# Chap 2: Tên có ý nghĩa

## Sử dụng tên bộc lộ mục đích

# Tránh sai lệch thông tin

Tránh đặt những gợi ý sai lầm làm lu mờ ý nghĩa thực sự của code. Ví dụ như viết tắt từ này có thể hiểu nhầm sang từ khác.   
ví dụ:

## Làm cho sự riêng biệt trở lên rõ ràng

Cách đặt tên biến theo kiểu a1, a2 , .. aN tạo ra sự phân biệt nhưng đó thực sự là một cách đặt tên biến tồi.

## Sử dụng tên đọc được

Với các tên không thể phát âm gây khó chịu và rắc rối khi trao đổi giữa các thành viên

## Sử dụng tên tìm kiếm được

Đặt tên một biến có thể tìm kiếm giúp bạn rút ngắn thời gian tìm kiếm tên biến, hàm và lớp đó một cách dễ dàng.

## Avoid Encodings

## Avoid Mental Mapping ( Tránh ánh xạ tinh thần)

## Class name

Tên class nên đặt là danh từ hoặc cụm danh từ

## Method name

Tên phương thức nên đặt là động từ hoặc cụm động từ

## Don’t be cute (Không đặt tên cute, chơi chữ)

## Pick one Word per Concept( Chọn một từ cho mỗi khái niệm)

## Thêm ngữ cảnh có ý nghĩa

## trong phương thức làm việc với địa chỉ , các tên biến như firstName, lastName, state,… có thể làm cho bạn dễ hiểu đó là các biến để lưu thông tin về địa chỉ. Tuy nhiên khi đứng một mình state liệu bạn có dám chắc là nó để lưu thông tin về địa chỉ. Do đó bạn có thể gắn thêm các thông tin về ngữ cảnh cho nó để làm cho người đọc thực sự hiểu nó hơn. Như là addrFirstName, addrLastName.

## 

## Không chơi chữ

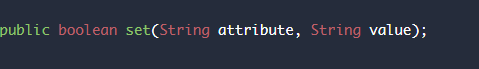
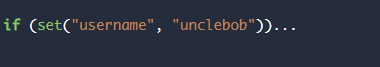
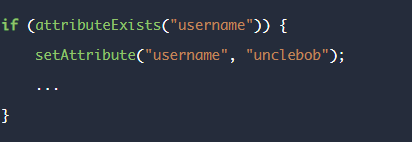
## Use Solution Domain name ( dùng tên của miền giải pháp)

## Use Problem Domain name(Dùng tên của miền nghiệp vụ)

## Don’t Add Gratuitous Context ( Đừng thêm những bối cảnh vô căn cứ)

## Final Words

# Chap 3: Function

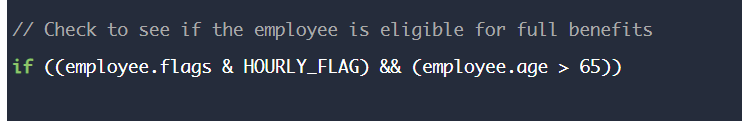
1. Small !!!  
   Nguyên tắc đầu tiên viết hàm là hàm đó nên nhỏ và nguyên tắc thứ 2 là hàm đó nên nhỏ hơn nữa.   
   Block and Indenting : Những khối lệnh if, else, while nên được chứa trong một dòng và những dòng này nên đặt thành một lời gọi hàm. Điều này không chỉ giữ cho function nhỏ mà còn cho biết thêm thông tin cụ thể việc nó đang làm bằng một cái tên mô tả độc đáo
2. Do one thing !!! (Làm một việc)   
   Hàm nên được rõ ràng và chỉ nên đảm nhiệm làm 1 thứ.
3. One Level of Abstraction per Function  
   Để chắc chắn rằng 1 hàm làm 1 việc , bạn chri nên gọi 1 level hàm trong 1 hàm. Nghĩa là bạn không nên gọi hàm trong hàm trong một hàm.
4. Switch Statement
5. Sử dụng tên mô tả những gì function làm.  
   Không ngại bởi vì nó là cái tên dài. Tên mô tả dài tốt hơn là một tên ngắn bí ẩn. Tên mô tả dài tốt hơn là một đoạn bình luận mô tả. Sử dụng một quy tắc đặt tên dùng những từ ngữ dễ đọc để làm tên hàm và dùng những từ khóa để mô tả chức năng của nó làm gì.
6. Số lượng tham số lý tưởng trong 1 hàm là 0, 1 sau đó là 2, 3 tham số nên tránh nếu có thể. Nhiều hơn 3 tham số cần sự lí giải đặc biệt.
7. Have No Side Effects(Không có tác dụng phụ)   
   Hàm của bạn hứa hẹn làm một điều , nhưng đôi khi có những phản ứng không mong muốn. Đôi khi nó làm thay đổi bất thường các biến trong class của chính nó. Đôi khi nó để các thông số thông qua và vào hàm hoặc hệ thống toàn cục. Trong cả 2 trường hợp nó sinh ra nhưng hiểu lầm quanh co và tai hại dẫn đến những sự phụ thuộc và liên kết kỳ lạ.  
   Ví dụ: appendFooter (s): Có phải là hàm này nối thêm s vào Footer vào cái gì không? Hay là nối thêm footer vào s? s là input hay output?  
   Sẽ tốt hơn khi gọi appendFooter là : report.appendFooter();
8. Command Querry Separation: Phân tách các lệnh truy vấn  
   Các hàm nên làm một cái gì đó hoặc trả lời một cái gì đó, nhưng không phải cả hai. Hoặc hàm sẽ thay đổi trạng thái của một đối tượng hoặc nó sẽ trả về một số thông tin về đối tượng đó. Làm cả 2 thường dẫn đến nhầm lẫn.   
   Ví dụ  
     
