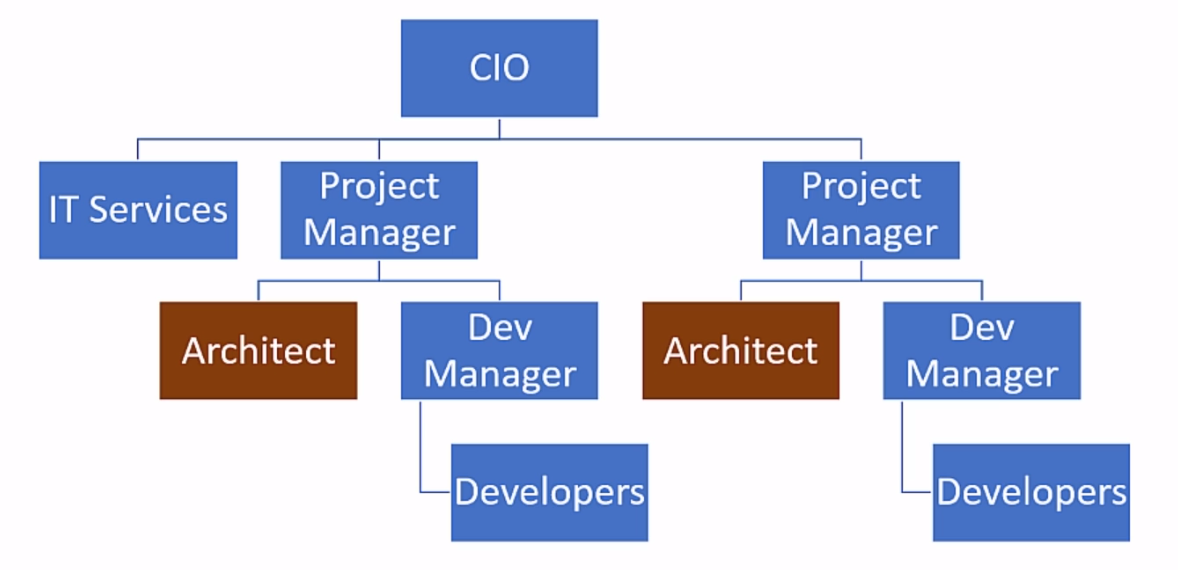
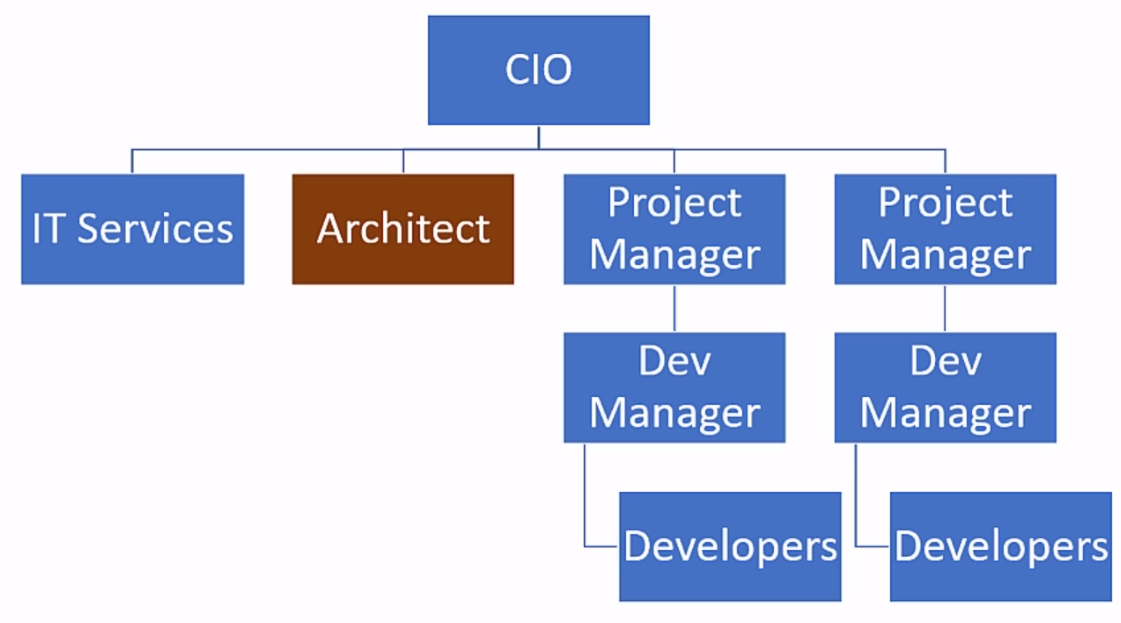
**Section 2: What is SA?**

1. Vậy SA là gì?
   1. Types or Architects:
      1. Infrastructure Architects (kiến trúc cơ sở hạ tầng): design cơ sở hạ tầng như servers, VMss, network, storages … Quen thuộc đối với requirements
      2. Software Architect: in a minute …
      3. Enterprise Architects: works with Top Level Management như CEO, CIO … Không liên quan gì đến development – oriented tasks, career path: Senior SA / PM
   2. Vậy trách nhiệm của SA là gì?
      1. Devs biết cái gì CÓ THỂ làm được, tuy nhiên Architect biết cái gì NÊN làm.
      2. Baseline Requirements:
         1. Fast
         2. Secure
         3. Reliable
         4. Easy to maintain
      3. Như vậy có thể tổng kết lại, SA là người sẽ design hệ thống để đạt được các tiêu chí như Fast, secure, reliable và dễ dàng maintain, đồng thời lựa ra công nghệ và nền tảng tối ưu và patterns hợp lý để đạt được múc đích cuối cùng.
2. The Architect in the Organizational Chart
   1. Architect là một role đặc biệt và phải xử lý rất nhiều thứ, cần phải nắm rõ rằng, Architect sẽ không liên quan đến những management tasks.



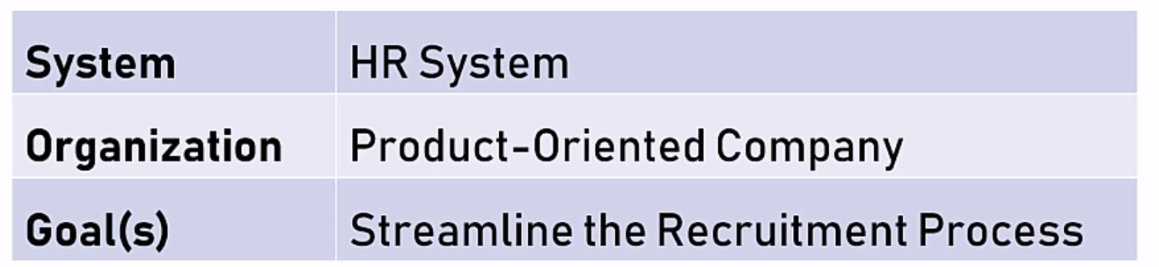


1. Architects & Code
   1. Architects nên có coding skills để cho phép he / she có thể test được các công nghệ mới, create các policies hoặc chứng mình một số concept của code.
      1. Là một SA, bạn sẽ phải design structure của platform là pattern nhất định, do đó cần có coding skill để có thể tự implement một phần của công nghệ để tự kiểm chứng về sự hiệu quả của nền tảng hay patterns mà mình đưa ra.
      2. Support devs:
      3. Nhận được sự tôn trọng (respect)
   2. Architects nên là người đưa ra các pattern dựa trên requirement

**--------------- --------------- --------------- --------------- --------------- ---------------**

**Section 3: The Architect's Mindset**

1. Understand the Business:
   1. Make yourself familiar with:
      1. Điểm mạnh
      2. ĐIểm yếu
      3. Đối thủ cạnh tranh
      4. Chiến lược phát triển
2. Define the system’s goals
   1. Đây không phải là câu trả lời cho câu hỏi “what the system should do?”, câu hỏi này sẽ thiên về trả lời cho requirements nhiều hơn
   2. Goals miêu tả effect của organization
   3. Thường được described bởi Client







