**XÂY DỰNG PHẦN MỀM**

**Chương 4**

1. **Challenges(5)**

* **Vấn đề phức tạp**(Wicked problem): là những vđ xảy ra rồi ngta mới biết
* **Quá trình cẩu thả, bừa bộn**(Sloppy Process)
* **Cân đối và ưu tiên** (Tradeoffs and Prioities): xét “ưu tiên” trước

->Cân đối mục tiêu và các điều kiện để thực hiện mục tiêu

->Cân đối yêu cầu, xem xét ycau nào có thể giảm bớt

- **Không xác định** (Nondeterministic):

-> Kết quả ko dự đoán được, vì 1 y/c có thể đưa 3 người ra 3 bản TK khác nhau

1. **Các khái niệm thiết kế chính (11)**

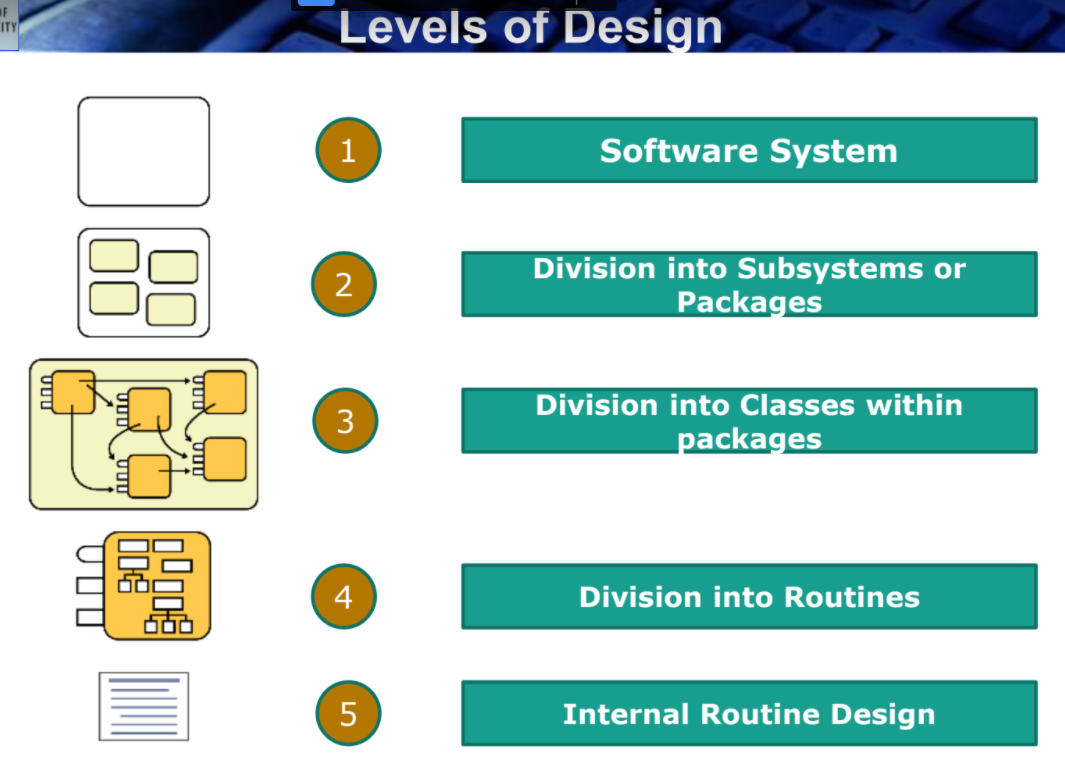
* **Ít phức tạp (**Minimal complexity): TK nên giảm thiểu độ phức tạp
* **Dễ bảo trì**(Ease of maintenance): thiết kế cho lập trình viên bảo trì
* **Gắn kết lỏng lẻo**(Loose coupling): giữ kết nối giữa các phần khác nhau của chương trình ở mức tối thiểu
* **Khả năng mở rộng** (Extensibility): Khi cần thêm c/n mới chỉ cần gán code mới, hạn chế tối đa vc chỉnh sửa code
* **Khả năng tái sử dụng** (Reusability): có thể tái sử dụng các tphan của nó trong các hệ thống khác nhau 🡪hạn chế công sức code.
* **High fan-in**: Cân nhắc khi thiết kế class nếu class cha chỉ có 1 class con 🡪 xem lại class con có nhất thiết tách ra không
* **Low-to-medium fan-out (sự phụ thuộc)**: 1 class ko nên phụ thuộc nhiều hơn 6 class khác
* **Khả chuyển** (Portability): dễ dàng di chuyển nó sang môi trường khác 🡪Vik 1 ứng dụng chạy trên nhiều mtr khác nhau
* **Tinh gọn** (Leanness): hệ thống không có phần thừa
* **Phân tầng** (Stratification): Cố gắng phân thành các tầng để có cái nhìn nhất quán các tầng với nhau

