**CHƯƠNG 4**

**1Design in Construction**

**2 Working Classes**

**3 High-Quality Routines**

**Software Construction**

**4 High Quality Code**

**5 The Pseudocode Programming Process**

1. **Thiết kế trong xdpm**

* Là giai đoạn sau đặc tả yêu cầu, là bước đầu chuyển biến từ đặc tả sang sản phẩm.
* Thiết kế phần mềm: sự kết hợp của quan niệm, phát minh để biến một đặc điểm kỹ thuật cho phần mềm máy tính vào phần mềm hoạt động.
* **5 vấn đề thử thách của thiết kế:**
* Design Is a Wicked Problem: là vấn đề xảy ra rồi người ta mới biết là vấn đề gì; không phải thiết kế nào cũng là wicked nhưng người lập trình phải đối mặt với wicked
* **Design Is a Sloppy Process:** quá trình trong thiết kế là quá trình không rõ ràng.
* **Design Is About Tradeoffs and Priorities (cân đối và ưu tiên):** làm việc phải sắp xếp độ ưu tiên trước, cân đối mục tiêu và các điều kiện để thực hiện mục tiêu, cân đối lại các yêu cầu, tập trung yêu cầu nào, giảm bớt yêu cầu nào. Ví dụ như yêu cầu tốn nhiều chi phí thì phải ưu tiên hoặc yêu cầu có hạn thì ưu tiên trc.
* Design Involves Restrictions: các yêu cầu phi thực tế mâu thuẫn với các chức năng khác -> xem xét có nhất thiết thực hiện mâu thuẫn không. Nếu mâu thuẫn thì hạn chế nó.
* Design Is Nondeterministic: kết quả thiết kế là không dự đoán đc vì 1 yêu cầu đưa 3 người 3 bản thiết kế khác.
* **Các khái niệm thiết kế chính trong xdpm(11):**
* **Minimal complexity (Độ phức tạp tối thiểu):** khi thiết kế càng đơn giản càng tốt
* **Ease of maintenance** **(Dễ bảo trì):** các yếu tố hỗ trợ cho nhau, khi đạt dc 1 yếu tố thì yếu tố khác dễ dàng đạt được
* ***Loose coupling: các thành phần trong hệ thống phải có mối quan hệ với nhau, khi điều chỉnh 1 thành phần sẽ ảnh hưởng đến tòan bộ hệ thống (quan trọng)***
* **Extensiblity (Khả năng mở rộng):** khi cần thêm chức năng mới thì chỉ cần thêm code mà không cần thay đổi code (hạn chế tối đa việc sửa code)
* **Reusability(khả năng tái sử dụng):** hạn chế công sức để code nếu module được sử dụng lại nhiều lần trong module khác mà viết lại sẽ mất nhiều thời gian.
* **High fan-in:** khi thiết kế các class xem các class có phù hợp hay chưa, nếu class đó chỉ được sử dụng bởi 1 class thì phải cân nhắc có cần tách ra 2 class ko
* **Low-to-medium fan-out(sự phụ thuộc):** đừng để 1 class quá phụ thuộc vào nhiều class khác
* **Portability(tính di động):** 1 ứng dụng có thể chạy trên nhiều môi trường khác nhau, hiện nay được hỗ trợ trên framework trên nhiều ngôn ngữ
* **Leanness:** nếu hệ thống có nhiều phần ko có trong yêu cầu của khách hàng mà đội ngũ cntt vẫn thêm vào, những bộ phận khác sẽ ko test phần đó -> hệ thống sẽ gặp rủi ro vì chưa chắc được chỗ đó có lỗi hay ko
* **Stratification(phân rã):** khi phân rã hệ thống thì 1 ng muốn đọc hiểu 1 module trong hệ thống ko cần phải đọc những module khác
* **Standard technique:** có những tiêu chuẩn được sử dụng cho rất nhiều hệ thống, vì vậy chúng ta ko nên thay đổi nó. Tiêu chuẩn khi thiết kế giao diện, khi thấy các icon là hiểu nó sẽ làm gì, chức năng gì
* **Các cấp độ trong thiết kế (5 cấp độ):**
* **Software System:** phát biểu về hệ thống, miêu tả hệ thống đó sẽ làm gì (mô tả)
* **Division into Subsystems or Packages:** dựa vào b1 để phân rã ra các component hoặc các packages (thiết kế kiến trúc)
* **Division into Classes within packages:** thiết kế giao diện cho kiến trúc của giai đoạn đó, sự kiên kết giữa các class
* **Division into Routines:** các chức năng, hàm
* **Internal Routine Design:** thiết kế cho các phương thức sử dụng nội bộ bên trong class hoặc packages nào đó mà bên ngoài không sử dụng được