1. Làm việc với Client’s clients
2. Talk to the right People with the Right language

**--------------- --------------- --------------- --------------- --------------- ---------------**

**Section 4: The Architecture Process**

1. Understand the System Requirements:
   1. Mặc dù chúng ta đã nói đến việc phải hiểu được system’s goals, tuy nhiên architectural process sẽ được bắt đầu với requirements = what the system should do?
   2. Thường được bắt đầu với những high-level requirements
   3. Thường được defined bở System Analyst
2. Understand the Non-Functional Requirements:
   1. Define technical và service level attributes
   2. Không phải lúc nào cũng được aware bởi System Analyst hay từ phía khách hàng
   3. Thường quan trọng hơn rất nhiều so với các regular requirements (thường là các functional requirements)
3. Map the Components
   1. Components là các thành phần của hệ thống.
   2. Giúp hiểu được các chức năng của hệ thống
   3. Communicate ý hiểu của bạn đối với khách hàng.
4. Select technology stack
5. Design the architecture
6. Write the architecture document
   1. Đây là tài liệu mô tả Process và architecture
   2. Phải là một tài liệu có thể đọc được, hiểu được và liên quan với tất cả những người involved với hệ thống, mỗi người đều có thể tìm thấy những thứ có values nằm bên trong tài liệu này.
7. Support the team

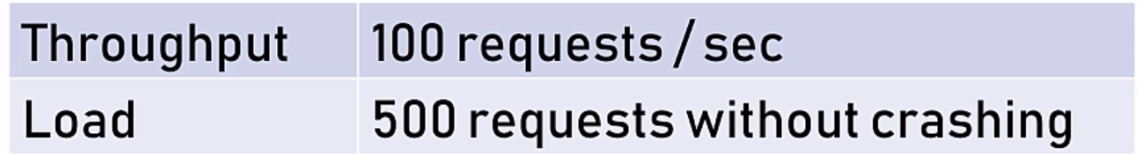
**--------------- --------------- --------------- --------------- --------------- ---------------**

**Section 5: làm việc với System Requirements**

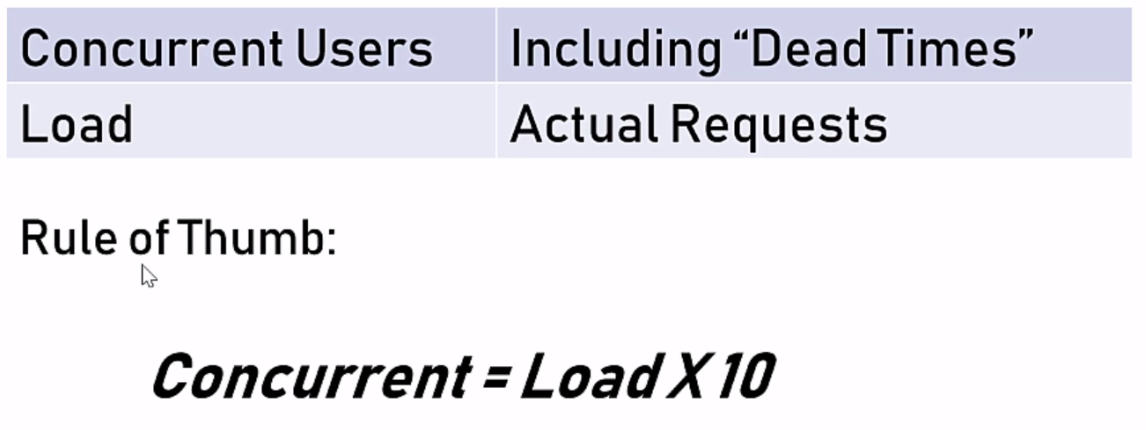
- Có thể hiểu requirement là bất cứ cái gì mà user hoặc customer cần có trong sản phẩm của mình

- Section này sẽ nói về 2 kiểu requirements và cách mà chúng ta được sử dụng bên trong Architecture.

1. Functional requirements (FR):
   1. Trả lời cho câu hỏi “What the system should do?”. Nó có thể là
      1. Business flows
      2. Business Services
      3. UI
      4. …
2. Non-functional requirements (NFR):
   1. Trả lời cho câu hỏi “What should the system deal with?”. NFR miêu tả những thuộc tính mà hệ thống cần phải có, tuy nhiên những yêu cầu này không bị trói buộc vào những hành vi hay logic nào đó của hệ thống. Dưới đây là 5 NFR mà chúng ta thường phải quan tâm nhất trên thực tế (vẫn còn nhiều hơn)
      1. Performance:
         1. Luôn luôn phải có những con số để nói lên performance mong muốn
         2. Độ trễ và thông lượng
         3. Độ trễ (latency): trả lời cho câu hỏi “how much time does it tack to perform a single task?”
         4. Throughput: trả lời cho câu hỏi “how many tasks can be Performed in a given time unit?”
      2. Load
         1. Nghe có vẻ load gần giống với throughput, nhưng đây là 2 khái niệm khác nhau, xem qua ví dụ:



* + 1. Data volumn:
       1. requirement này defined how much data (theo GB hoặc TB hoặc một đơn vị cụ thể nào đó) có thể được lưu trữ theo thời gian
       2. Requirement này rất quan trọng để chúng ta có thể
          1. Lựa chọn được kiểu database type thích hợp
          2. Thiết kế queries
          3. Storage planning
    2. Concurrent User (người dùng đồng thời)
       1. Trả lời cho câu hỏi “how many users will be using the system simultaneously?”



* + 1. SLA (service level agreement)
       1. Required uptime for the system
  1. NFR là những requirements thực sự cần phải quan tâm nếu chúng ta là một SA.

**--------------- --------------- --------------- --------------- --------------- ---------------**

**Section 6: Types of Application**

- Được quyết định sau khi requirements are set.

* Các application types thường gặp:
  + Web Apps
  + Web API
  + Mobile
  + Console
  + Service
  + Desltop

1. Web Apps:
   1. Phù hợp nhất cho các hệ thống yêu cầu
      1. UI
      2. User initiated Actions
      3. Large Scale
      4. Short, focused actions
   2. Request –Response based
2. Web API: gân tương tự với web app tuy nhiên vẫn có các khác biệt như sau:
   1. Không có UI, Web API sẽ public các API để các chương trình khác có thể access và execute các actions
   2. Thường được dựa trên REST based, là một sự kết hợp của
      1. URL
      2. Parameters
      3. HTTP Verb
   3. Web API sẽ return về dữ liệu, không phải là HTML (cả trang web)
   4. Very accessible
   5. Phù hợp với những hệ thống:
      1. Data retrieval and store (truy xuất và lưu trữ dữ liệu)
      2. Client initiated actions
      3. Large scale
      4. Short, focused actions
   6. Cũng thuộc loại request – response based
3. Mobile:
   1. App chạy trên mobile phone
   2. Phù hợp với những hệ thống
      1. UI (games, social appss)
      2. FE for web API
      3. Location
4. Console applications:
   1. No fancy UI (toàn là các log)
   2. Require techinical knowledge
   3. Limited interaction
   4. Long or short running processes
   5. Phù hợp với các hệ thống
      1. Long – running processes
      2. Short actions by trained power users
5. Service:
   1. Gần tương tự với console:
      1. No UI at all
      2. Được quản lý bởi service manager
   2. Phù hợp với những hệ thống yêu cầu:
      1. Long processes (ví dụ: monitoring folders và processing files)
6. Desktop:
   1. Chạy trên PC, has all its resources on the PC
   2. Có thể connect to the web,
   3. Thường có fancy UI
   4. Ví dụ: word, excel …
   5. Thường phù hợp với để xây dựng các ứng dụng
      1. Gaming
      2. User centric actions (Word, excel … )