   Hàm này sẽ set giá trị cho 1 thuộc tính và trả về true nếu nó thành công và false nếu không có thuộc tính tồn tại.  
     
   Vấn đề ở đây là : Không biết set dùng theo cách:   
   - Nếu “username” đã tồn tại giá trị “unclebob” trước chưa  
   - Set thuộc tính “username” với gái trị “unclebob”  
   Điều này gây ra nhầm lẫn :  
     
   Ta có thể giải quyết vấn đề này bằng cách đặt lại tên và phân tách các câu lệnh truy vấn để sự mơ hồ không xảy ra:   
   
9. Bạn nên viết hàm như thế nào ?

Viết phần mềm cũng như viết các thứ khác. Khi bạn viết báo hoặc tạp chí. Bạn viết các ý tưởng của bạn đầu tiên, sau đấy chỉnh sửa nó cho tới khi đọc có vẻ ổn. Bản thảo đầu tiên có thể là vụng về và vô tổ chức, vì vậy bạn cơ cấu lại n ó và tinh chỉnh nó cho đến khi nó có thể đọc theo cách bạn muốn nó đọc.

Khi viết 1 hàm, chúng quá dài và phức tạp. Có rất nhiều indenting và các vòng lặp lồng nhau. Nó có quá nhiều đối số. Các tên tùy ý, có các mã lặp lại. Và cũng có các đoạn mã unit tests để test chỗ code lôm côm này.   
Vì vậy, sau đó bạn xem lại và tinh chỉnh đoạn mã đó, tách các hàm nhỏ hơn, thay đổi tên, loại bỏ trùng lặp. Bạn cần thu nhỏ các phương thức và sắp xếp lại chúng, đôi khi bạn cần phá vỡ toàn bộ các lớp đã xây dựng từ trước đó.  
Tổng kết :  
Khi sử dụng switch-case, cân nhắc sử dụng ABSTRACT FACTORY.

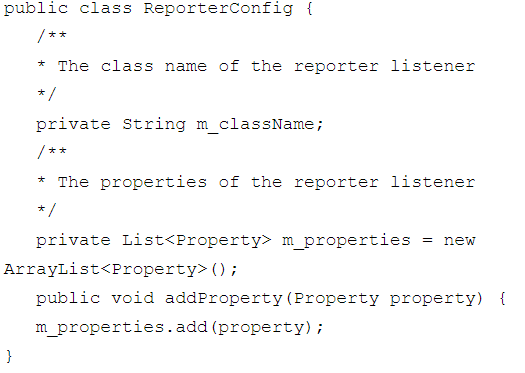
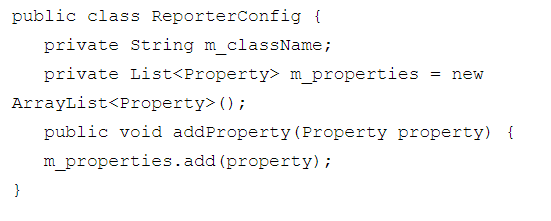
* Cân nhắc và tìm cách giảm đối số truyền vào của 1 hàm, xuống 1 đến 2 là tốt để có thể test và dễ hiểu.
* Tên hàm cũng cực kì quan trọng, tránh để double-take. Tên đẹp nhất là cum verb/noun(cho loại hàm 1 đối số).
* Nguyên tắc mang tính chất tham khảo, tùy mục đích mà bạn có thể áp dụng cho không. Nếu các project không có các đoạn unit tests hoặc không coi trọng clean, bạn có thể tạo ra các hàm với nhiều đối số, cờ để thuận tiện nhất cho project