1. **Các vấn đề liên quan đến thiết kế: Lớp, phương thức.**
   1. Lớp: working classes

* Chương trình lớn chia thành nhiều chương trình con chứa các tập lệnh-> dễ quản lí, chỉnh sửa code
* Lập trình hàm quan tâm đến chức năng của hệ thống (functional first)
* Lập trình HĐT quan tâm đế dữ liệu, sử dụng dữ liệu dự trù trước thuộc tính trong tương lai (data first)
* Public thay đổi thì không kiểm soát được
* Private –set: thay đổi kiểm soát được
* **Class Foundation: Abstract data type (ADTs) nền tảng lớp: kiểu dữ liệu trừu tượng**
* **Nhung thanh phan Public cua class ben ngoai co the goi duoc**
* Kiểu dữ liệu trừu tượng ( abstract ) là 1 tập dữ liệu và các chức năng làm việc trên dữ liệu đó.
* Các hoạt động:

▪ Mô tả dữ liệu cho phần còn lại của chương trình

▪ Cho phép phần còn lại của chương trình thay đổi dữ liệu

* Lợi ích của việc sử dụng ADT:

▪ Ẩn chi tiết cài đặt (1 dạng của trừu tượng hóa).

▪ Những thay đổi không ảnh hưởng đến toàn bộ chương trình

▪ Làm cho interface nhiều thông tin hơn

▪ Dễ dàng cải thiện hiệu suất hơn

▪ Chương trình rõ ràng là chính xác hơn

* **Good class interface**
* Để tạo 1 interface tốt: xác định phần nào trong class ta phải public -> quan tâm đến cách đặt tên sao cho nhất quán với nhau
* Cung cấp những cặp dịch vụ đối lập nhau (vd: addEmployee, removeEployee).
* Những gì không liên quan chuyển qua class khác
* Mỗi class mô tả 1 loại đối tượng
* Nhiều tên là thành phần giao tiếp bên ngoài, được ta đặt tên là cái gì và làm sao tránh được (vd: làm phương thức A trước sau đó tới phương thức B, tạo phương thức C đưa pt A và B vào hoặc ghi chú pt).
* Nên nhận ra sự suy giảm tính nhất quán của interface sau mỗi lần mentornance.
* Không public các thành phần mà nó không public.
* Quan tâm đến đặc điểm trừu tượng và cohesion (là sự liên quan giữa các công việc được thực hiện trong 1 phương thức)
* Tạo một abstraction (sự trừu tượng) tốt cho interface (giao diện) để thể hiện và đảm bảo rằng các chi tiết vẫn ẩn sau sự trừu tượng (abstract).
* Tính trừu tượng: là khả năng xem một hoạt động phức tạp ở dạng đơn giản hóa
* Những điểm chính:

▪ Hãy chắc chắn rằng bạn hiểu những gì trừu tượng mà lớp đang triển khai (những cái nào nên được phơi bày)

▪ Cung cấp các dịch vụ đi đôi với các mặt đối lập của chúng

▪ Chuyển thông tin không liên quan sang lớp khác

▪ Làm cho interface có lập trình thay vì ngữ nghĩa khi có thể

▪ Cẩn thận với sự xói mòn tính trừu tượng của interface đang được sửa đổi (Ví dụ)

▪ Không thêm các thành viên công khai không phù hợp với interface trừu tượng(abstract)