Ex: Thk một tầng tương tác với code cũ nếu ht mới cần sd lại nhiều code cũ

* **Tiêu chuẩn kỹ thuật**(Standard techniques): Các vđề trải qua nhiều DA 🡪 thành tiêu chuẩn 🡪 TK theo những tiêu chuẩn đó.

Ex: Microsoft tạo ra các tiêu chuẩn cho KH sdung tốt, tạo sự lệ thuộc người dùng lớn

1. **Các mức trong TK(5)**



1.Phát biểu, mô tả về HT

2. Phân rã ra các **component**, các **pakage**, lựa chọn các **framework**, **TK kiến trúc**

3.Phân rã các class

4. Xem trong các class nội dung có gì, các **pthuc, hàm**

5. TK cho các **pthuc internal** (phương thức nội bộ, chỉ dùng cho bên trong, bên ngoài ko truy cập đc)

Đa hình 🡪 static (tĩnh): xđinh đc pthuc nao ngay khi biên dịch

Dynamic(động): khi chạy mới bik

**3. Kiểu dữ liệu trừu tượng (ADT)**

* Là 1 tập data và các chức năng làm việc trên dữ liệu đó
* 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

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

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

▪ Làm cho giao diện nhiều thông tin hơn

▪ Dễ dàng cải thiện hiệu suất hơn

▪ Chương trình rõ ràng 🡪 chính xác hơn (vì cấu trúc hthong theo từng đối tượng)

1. **Tạo ra một interface tốt**

* Xác định phần nào trong class ta phải public 🡪 Quan tâm cách đặt tên, sao cho tên nhất quán với nhau
* Cung cấp những cặp dịch vụ đối lập nhau
* Những gì ko liên quan 🡪 Chuyển qua class khác. Mỗi class mô tả một loại đối tượng

Ex: AddEmployee, RemoveEmploy

* Đặt tên theo hướng programatic
* Nhận ra sự suy giảm tính nhất quán của Interface
* Ko public các tp mà nó ko public
* Quan tâm đến các đặc điểm trừu tượng và cohesion (sự lq giữa các cv)

1. **Tính đóng gói (Good Encapsulation)**

* Hạn chế tối đa khả năng truy cập đến class và các tp của class
* Dữ liệu ko public 🡪 Đảm bảo tính toàn vẹn
* Khi TK, đặt vị trí ở người TK, ko đặt vị trí ở người dùng
* Không public 1 pthuc chỉ vì nó sử dụng các pthuc public khác
* Coupling (Gắn kết) sự lket các class vs nhau, mỗi class ko quá chặt (khó thay đổi)

1. **Vấn đề TK và TH (design and implementation issue)**

* Nên nghi ngờ khi 1 class chỉ có 1 instance hoặc class cha chỉ có 1 class con
* Nghi vấn 1 class override 1 pthuc mà ko code gì
* Tránh kế thừa quá sâu

+ has-a : 1 đt là thuộc tính cua dt khác, vd: class DiaChi với NhanVien

+is-a : Mối qhe thừa kế, lớp con có thuộc tính lớp cha

* Instance là 1 thực thể cụ thể của class
* Instance có trạng thái, class ko trạng thái. Đối t khác nh, trạng th khác nhau

1. **Rountiner**
2. **Valid Reasons to create a rountine** (6)