**--------------- --------------- --------------- --------------- --------------- ---------------**

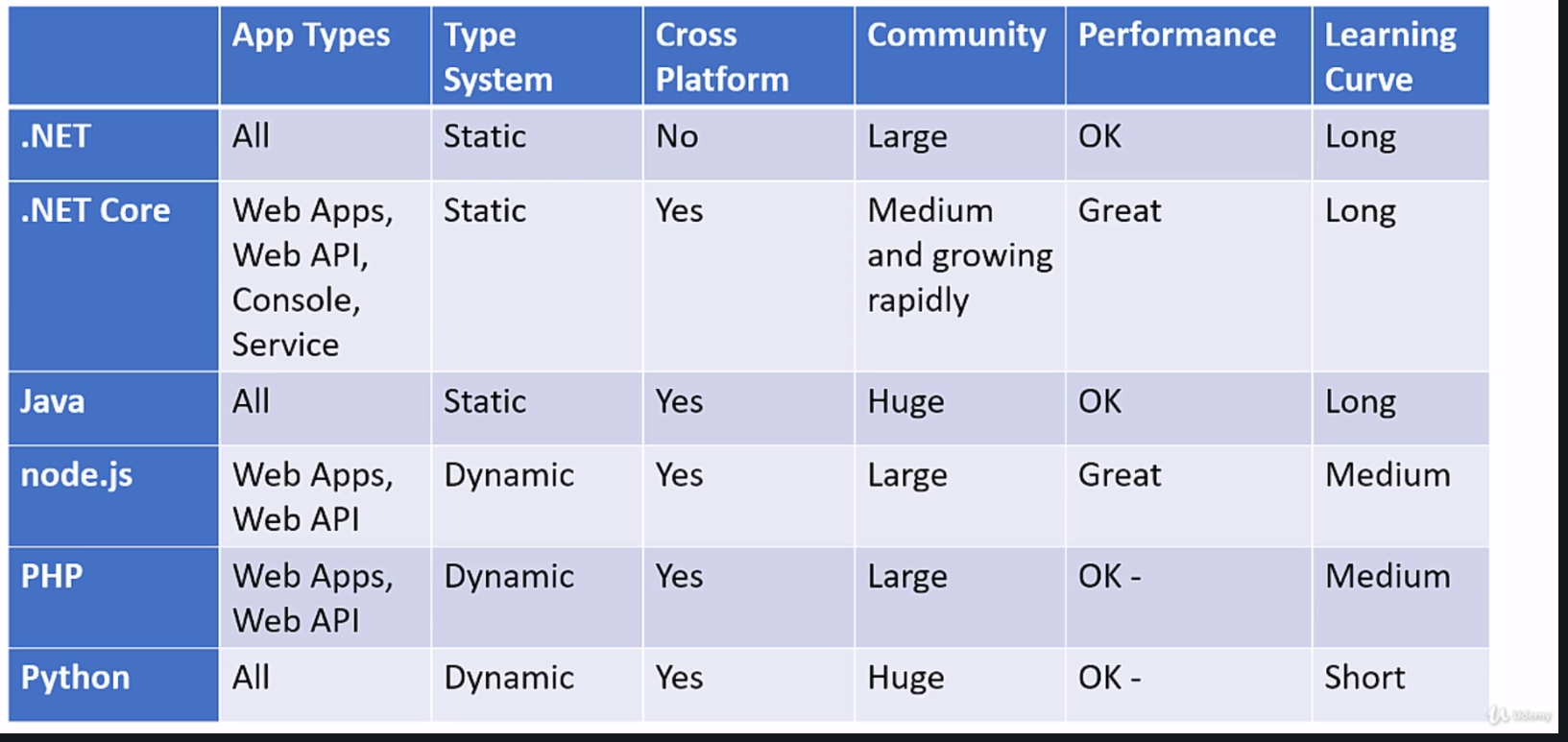
**Section 7: Selecting technology stack**

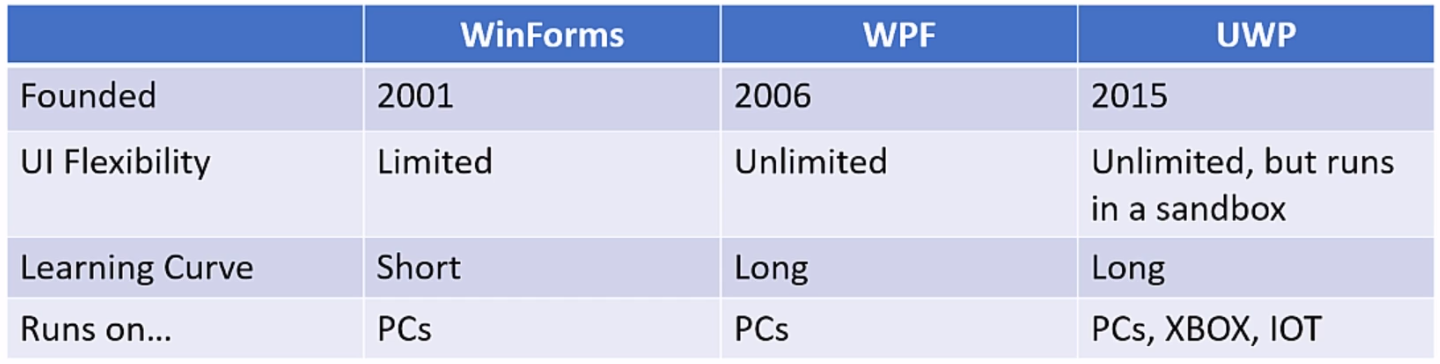
- Cực kỳ quan trọng, vì nó sẽ ảnh hưởng đến công nghệ, nền tảng được sử dụng trong hệ thốn, skills mà development team phải có để implement code …

- Đây là thứ gần như không thể thay đổi

- Quyết định được đưa ra phải được dựa trên:

* Made with clear mind
* Heavily documented
* Group effort





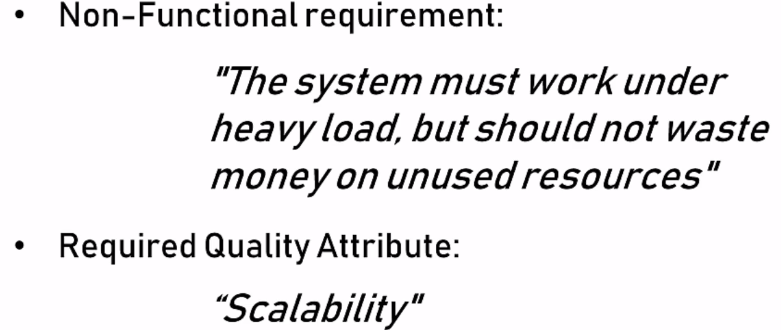
**--------------- --------------- --------------- --------------- --------------- ---------------**

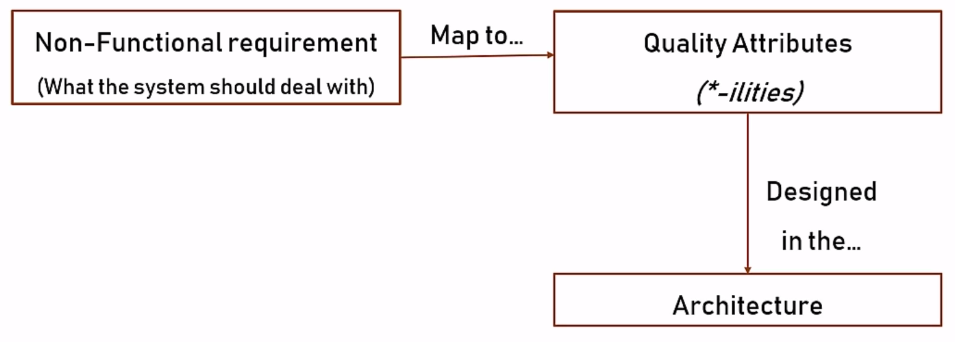
**Meet the \*-ilities**

**- \*-ilities** = quality attributes

- Technical capabilities (năng lực kỹ thuật) that should be used in order để đáp ứng được các NFR