▪ Cân nhắc sự trừu tượng và sự gắn kết với nhau

* **Good encapsulation ( sự đóng gói tốt )**
* Hạn chế tối đa khả năng truy cập đến class và thành phần của class
* Dữ liệu không public để đảm bảo tính toàn vẹn của dữ liệu
* Khi thiết kế 1 class không được đứng ở góc độ của người dùng mà phải thiết kế theo yêu cầu.
* Không được để thuộc tính public khi phương thức public
* Coupling(tính liên quan, phạm vi) là liên kết giữa các class với nhau khác với Cohesion là liên kết giữa các thành phần trong cùng 1 class.
  + - Không nên liên kết quá chặt giữa các class vì sự phụ thuộc quá nhiều dẫn đến việc khó thay đổi
    - Các thành phần trong hệ thống phải có liên kết với nhau.
* Protected là chỉ có class cha và class con của nó sử dụng được các thuộc tính, các đối tượng ngoài không thể sử dụng được.
* **Design and implementation issues ( các vấn đề triển khai và thiết kế )**
* “has a”: 1 đối tượng là thuộc tính thông qua 1 đối tượng khác. Ví dụ: class DiaChi với NhanVien
* “is a”: Mối quan hệ thừa kế; thừa kế thừa kế khi lớp con có nhiều thuộc tính đặc trưng của lớp cha.
* Thiết kế và tài liệu để kế thừa hoặc cấm nó: Nếu một lớp không được thiết kế để kế thừa từ đó, hãy làm cho các thành viên của nó không phải là ảo trong C ++, cuối cùng trong Java
* Tuân thủ Nguyên tắc thay thế Liskov (LSP):

• bạn không nên kế thừa từ một lớp cơ sở trừ khi lớp dẫn xuất thực sự "là một" phiên bản cụ thể hơn của lớp cơ sở

• Các lớp con phải có thể sử dụng được thông qua giao diện lớp cơ sở mà không cần người dùng biết sự khác biệt (các lớp dẫn xuất mới đang mở rộng các lớp cơ sở mà không thay đổi hành vi của chúng).

* Hãy nghi ngờ (suspicious) về các lớp chỉ có một trường hợp
* Hãy nghi ngờ về các lớp cơ sở mà chỉ có một lớp dẫn xuất ( derived class )
* Hãy nghi ngờ về các lớp ghi đè một phương thức và không làm gì bên trong phương thức dẫn xuất (phương thức trong lớp cơ sở không phải là một phương thức phổ biến)
* Tránh những cây kế thừa sâu
* Thích đa hình hơn để kiểm tra kiểu mở rộng:

1. **Quy trình chất lượng cao**

* **Valid Reasons to Create a Routine**
* 1 quy trình là 1 phương pháp hoặc thủ tục riêng lẻ có thể lập hóa đơn cho 1 mục đích duy nhất.
* Class là 1 đối tượng; không có trạng thái, định nghĩa có nhiều cái instance của object đó
* Instance: có trạng thái (new object).
* 1 đối tượng duy nhất không nên new Object (tạo đối tượng) mà tạo class static, tạo hằng trong class đó, không phải dạng biến mà là dạng hằng, gán sẵn giá trị.
* Giảm sự phức tạp
* Giới thiệu tính trừu tượng dễ hiểu: đưa 1 phần code vào quy trình được đặt tên tốt là 1 trong những cách tốt nhất để ghi lại mục đích là:
  + - Tránh code trùng lặp
    - Hỗ trợ thừa kế
    - Hiệu suất của việc tạo ra hệ thống
    - ẩn đi chi tiết chương trình con -> chỉ cần gọi lại tên
* **Cohesion**
* Cohesion là liên kết giữa các thành phần trong cùng 1 class (ví dụ: mối liên kết giữa các phương thức trong cùng 1 class)
* Cohesion: đề cập đến mức độ liên quan chặt chẽ của hoạt động trong 1 quy trình.
* Functional cohesion: là loại liên kết tốt nhất và mạnh nhất, xảy ra 1 quá trình thực hiện 1 và chỉ 1 công việc (không phải lệnh trong hàm).
  + - Ví dụ: Functional cohesion của chương trình giải phương trình bậc 2.
      1. Nhập a, b, c (viết chương trình nhập cho 1 giá trị -> gọi lại 3 lần).
      2. Tính delta
      3. Xác định nghiệm.
* Sequential cohesion: các hàm thực hiện công việc theo 1 thứ tự nhất định, chia sẻ dữ liệu từ bước này sang bước khác đảm bảo công việc vẫn không ảnh hưởng.
* Communicational cohesion: các công việc trong hàm đó sử dụng cùng 1 dữ liệu nhưng không liên quan đến nhau (không theo 1 thứ tự nhất định).
* Temporal cohesion: các hoạt động được kết hợp thành 1 quy trình vì tất cả chúng được thực hiện cùng 1 lúc, liên quan đến thời gian; các công việc trong cùng 1 hàm phải kết thúc thành công. Gợi nhớ đến transaction, nhiều công việc bên trong chỉ hoàn tất khi dữ liệu công việc bên trong nó hòan tất, nếu không nó sẽ rollback => đảm bào thực hiện.
* Sequential và communicational không tốt bằng Functional vì nó lệ thuộc và thực hiện nhiều việc.
* **Good Routine Names**
* Phương thức trả về giá trị nào ta nên đặt tên về giá trị đó.
* Đặt tên đối nghịch nhau; nhất quán; dễ hiểu.
* Phương thức thực hiện công việc thì nên dùng động từ mạnh đặt tên cho phương thức.
* **High Quality Code**
* **Static factory methods**
* Phương thức có thể sử dụng thông qua class mà ko cần tạo đối tượng
* Biến static trong 1 class là biến dùng chung cho tất cả các đối tượng trong class đó (những biến không phải static thì nó thuộc vùng nhớ riêng của từng đối tượng). Khi đối tương thay đổi giá trị của biến static thì tất cả các giá trị static đều bị ảnh hưởng.
* Khối lệnh static trong 1 class chỉ được thực hiện 1 lần duy nhất khi bất kỳ thành phần nào trong class này gọi đến -> hiệu suất tăng lên, tiết kiệm bộ nhớ.
* **Enforce noninstantiability with a private constructor**
* Không triệt để vì người khác có thể tạo class khác từ class này.
* Dùng private, constructor không tạo ra được đối tượng.
* Nếu tạo 1 class mà trong đó không có constructor và khi ta biên dịch hệ thống tự động chèn vào constructor public cho class đó.
* Constructor của lớp con có lệnh đầu tiên là g\_constructor của lớp cha nếu không có nó tự động thêm 1 constructor để private ở lớp cha thì lớp con không gọi được.
* **Not required to create a new object**
* Phương thức khi được gọi chỉ trả về 1 trong 2 giá trị TRUE or FALSE tương ứng với 2 giá trị T or F truyền vào.
* Dùng phương pháp tĩnh để tạo ra đối tượng không cần thiết
  + - Dữ liệu cơ bản Boolean -> có các phương thức cho chuyển đổi dữ liệu trả về đối tượng đó.
    - Ví dụ: tạo mẫu Singleton chỉ tạo 1 đối tượng, khi có đối tượng rồi thì không tạo nữa vì Singleton chỉ trả về 1 đối tượng trong 1 ngữ cảnh nào đó, trong ngữ cảnh khác phải tạo ra đối tượng khác.
* **Avoid creating unnecessary objects**
* Mỗi 1 hằng chuỗi là 1 con trỏ
  + - String s = “string123” : con trỏ kiểu string
    - String s = new String(“string123”) : con trỏ trỏ đến con trỏ (tốn 3 bộ nhớ).
* Không tối ưu vì mỗi lần gọi hàm lại tạo ra-> tốn thời gian, tốn bộ nhớ.
* **Minimize the scope of local variables**
* Ưu điểm:
  + - Có thể bắt đầu từ phần tử có 1 vị trí tùy ý
    - Có thể thay đổi kích thước mảng chạy
    - Có thể lấy giá trị trước và sau của phần tử ở thời điểm đang chạy
* Biến cục bộ là biến được tạo ra bên trong 1 phương thức sử dụng từ thời điểm nó được khai báo; chỉ được sử dụng bên trong phạm vi nó được khai báo -> nên khai báo biến cục bộ ngay khi cần sử dụng nó, không nên khai báo quá rộng phạm vi sử dụng.
* **Prefer for-each loops to traditional for loops**
* **Avoid strings where other types are more  
  appropriate**
* Hạn chế dữ liệu kiểu chuỗi nếu có dữ liệu khác thay thế
* Ưu điểm của dữ liệu chuỗi: nó biết bản thân giá trị nó là cái gì nhưng khi xử lí lại tốn kém.
* Không dùng dữ liệu chuỗi để nhận dạng hoặc làm khóa tìm kiếm.
* **Adhere to generally accepted naming  
  conventions**
* Nên làm theo qui ước vì có 1 số framework chỉ hiểu khi làm đúng theo qui ước.
* **Exception**
* Uncheck exception: không khắc phục được (lỗi do người lập trình, người sử dụng không sửa được, người lập trình phải mở code ra sửa).
* Check exception: dễ khắc phục lỗi.