* Giảm độ phức tạp
* Tránh lặp code
* Hỗ trợ kế thừa
* Cải thiện hiệu suất
* Ẩn đi chi tiết ctrinh con 🡪 Chỉ gọi lại tên
* Đặt tên thể hiện được chức năng

1. **Good Rountine Name** (5)

* Đặt tên thể hiện được chức năng hàm
* Đặt tên đối nghịch nhau 🡪 Nhất quán, dễ hiểu
* Không đặt tên hàm bằng số
* Dùng động từ mạnh
* Cần đặt tên dài khi cần thiết

1. **Cohesion**: Các cv liên quan với nhau trong 1 hàm mang tính tương đối(4)

* ***Function cohesion***: khi chỉ thực hiện 1 và chỉ 1 cv
* ***Sequential cohesion***: các cv thực hiện theo 1 tình tự nhất định, có thể share dữ liệu vs nhau

(ko tốt bằng function vì nó lệ thuộc và th nhiều vc)

* ***Comunicational cohesion***: dùng chung dl nhưng ko liên quan tới nhau
* ***Temponal cohesion***: Dl liên quan đến time, kết thúc cùng thời điểm

1. **High quality code**
2. Các phương pháp factory tĩnh **(Static factory methods)**

* Họ có tên **(They have names)**

Cùng kiểu dữ liệu, cùng tên, cùng chữ ký 🡪 Ko chạy đc 🡺Dùng static

* Họ không bắt buộc phải tạo đối tượng mới **(They are not required to create a new object)**

Dùng pthuc static ko tạo ra các đối tượng ko cần thiết (vd T F slide)

* Có thể trả về một đối tượng của bất kỳ kiểu phụ **(Can return an object of ansubtype)**

(vd slide) Costructor lớp nào trả về lớp đó, ko trả về đt lớp con. Một số kiểu lớp có thể nhận tham chiếu của lớp con

Khi ko kb public, private 🡪 tự động chuyển sang Friendly và chỉ sd đc 1 package

+ Phương thức static: sd qua tên class mà ko cần tạo đt 🡪 cần tối ưu lại code

+ Biến static: Biến dùng chung cho all đt lớp đó.

1. Thực thi tính không thay đổi với một Constructor riêng tư **(Enforce noninstantiability with a private constructor)**

**-** Nếu tạo 1 class mà ko có constuctor nào 🡪 Khi complier hthong sẽ tự tạo 1 cons public cho class đó

**-** Cons class con có lệnh đầu tiên là gọi cons class cha. Để private class cha thì con ko gọi đc

1. Tránh tạo các đối tượng không cần thiết **(Avoid creating unnecessary objects)**

(vd slide) Code ko tối ưu vì mỗi lần gọi hàm thì sẽ lại tính BoomStart, BoomEnd

-Mỗi một hằng chuỗi là 1 con trỏ

+TH1: s là con trỏ trỏ đến con trỏ (tốn 3 bộ nhớ)

+TH2: s là con trỏ kiểu string

1. Giảm thiểu phạm vi của các biến cục bộ **(Minimize the scope of local variables)**

**-** Biến cục bộ đc tạo ra bên trong 1 phương thức, sd từ thời điểm nó đc khai báo. Chỉ sd phạm vi nó đc kb

**-** Nên kb biến cục bộ ngay khi sd. Ko nên kb quá rộng pvi sd

Vd: int i=0 (Gián gtri cho i) khác int i; (Khởi tạo)

1. Ưu tiên cho từng vòng lặp thành vòng lặp for truyền thống **(Prefer for-each loops to traditional for loops)**

