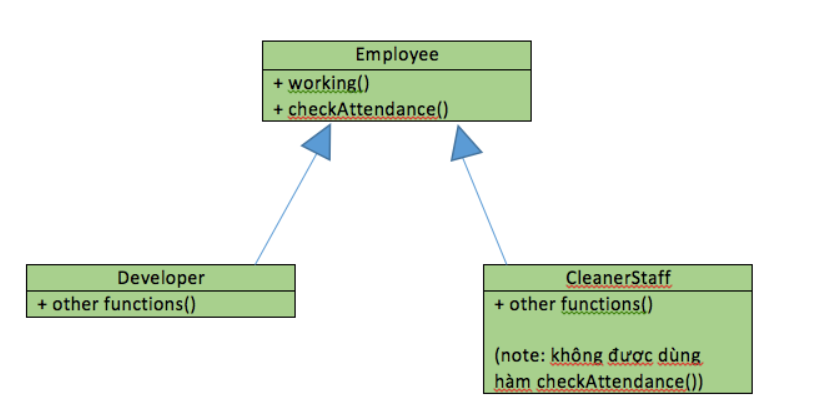
**1.Phương Thức Factory:**

**2. Solid**

**https://topdev.vn/blog/solid-la-gi/**

**SOLID** là viết tắt của 5 chữ cái đầu trong 5 nguyên tắc thiết kế hướng đối tượng. Giúp cho lập trình viên viết ra những đoạn code dễ đọc, dễ hiểu, dễ maintain. Nó được đưa ra bởi [Robert C. Martin](http://www.goodreads.com/author/show/45372.Robert_C_Martin) và Michael Feathers. 5 nguyên tắc đó bao gồm:

* **S**ingle responsibility priciple (SRP)
  + Mỗi lớp chỉ nên chịu trách nhiệm về một nhiệm vụ cụ thể nào đó mà thôi.
  + Nguyên lý đầu tiên ứng với chữ S trong SOLID, có ý nghĩa là một class chỉ nên giữ một trách nhiệm duy nhất. Một class có quá nhiều chức năng sẽ trở nên cồng kềnh và trở nên khó đọc, khó maintain.
  + Giảm đi sự phức tạp của class
  + Vd: Thay vì DBHelper (save, open, get, close) DBSave, DBOpen….
* **O**pen/Closed principle (OCP)
  + Hạn chế sửa đổi một Class có sẵn, nhưng có thể mở rộng bằng kế thừa.
  + Theo nguyên lý này, mỗi khi ta muốn thêm chức năng cho chương trình, chúng ta nên viết class mới mở rộng class cũ (bằng cách kế thừa hoặc sở hữu class cũ) chứ không nên sửa đổi class cũ. Việc này dẫn đến tình trạng phát sinh nhiều class, nhưng chúng ta sẽ không cần phải test lại các class cũ nữa, mà chỉ tập trung vào test các class mới, nơi chứa các chức năng mới.
  + Thông thường việc mở rộng thêm chức năng thì phải viết thêm code, vậy để thiết kế ra một module có thể dễ dàng mở rộng nhưng lại hạn chế sửa đổi code ta cần làm gì. Cách giải quyết là tách những phần dễ thay đổi ra khỏi phần khó thay đổi mà vẫn đảm bảo không ảnh hưởng đến phần còn lại.
* **L**iskov substitution principe (LSP)
  + Các đối tượng (instance) kiểu class con có thể thay thế các đối tượng kiểu class cha mà không gây ra lỗi.
  + 
  + Nếu chúng ta tạo ra một lớp CleanerStaff kế thừa từ lớp Employee, và implement hàm working() cho lớp này, thì mọi thứ đều ổn, tuy nhiên lớp mới này cũng lại có hàm checkAttendance() để điểm danh, mà như thế là sai quy định dẫn đến chương trình bị lỗi. Như vậy, thiết kế lớp CleanerStaff kế thừa từ lớp Employee là không được phép. Có nhiều cách để giải quyết tình huống này ví dụ như tách hàm checkAttendance() ra một interface riêng và chỉ cho các lớp Developer, Tester và Salesman implements interface này.
  + Vd: Dog và Bird implement interface Animal{void fly()} Tạo thêm 1 interface AnimalAbleFly
* **I**nterface segregation principle (ISP)
  + Thay vì dùng 1 interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể.
  + Nguyên lý này rất dễ hiểu. Hãy tưởng tượng chúng ta có 1 interface lớn, khoảng 100 methods. Việc implements sẽ rất vất vả vì các class impliment interface này sẽ bắt buộc phải phải thực thi toàn bộ các method của interface. Ngoài ra còn có thể dư thừa vì 1 class không cần dùng hết 100 method. Khi tách interface ra thành nhiều interface nhỏ, gồm các method liên quan tới nhau, việc implement và quản lý sẽ dễ hơn.
  + Vd: Interface Animal{ void eat(); void run(); void fly() } Tạo 3 Interface extend Animal
* **D**ependency inversion principle (DIP)
  + 1.Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction.
  + 2.Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại (Các class giao tiếp với nhau thông qua interface (abstraction), không phải thông qua implementation.)
  + Có thể hiểu nguyên lí này như sau: những thành phần trong 1 chương trình chỉ nên phụ thuộc vào những cái trừu tượng (abstraction). Những thành phần trừu tượng không nên phụ thuộc vào các thành phần mang tính cụ thể mà nên ngược lại.
  + Những cái trừu tượng (abstraction) là những cái ít thay đổi và biến động, nó tập hợp những đặc tính chung nhất của những cái cụ thể. Những cái cụ thể dù khác nhau thế nào đi nữa đều tuân theo các quy tắc chung mà cái trừu tượng đã định ra. Việc phụ thuộc vào cái trừu tượng sẽ giúp chương trình linh động và thích ứng tốt với các sự thay đổi diễn ra liên tục.
* Tổng kết SOLID là 5 nguyên tắc cơ bản trong việc thiết kế phần mềm. Nó giúp chúng ta tổ chức sắp xếp các function, method, class một cách chính xác hơn. Làm sao để kết nối các thành phần, module với nhau. Rõ ràng, dễ hiểu Teamwork là điều không thể tránh trong lập trình. Áp dụng SOLID vào công việc bạn sẽ tạo ra các hàm tốt, dễ hiểu hơn. Giúp cho bạn và đồng nghiệp đọc hiểu code của nhau tốt hơn. Dễ thay đổi SOLID giúp tạo ra các module, class rõ ràng, mạch lạc, mang tính độc lập cao. Do vậy khi có sự yêu cầu thay đổi, mở rộng từ khách hàng, ta cũng không tốn quá nhiều công sức để thực hiện việc thay đổi. Tái sử dụng SOLID khiến các lập trình viên suy nghĩ nhiều hơn về cách viết phần mềm, do vậy code viết ra sẽ mạch lạc, dễ hiểu, dễ sử dụng.

**3.conhesion vs .. la gi.**

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