**CHƯƠNG 5**

1. **Nguyên tắc thiết kế.**

* Khi thiết kế class và các thành phần (hàm, phương thức) => mở (open) cho việc mở rộng; đóng (close) cho việc chỉnh sửa code
* Khi có nhu cầu nên hạn chế tối đa việc chỉnh sửa code cũ, hỗ trợ mở rộng code mới. Vì khi thêm code mới ta chỉ kiểm thử phần code thêm mới đó xem có trơn tru không,
* Khi sửa code -> rủi ro cao; không đảm bảo rằng trong quá trình chỉnh sửa không đụng đến các phần liên quan; phải test lại các chức năng; phải kiểm soát mã nguồn chỉnh sửa khó khăn đặc biệt là hệ thống đang trong quá trình sử dụng.
* Sử dụng interface giảm đi sự phụ thuộc của các thành phần bên trong hệ thống.
* Coupling các thành phần hệ thống liên kết không quá chặt
* **Dependecy Invertion**
* Các tầng ở mức cao có độ phức tạp nhiều, nếu nó tác động trực tiếp tới tầng dưới -> lệ thuộc vào tầng dưới-> độ phức tạp tần trên tăng lên-> dùng tầng trung gian (abstraction layer).
* **Interface Segregation**
* Khi 1 class implement 1 interface mà có các phương thức trong interface đó class không dùng đến -> chia interface thành các interface có phương thức liên quan đến nhau
* Thay vì dùng 1 interface lớn ta nên tách ra thành nhiều interface nhỏ với nhiều mục đích cụ thể
* **Single Responsibility**
* Mỗi class chỉ nên mô tả 1 loại đối tượng, mỗi class thực hiện một nhiệm vụ
* Khi 1 class, 1 phương thức cần nhiều thông tin dữ liệu -> thực hiện việc yêu cầu gọi lại các phương thức khác
* Một class chỉ nên giữ một trách nhiệm duy nhất (chỉ có thể sửa đổi class với một lý do duy nhất)
* **Liskow substitution**
* Lớp con thừa kế lớp cha và có những đặc điểm riêng của nó
* Nếu phương thức lớp cha rỗng hoặc throw exception phải xem lại
* Lớp con override lớp cha là bình thường
* Trong một chương trình các object của class có thể thay thế class cha mà không làm thay đổi tính đúng đắn của chương trình

1. **Design pattern**

* Mô tả giúp ta hiểu mẫu pattern có từ đâu, giải quyết vấn đề gì.
* Mỗi mẫu mô tả một vấn đề lặp đi lặp lại trong môi trường, mẫu đi từ thực tế (từ các mẫu triển khai thực tế, họ gặp vấn đề nhiều lần trong nhiều hệ thống) => đưa ra các mẫu có các giải pháp (các giải pháp chính chứ không giải quyết hết tất cả vấn đề cân chỉnh cho phù hợp)
* Thành phần mẫu thiết kế
* Tên mẫu và phân loại (mẫu đó thuộc nhóm nào)
* Mục tiêu: là giải quyết các vấn đề gì
* Hệ quả khi ta sử dụng mẫu đó
* Sử dụng mẫu thiết kế
* Mỗi pattern là nền tảng tổng quan cho người phát triển
* Best practices: mẫu sinh ra từ thực tế các vấn đề đã gặp
* Các loại thiết kế
* Creational Patterns
* Structural Patterns
* Behavioral Patterns