- Các **\*-ilities** phổ biến mà chúng ta sẽ discuss bao gồm: scalability, manageability, modularity, extensibility, testability



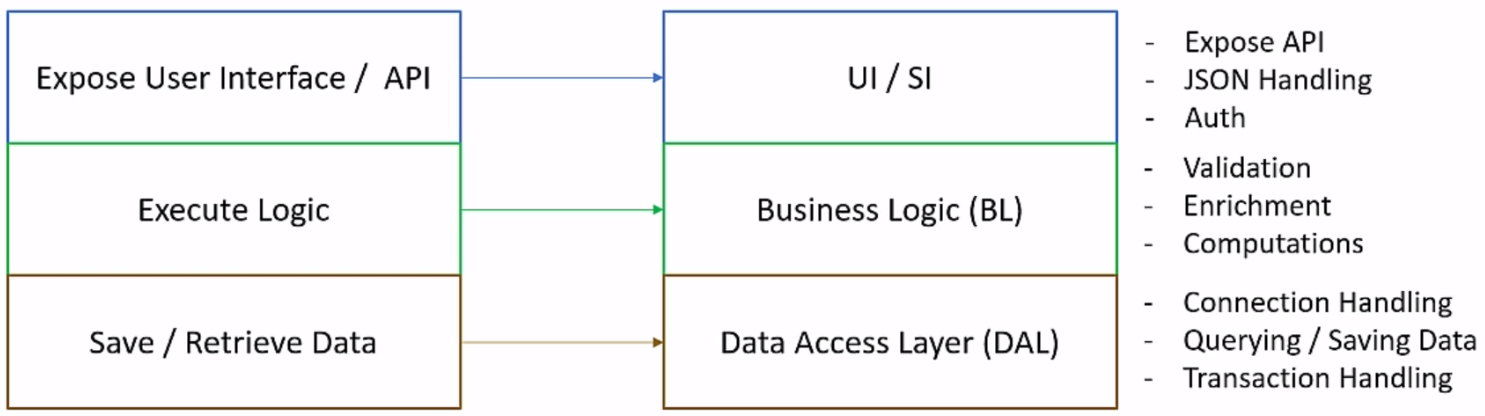


1. Scalability:
   1. Là khả năng mà ứng dụng có thể hỗ trợ việc thêm các resources mà không có bất cứ gián đoạn nào.
   2. Non-scalable system:
      1. Look for non-scalable code
      2. Rewrite non-scalable code
   3. Scalable system:
      1. Add VM
      2. Notify the load balancer
2. Manageability
   1. Là khả năng manage the application, hay có thể nói là “khả năng biết được hệ thống đang chạy như thế nào và có thể đưa ra các actions tương ứng”
3. Modularity:
   1. Modular system là system được xây dựng dựa trên các blocks, có thể thay đổi hoặc thay thế mà không làm ảnh hưởng đến toàn hệ thống, đồng thời tính tái sử dụng của các blocks này cũng rất tốt.
4. Extensibility
   1. Là khả năng một hệ thống có các chức năng có thể được mở rộng mà không cần phải chỉnh sửa hay gây ra các thay đổi ở những đoạn code cũ.
5. Testability:
   1. Là mọt attribute choi biết hệ thống được xây dựng có thể dễ dàng được kiểm thử hay không.

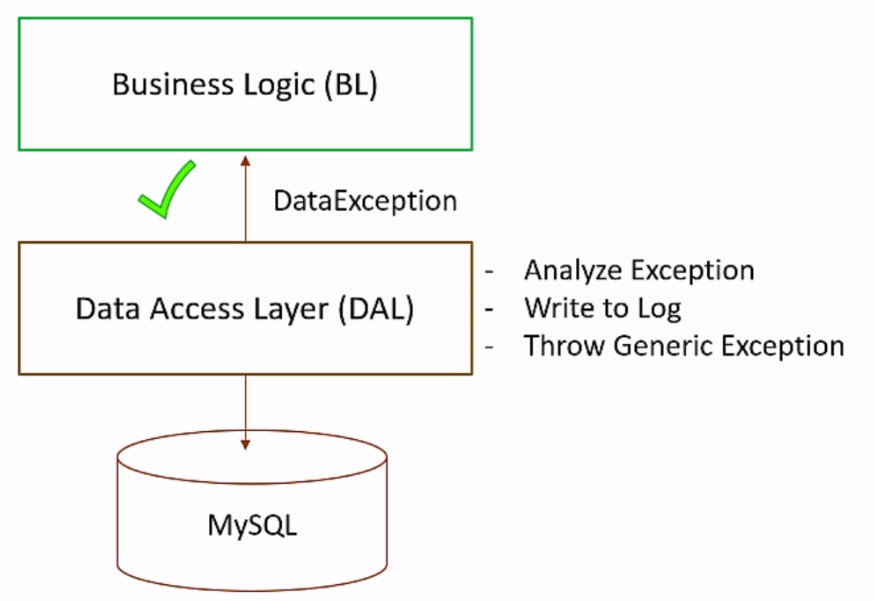
**--------------- --------------- --------------- --------------- --------------- ---------------**

**Section 9: Components’s architecture**

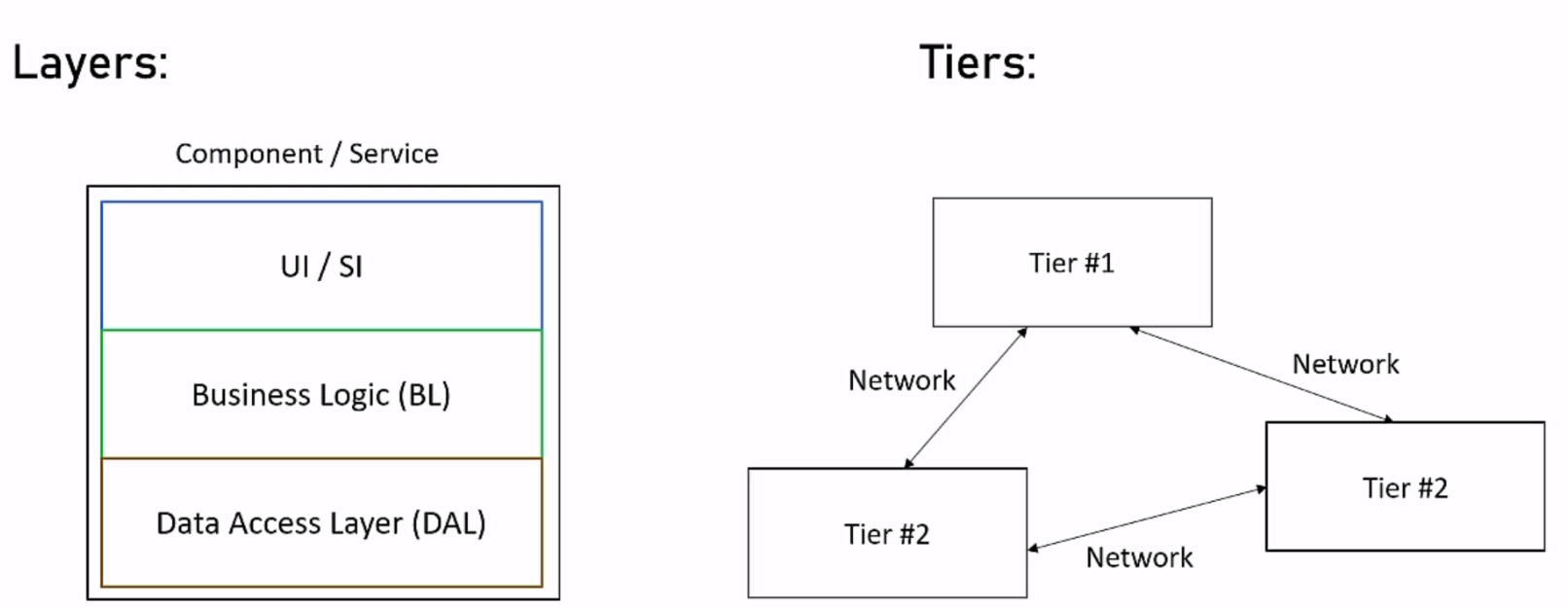
1. Software component là gì?
   1. Còn có thể gọi là Service
   2. Là một đoạn code được chạy trong 1 luồng (single process)
   3. Các hệ thống hiện đạ đều là các hệ thống phân tán (distributed systems), có nghĩa là nó được tạo nên bởi các component riêng biệt, được deployed ở các process khác nhau
   4. Component’s architecture: (level 1, mục tiêu của section này)
      1. Inner components
      2. Sự tương tác giữa các components
      3. Giúp hệ thống chạy nhanh hơn và dễ dàng bảo trì
   5. System architecture (level 2)
      1. Bigger picture
      2. Có thể mở rộng (scalable), đáng tin cậy (reliable), nhanh và dễ dàng bảo trì
2. Layers:
   1. Layer là một hiện thân cho chức năng được thể hiện một cách horizontal (represent horizontal functionality), có thể xem ví dụ sau để hiểu hơn:



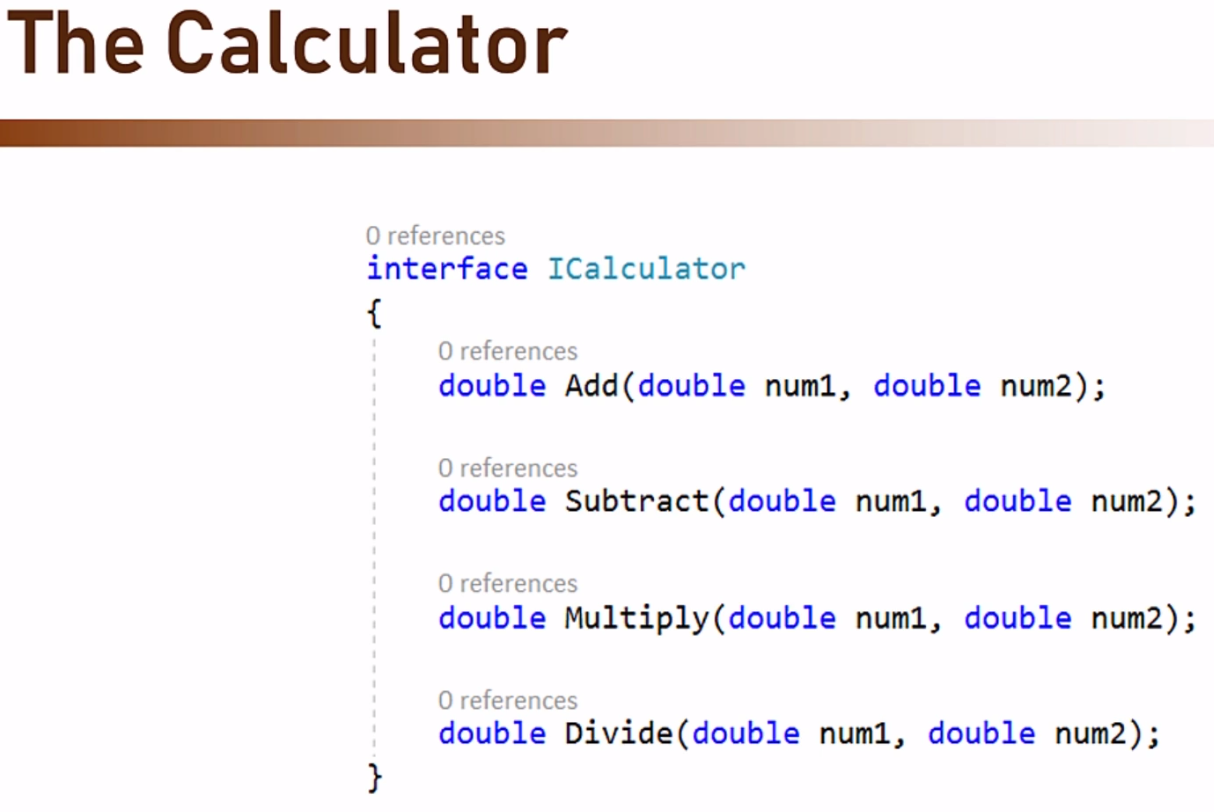
* 1. Mục đích của layers:
     1. Để định hướng code của chúng ta có đúng định dạng và tập trung vào các chức năng nhất định và mục đích rõ ràng
     2. Modular
  2. Concepts of layers:
     1. Code flow: một layer chỉ có thể được gọi là một layer nếu như nó chỉ có thể call đến 1 layer khác ở dưới nó, ví dụ như US / SI ở trên sẽ chỉ call đến BL layer mà thôi, chứ nó sẽ không call directly đến DAL layer được, và nó là 1 chiều, có nghĩa là BL layer sẽ không call ngược lại lên UI / SI layer được.
     2. Loose coupling: có thể hiểu là chúng ta sẽ làm thế nào để layer của chúng ta communicate với các layer khác theo cách mà nó sẽ ít xảy ra impact hết mức có thể nếu như có sự thay đổi
        1. Chúng ta có thể đạt được điều này bằng **Dependency Injection.**
     3. Exception handling: đây là concept liên quan đến việc mỗi layer sẽ giấu đi những inner exceptions và không để cho layer ở bên trên nó “biết” về exception này, mà để cho layer này hiển thị một generic non-specific error message