# Chap 4: Comment

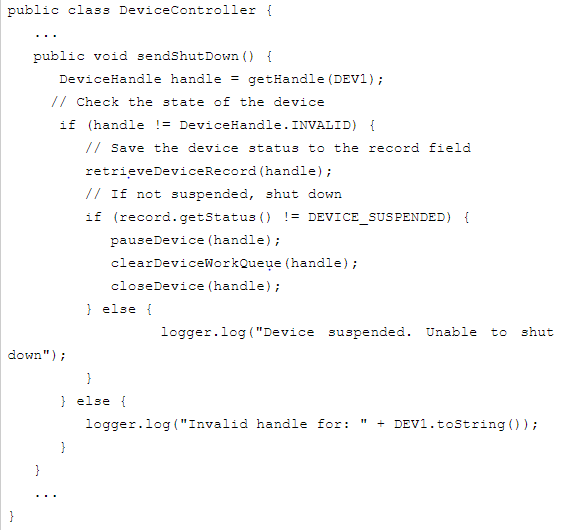
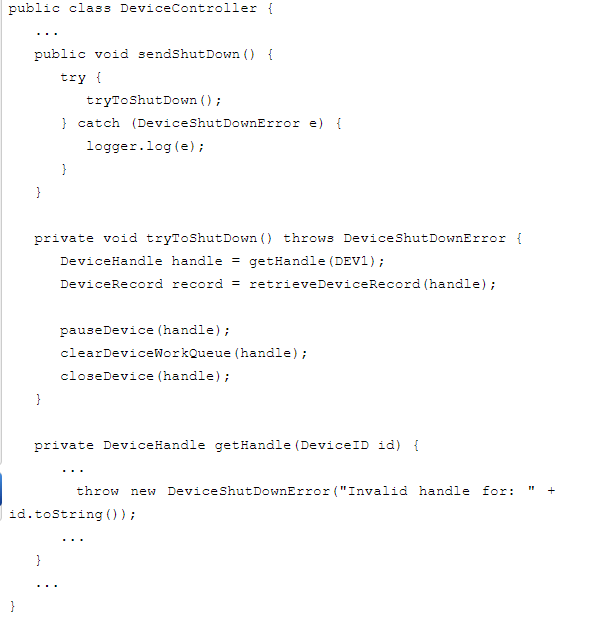
1. Comments Do not Make Up for Bad Code  
   Một trong những lý do chính để bạn comment là bad code. Bạn viết xong 1 đoạn code, bạn đọc lại và thấy nó thật khó hiểu và vô tổ chức. Sau đó bạn thêm một vài đoạn comment vào giải thích cho các đoạn code đó và tự nhủ với mình “Đoạn code đã tốt hơn rồi”. Không, sẽ tốt hơn là bạn nên xóa nó đi. Một đoạn mã tốt với chỉ với một vài comment sẽ tốt hơn rất nhiều so với sự lộn xộn và phức tạp của một đoạn mã với quá nhiều comment.
2. Explain Yourself in Code : Tự giải thích trong code  
     
   Có comment giải thích đoạn if cho bạn. Vậy tại sao ta không để nó tự giải thích như này chẳng hạn  
     
   Chỉ mất vài giây để người đọc hiểu đoạn code muốn đề cập đến vấn đề gì thay vì đọc comment
3. Good Commnet   
   Nhận xét nên hạn chế được đưa vào code, tuy nhiên dưới đây là một số good comment bạn nên đưa vào để bổ sung các thông tin hữu ích cho các đoạn code của bạn.  
   - Comment về phấp lý: Đó là các comment để cho người khác biết ai viết đoạn code đó. Bạn nên comment để cho người khác biết  
   - Các comment chứa thông tin: Đó là các thông tin khá hữu ích, cung cấp các thông tin cơ bản nhất về 1 hàm (chẳng hạn đầu vào, đầu ra)  
   - To Do Commnet: Đó là comment các công việc bạn chưa kịp thực hiện hoặc các chức năng bạn có thể phát triển trong tương lai
4. Tổng kết  
   Hạn chế đưa comment vào code của bạn, hãy cố gắng để các hàm, các biến, các lớp tự nói lên chức năng của nó. Một vài trường hợp “good comment” như comment về pháp lý để chứng thực tác giả, comment tổng quan về chắc năng hàm (input, output), comment để đưa ra lý do quyết định thực hiện quyết định (tại sao lại if, for, while ở chỗ này). Còn lại hầu như là các “bad comment”.