\*Ưu điểm: - Có thể bắt đầu từ vị trí phần tử tùy ý

- Có thể thay đổi kích thước mảng

- Có thể lấy gtri trước, sau của ptu thời điểm đang chạy

1. Tránh các chuỗi mà các loại khác nhiều hơn hích hợp **(Avoid strings where other types are more appropriate)**

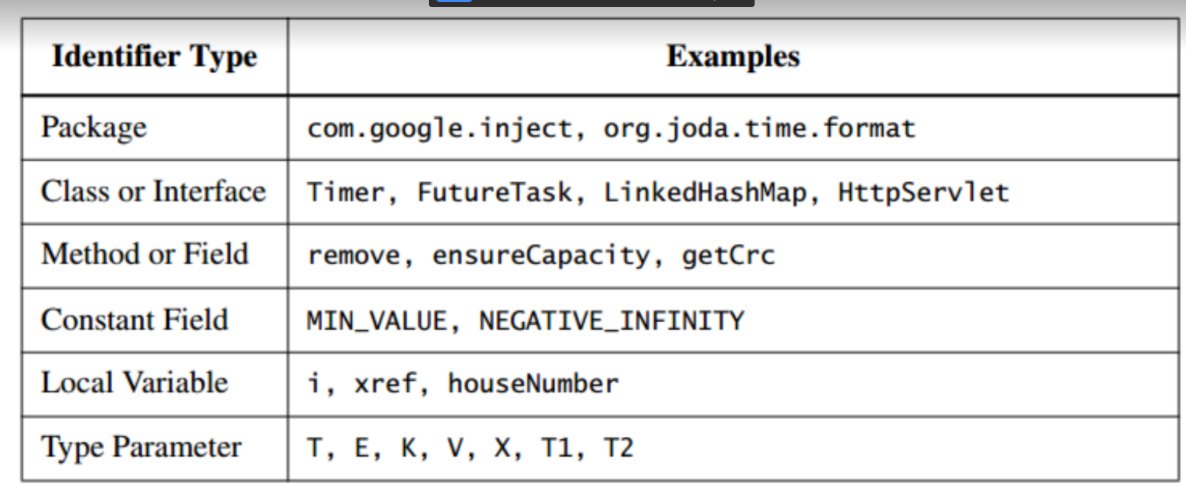
**-** Hạn chế dl kiểu chuỗi nếu có dl khác thay thế

\* Ưu điểm: - Nó biết bản thân gtri nó là gì nhưng khi xly lại tốn kém

- Ko nên dùng chuỗi để nhận dạng or lm khóa tìm kiếm

1. Tuân thủ cách đặt tên được chấp nhận chung các quy ước**. (Adhere to generally accepted naming conventions)**

**-** Nên làm theo quy ước vì có 1 vài framework chỉ hiểu khi làm theo quy ước

vd: 

1. Ngoại lệ (**Exception**)

-Lỗi khắc phục đc

-Lỗi ctrinh ko khắc phục( ng sd ko thể sửa, chỉ có ng coder mở code lên sửa đc)

**Chương 5**

**1. Nguyên tăc thiết kế (Design Principles)**

1. **Mỗi class mô tả 1 đối tượng, thực hiện 1 nv**(**Single Responsibility**)

* Giảm đi sự phức tạp của class

Vd: Thay vì DBHelper (save, open, get, close) 🡪 DBSave, DBOpen….

1. **Mở cho sự mở rộng class, đóng cho sự chỉnh sửa class** **(Open Close)**

* Thay vì sửa đổi code, chúng ta nên mở rộng kế thừa.

Vì khi thêm code mới , ta chỉ cần kiểm thử pm mới có ok ko

Còn khi sửa code rủi ro cao, dễ đụng đến các phần khác, phải test lại hết, kiểm soát mã nguồn chỉnh sửa khó khăn (đặc biệt vs các hthong đã đc sd)

Vd: thêm một phương thức vận chuyển mới, chúng ta lại phải bổ sung một case nữa trong method ***calculateShipping*** 🡪 chúng ta nên tách rời logic xử lý tính phí vận chuyển vào một interface ***Shipping***

1. **Trong ctr, các class con có thể thay thế các đt class cha mà ko làm mất đi tính đúng đắn của ctrinh. (Liskov Substitution)**

Vd: Dog và Bird implement interface Animal{void fly()} 🡪Tạo thêm 1 interface AnimalAbleFly

1. **Thay vì 1 interface lớn ta tách thành nhiều interface nhỏ hơnvs mục đích cụ thể** (**Interface Segregation Principle)**

Vd: Interface Animal{ void eat(); void run(); void fly() } 🡪 tạo 3 Interface extend Animal