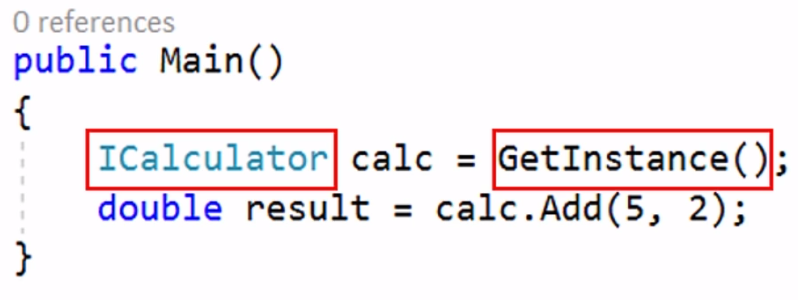


* 1. Layers vs Tiers

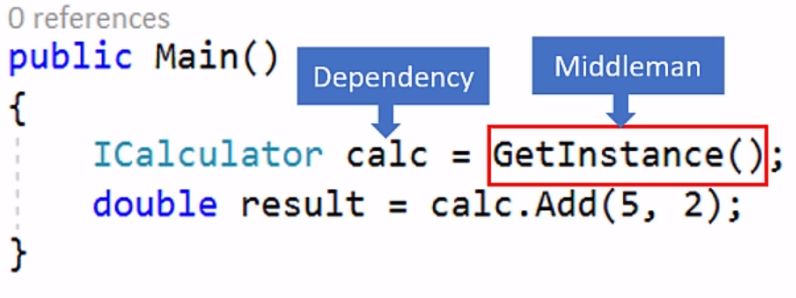


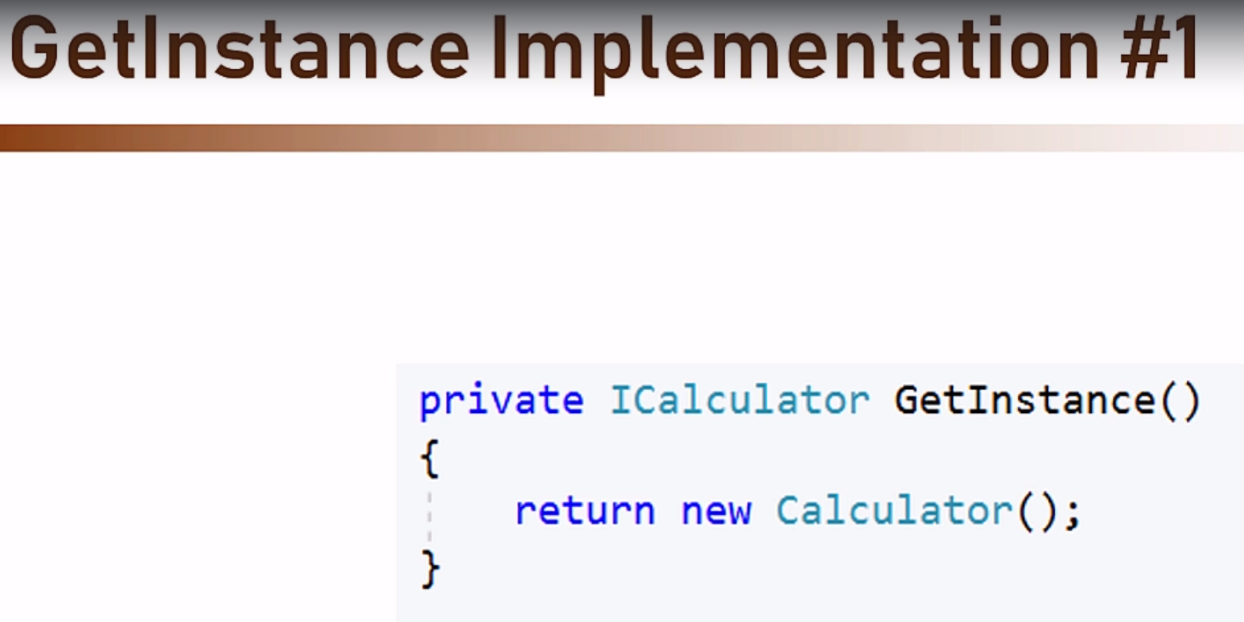
1. Interface:
   1. Có thể hiểu interface là một contract
   2. Interface sẽ khai báo các signatures của một implementations

