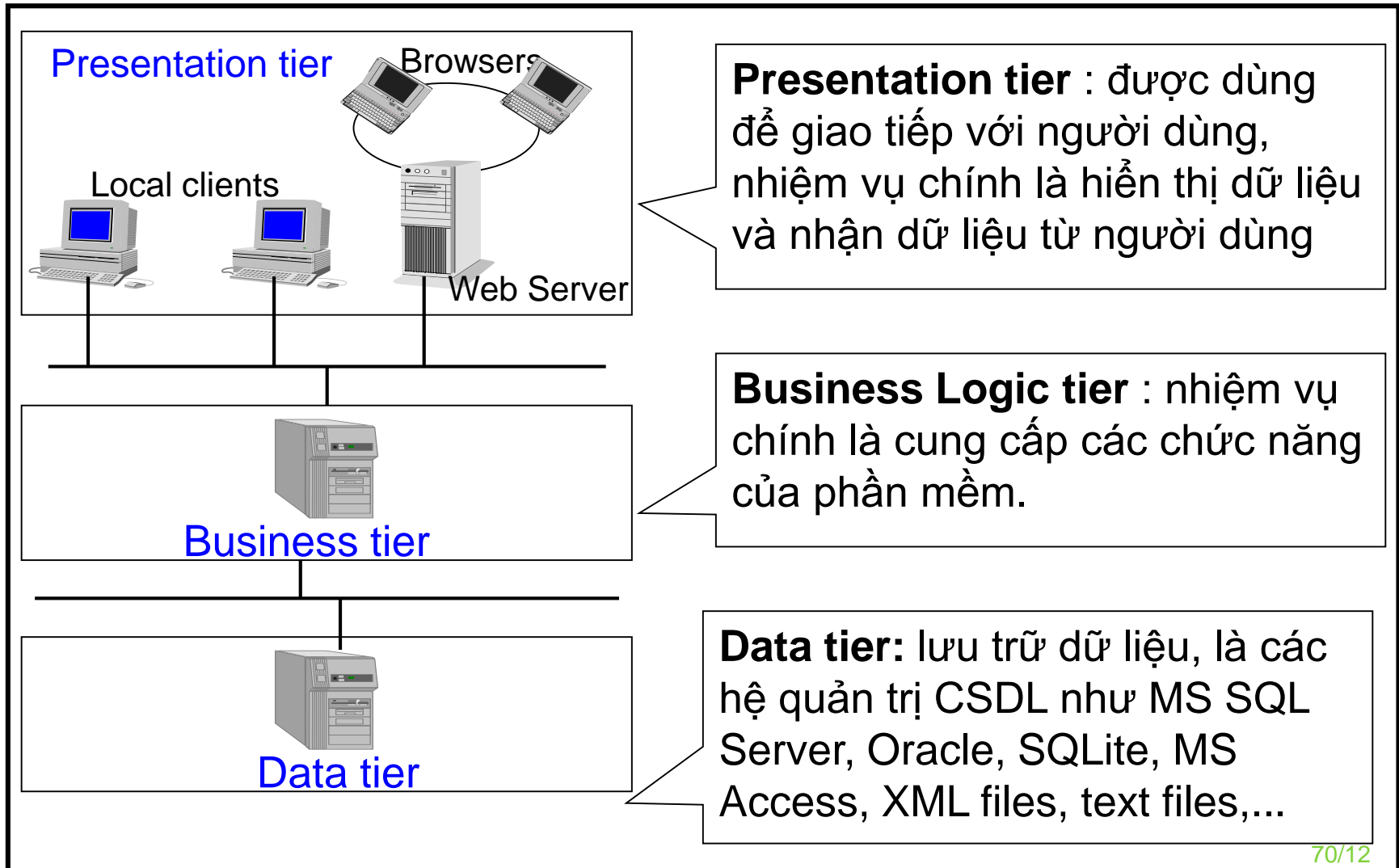
Logical view



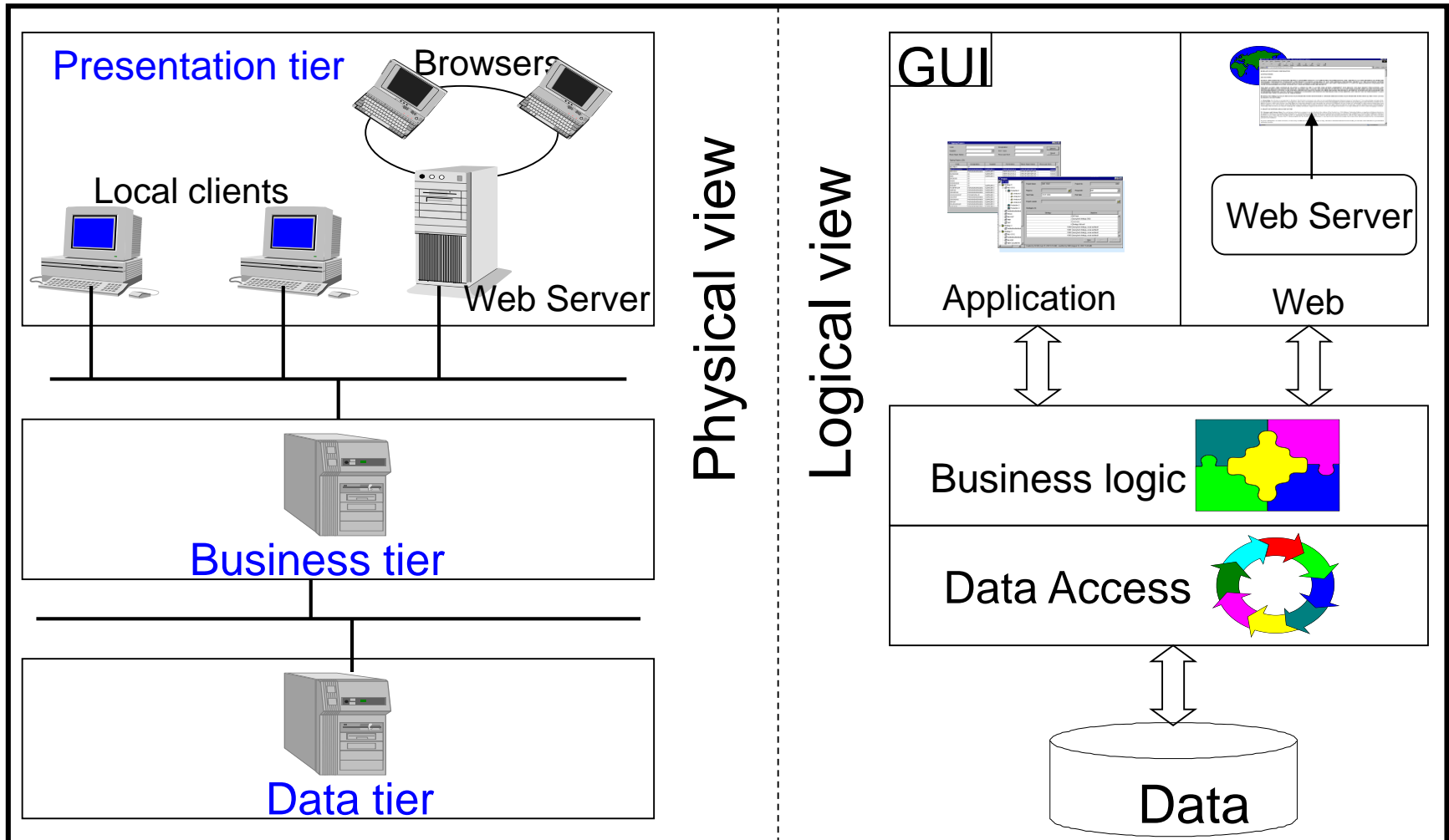
Khái niệm mô hình 3-Tiers

- 3-Tiers có tính vật lý (physical): là mô hình client-server (mỗi tier có thể đặt chung 1 nơi hoặc nhiều nơi, kết nối với nhau qua Web services, WCF, Remoting...). Như hình vẽ ta thấy 3 tầng rõ rệt 3 tầng:

Khái niệm mô hình 3-Tiers



3-tier, 3-layer



Các ưu và nhược điểm của 3-tiers

- Ưu điểm:
 - Dễ dàng mở rộng, thay đổi quy mô của hệ thống: Khi cần tải lớn, người quản trị có thể dễ dàng thêm các máy chủ vào nhóm, hoặc lấy bớt ra trong trường hợp ngược lại.
- Nhược điểm:
 - Việc truyền dữ liệu giữa các tầng sẽ chậm hơn vì phải truyền giữa các tiến trình khác nhau, dữ liệu cần phải được đóng gói -> truyền đi -> mở gói trước khi có thể dùng được.

Bài tập

1. Các kỹ thuật lập trình đã có? Đặc trưng của mỗi loại?
2. Tiêu chuẩn lựa chọn ngôn ngữ lập trình?
3. Các miền ứng dụng và các ngôn ngữ phù hợp?
4. Ưu nhược điểm của mô hình 3 lớp