1. **Các tầng cấp cao ko nên phụ thuộc vào tầng cấp thấp(Inversion)** (độ phức tạp tầng trên sẽ bị tăng lên vì nó ở mức cao nên tác động trực tiếp tới tầng dưới) 🡪 Dùng tầng trung gian (abstraction layer)

Abstract nên phụ thuộc vào chi tiết và ngược lại

High Level Classes 🡪 Abstraction Layer 🡪 Low level classes

Vd: chúng ta có 2 module cấp thấp ***BackendDeveloper*** và ***FrontendDeveloper*** và 1 module cấp cao ***Project 🡪*** bổ sung thêm một abstraction Developer để các module trên phụ thuộc vào

**2. Design Pattern**

\*Khái niệm:

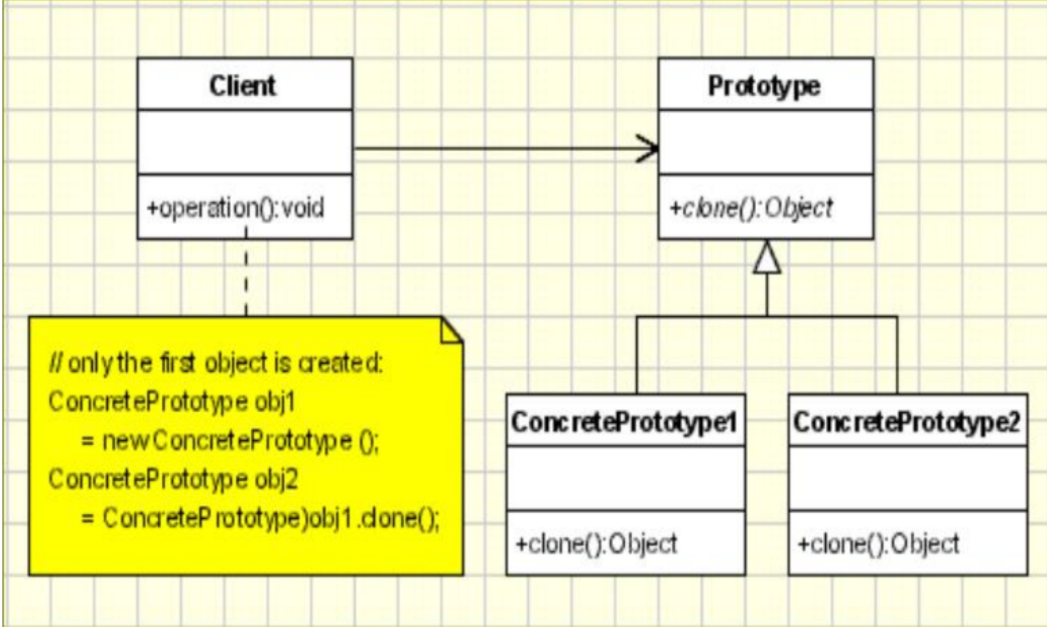
- Mỗi mẫu mô tả 1 vđe lặp đi lặp lại trong môi trường (những vấn đề gặp nhiều lần trong nhiều hthong từ thực tế) 🡪 Giải pháp là đưa ra các core solution giải pháp chính, chứ ko giải quyết all vde

\*Types:

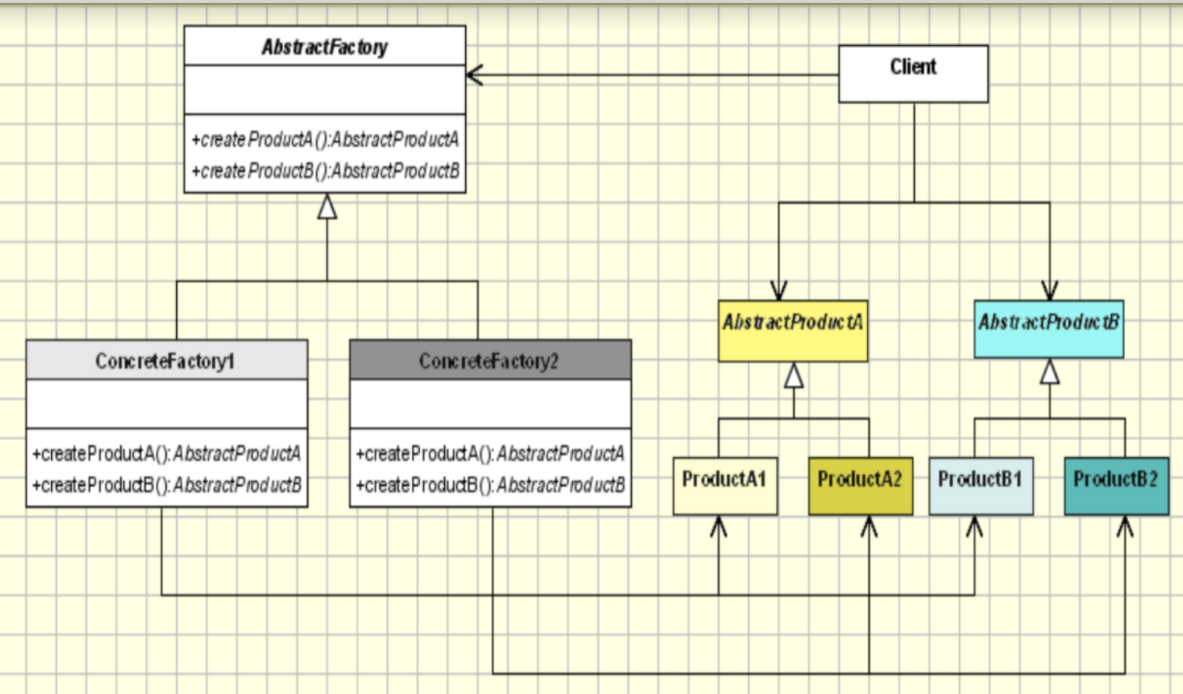
**- Creational Patterns:** Cách tạo đt thay vì tạo trực tiếp bằng cách sd nhà điều hành ms thì tạo các đối tượng trong khi ẩn creation logic

Vd 3 cái:

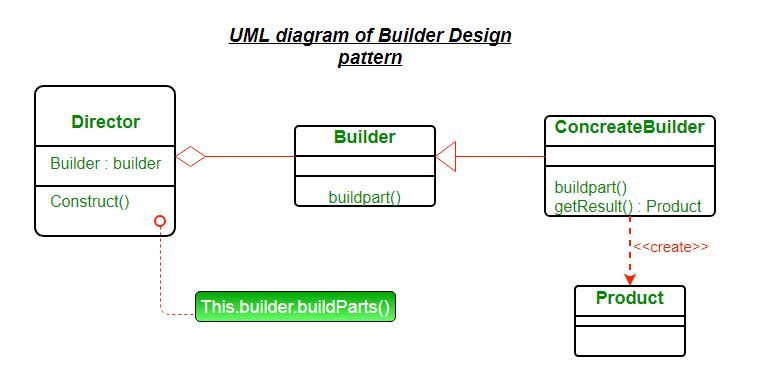
*+* ***Prototype Pattern***: Khởi tạo một đối tượng bằng cách clone 1 đt đã có sẵn thay vì dùng new. Tạo ra bản sao giống 100% so vs bản gốc. Chúng ta có thể chỉnh sửa mà ko ảnh hưởng đến đt gốc.



*+****AbstractFactory( Tạo các đt tương tự nhau, tạo đt đặc biệt từ lớp cha)***: Định nghĩa interface/abstract tạo ra 1 đối tượng liên q (hoặc phụ thuộc) mà ko cần chỉ rõ lớp con cụ thê của chúng



*+* ***Builder Pattern (Nhiều tham số truyền vào, phải tạo rất nhiền cons trong TH khác nhau)***: Tạo 1 đt phức tạp dựa trên đt đơn giản. Tiếp cận từng bước và xd độc lập vs các đt khác



* **Behavioral Patterns:** Quan tâm đến giao tiếp giữa các đối tượng
* **Structural Patterns:** Quan tâm đến lớp và đối tượng thành phần