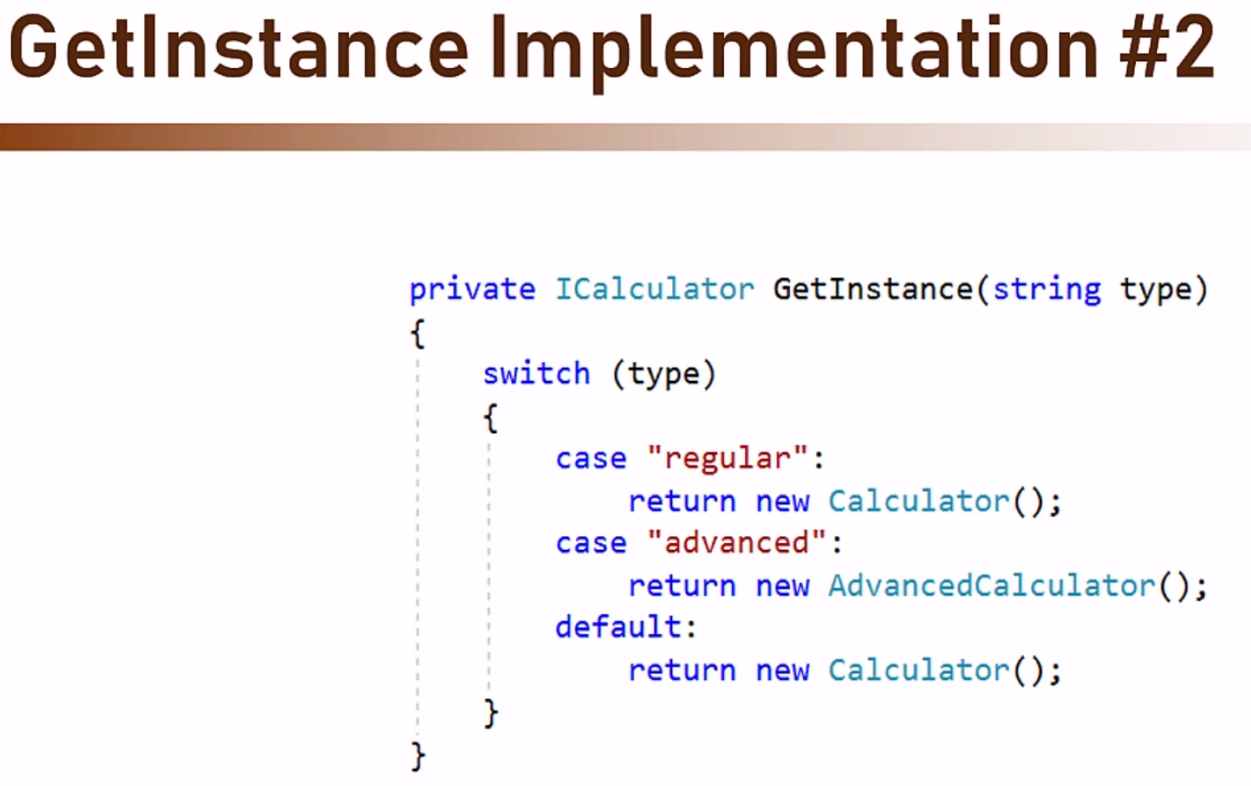


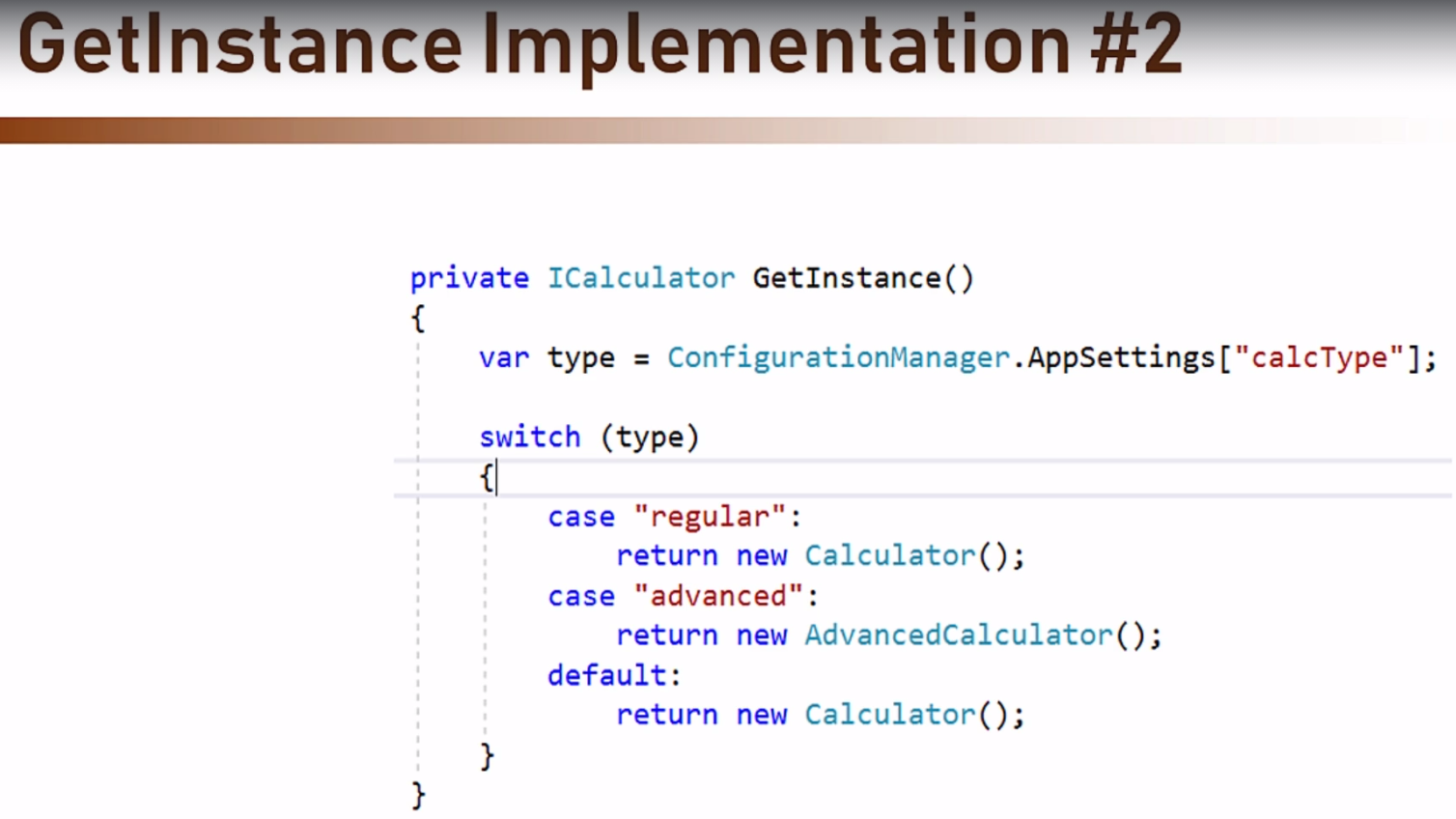


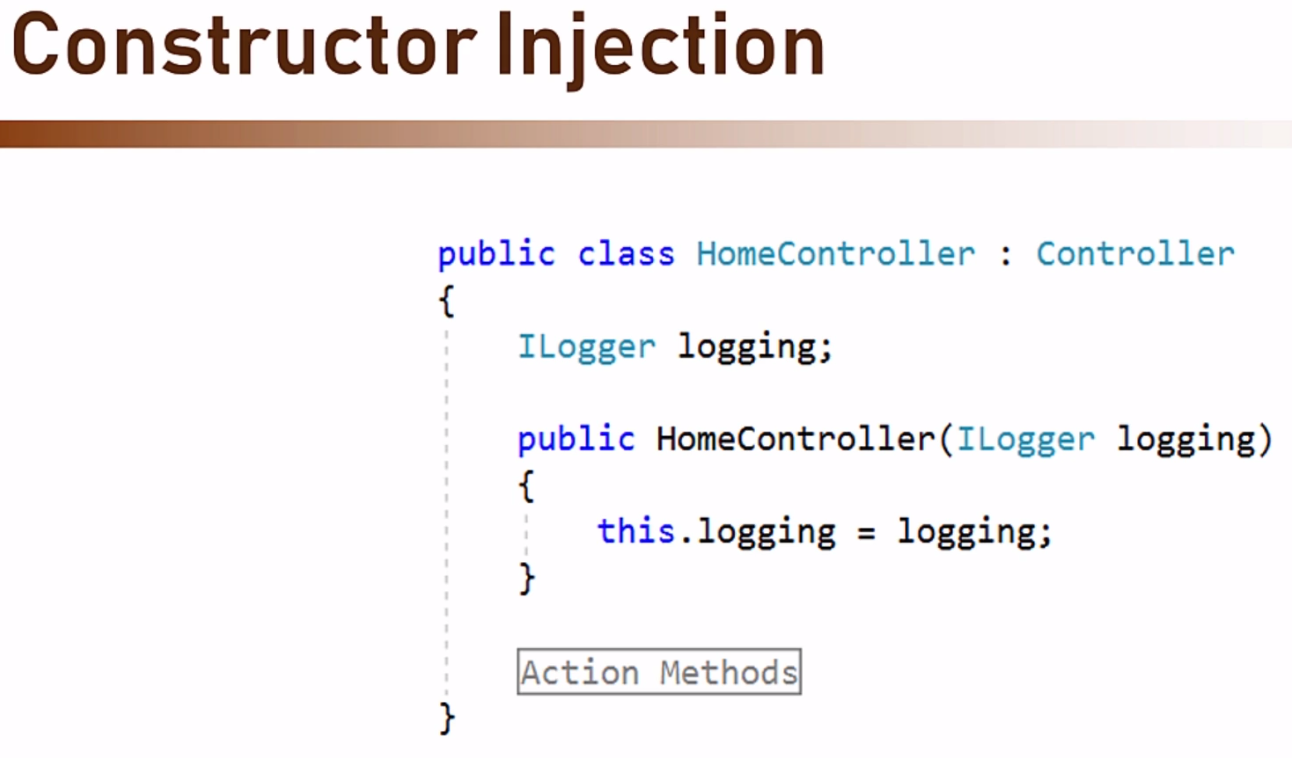
1. Dependency Injection:
   1. Bổ sung cho interface pattern mà chúng ta đã nói tới
   2. DI là một technique mà nhờ đó một obj có thẻe cung cấp các dependencies cho mọt object khác.
   3. Giúp cho code trở nên modular, linh động hơn và dễ dàng maintain



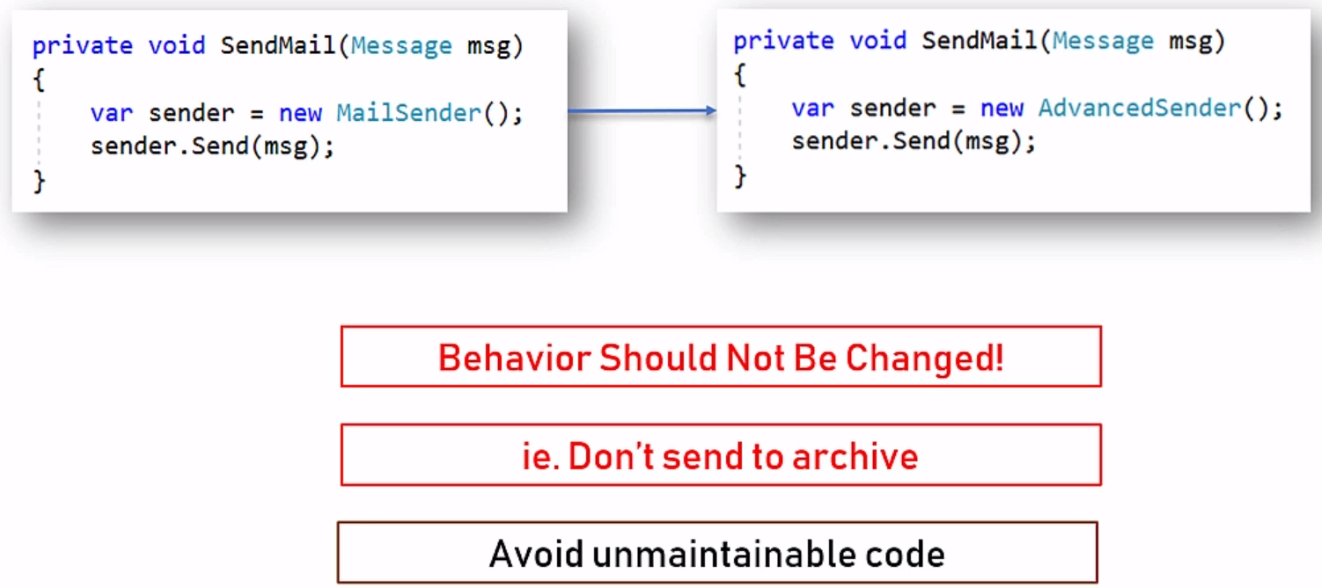




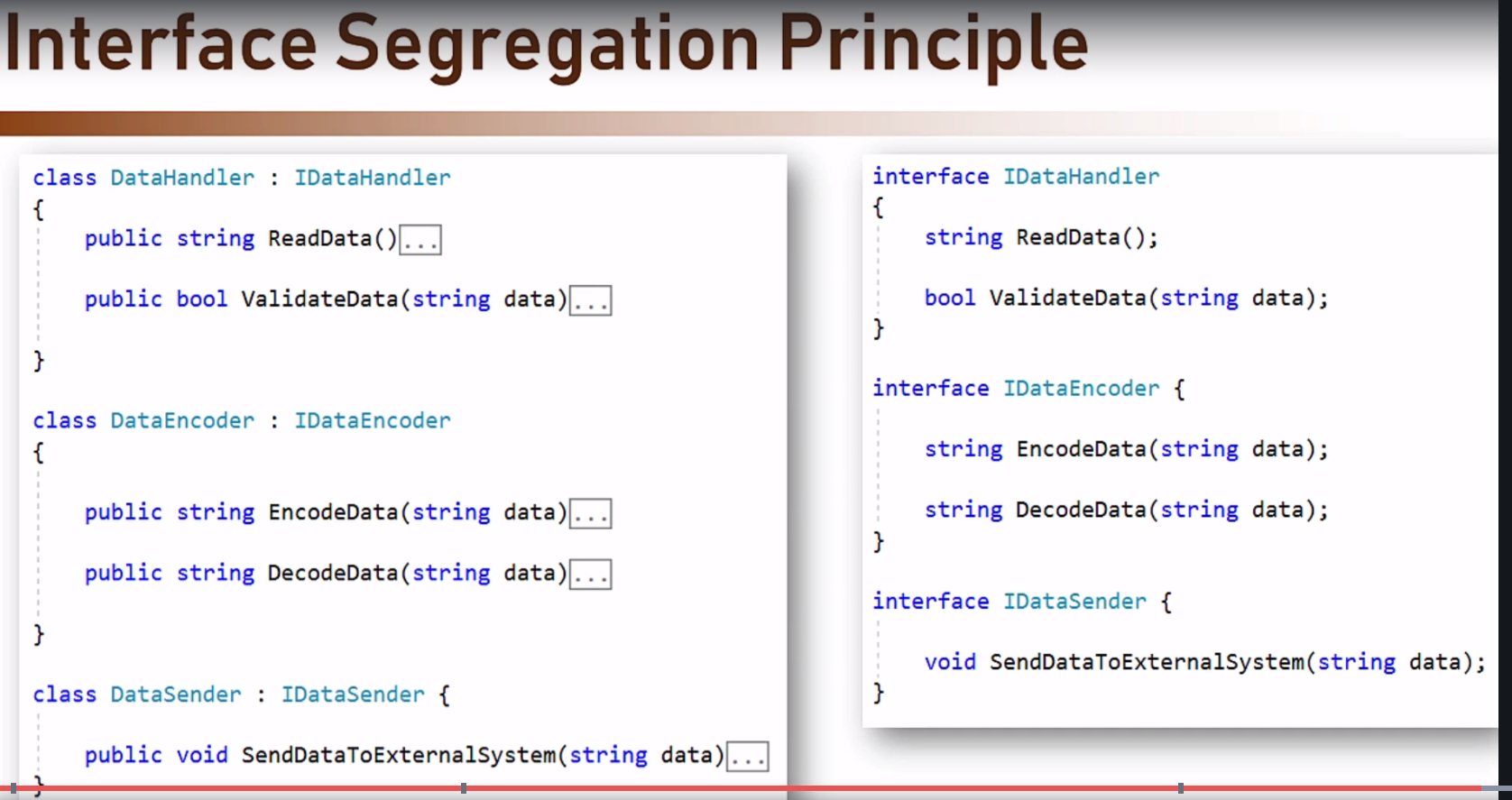




1. SOLID:
   1. Là viết tắt của 5 chữ tương ứng với:
      1. Single responsibility principle
         1. Mối class, module hoặc method chỉ nên có một và chỉ một mục đích duy nhất.
      2. Open/close principle
         1. Một software entity chỉ nên open để extension (mở rộng) nhưng closed for modification
         2. Sử dụng đến tính kế thừa (class inheritance)
         3. Plug-in mechanism
         4. …
      3. Liskov substitution principle
         1. Nếu như S là một type của T, thì object của type T có thể được replaced với object của type S, mà không làm thay đổi bất cứ thuộc tính nào của ứng dụng

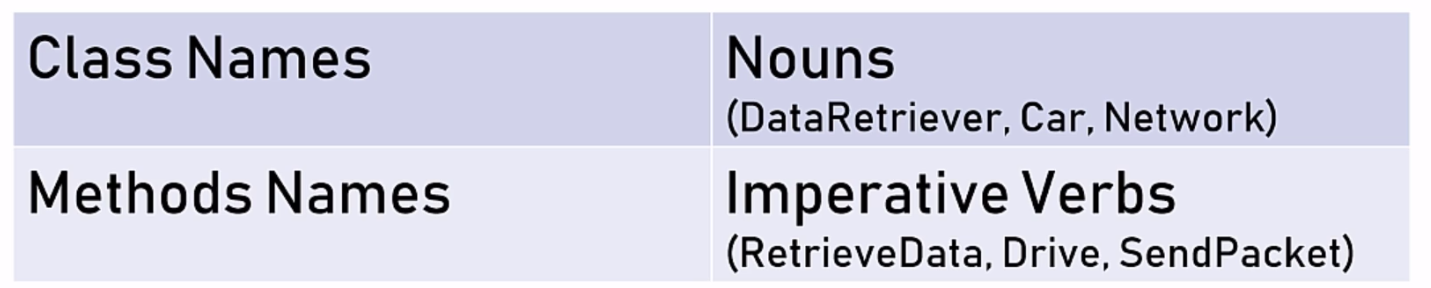


* + 1. Interface segregation principle
       1. Many client – specific interfaces sẽ tốt hơn là một interface có 1 múc đích chung (one general – purpose interface)



* + 1. Dependency inversion principle: chính là DI mà chúng ta đã biết.

1. Naming conventions
   1. Là một tập hợp các rule để chúng ta đặt tên cho các elements trong code
   2. Giúp code trở nên dễ đọc và dễ hiểu hơn.
   3. Không được thực hiện bởi trình biên dịch (phải do devs tự defined)
   4. Có 2 dạng:
      1. Structure (casing, underscores, …)
      2. Content
   5. Các dạng cơ bản:
      1. camelCase:
      2. lowercase\_seprated\_by\_underscore
      3. CAPITALIZED\_WITH\_UNDERSCORE
      4. Hungarian Notation: type information sẽ là 1 phần của tên, ví dụ: **string strFirstName**
   6. Content:



1. Exception handling:
   1. Chỉ catch exception nếu như chúng ta có gì đó để xử lý nó
      1. Đương nhiên, logging only sẽ không tính là actions
   2. Chỉ catch các exception riêng biệt, có thể hiểu là catch exception phải đúng lý do, nguồn gốc gây ra của exception
   3. Sử dụng try –catch block ở những code block nhỏ nhất có thể.
      1. Để catch error một cách chính xác, vì có thể có quá nhiều code là exception gây ra có thể khác nhau.
2. Logging:
   1. Logging giúp tracking errors
   2. Thu thập data (gather data)
      1. Xác định module nào được vistied nhiều nhất?
      2. Store performance data và tìm hiểu xem method nào hoạt động kém hiệu quả …
      3. Xác định user’s flow
   3. Không quan trọng chúng ta save log ở
      1. Files
      2. Database
      3. Event log