# Chap 5: Fomartting

Khi mọi người nhìn vào, chúng tôi muốn Code nhìn vào được gọn gàng, nhất quán, chi tiết. Muốn mọi người cảm nhận được  đây là do một chuyên gia làm việc. Nếu thay vào đó họ thấy được một khối lượng code xáo trộn trông giống như được viết bởi một loạt thủy thủ say rượu, sau đó họ có thể kết luận rằng sự thiếu chú ý đến chi tiết tràn ngập trong mọi khía cạnh khác của dự án.  
  


1. Vertical Formatting( Theo chiều dọc)   
     
   - The newspaper Metaphor: Chúng ta muốn mã nguồn giống như một bài báo. Tên đơn giản nhưng giải thích rõ ràng. Cái tên cho chúng ta biết có đang ở đúng module hay không. Ở phía đầu cung cấp những khái niệm ở cấp cao và các thuật toán. Chi tiết sẽ tăng thêm khi chúng ta di chuyển xuống dưới. Cho đến khi kết thúc, chúng ta tìm những hàm ở mức thấp nhất và chi tiết của file nguồn.
2. Vertical Openness Between Concept (Sự cởi mở giữa các khái niệm)  
   Gần như tất cả Code đề được đọc từ trái sang phải, từ trên xuống dưới. Mỗi dòng đều đại diện cho một biểu thức hoặc một mệnh đề. *Những khái niệm nên được tách ra với một dòng trắng*. => Tăng khả năng đọc Code
3. Vertical Density (Mật độ)  
   Cở mở chia tách các khai niệm, sau đó mật độ liên kết chặt chẽ lại.  
   => Comment làm phá hỏng sự liên kết:  
     
     
   => Dễ dàng hơn để đọc:  
     
     
   -Vertical Distance (Khoảng cách) : Các khái niệm có liên quan với nhau nên được giữ gần nhau, tránh tách chúng ra thành từng file khác nhau trừ khi có lý do chính đáng.
4. Horizontal (Theo chiều ngang): Cố gắng giữ cho dòng được ngắn. Đừng bao giờ phải di chuyển sang phải
5. Team Rules  
     
   Mỗi lập trình viên đều có một nguyên tắc định dạng riêng. nhưng khi làm việc nhóm phải tuân theo nguyên tắc của nhóm. Một nhóm phát triển nên thỏa thuận về một kiểu định dạng duy nhất, sau đó mỗi thành viên trong nhóm nên sử dụng kiểu định dạng đó. => Tính nhất quán.

# Chap 7: Error Handling

1. Use Exceptions Rather Than Return Codes (Sử dụng ngoại lệ hơn là trả về Codes)  
     
     
     
   Vấn đề ở đây là nó lộn xộn với người gọi. Người gọi phải kiểm tra trực tiếp sau khi gọi. Không may, nó rất dễ bị quên lãng. Vì lý do đó, tốt hơn là cho nó ra một exception khi bạn gặp phải lỗi. Nó làm mã được sạch hơn và logic không bị che khuất bởi xử lý lỗi.  
     
     
     
     
     
     
     
     
   Code tốt hơn bởi vì có hai quan hệ lộn xộn ở đây, thuật toán cho thiết bị tắt máy và xử lý lỗi, và bây giờ nó đã được tách ra.

# Boundaries

* Custom làm đơn giản hóa các API theo yêu cầu của project , -🡪 khi sử dụng các chức năng này mình không cần quan tâm đến thư viện bên thứ 3 mà chỉ cần hiểu các chức năng mà mình đã public theo nhu cầu. Làm đơn giản hóa, ý nghĩa cho các hàm

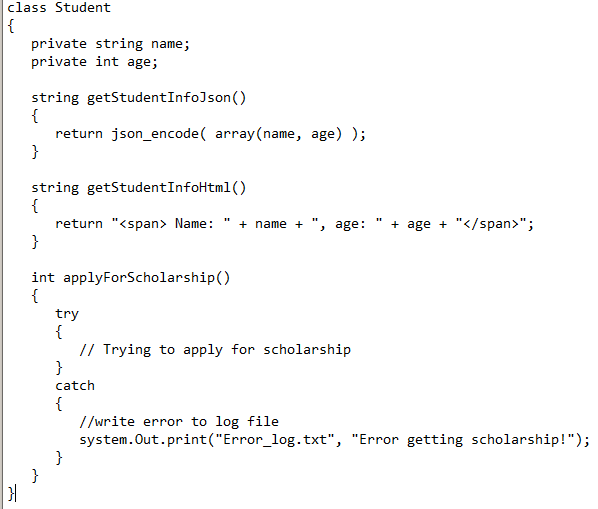
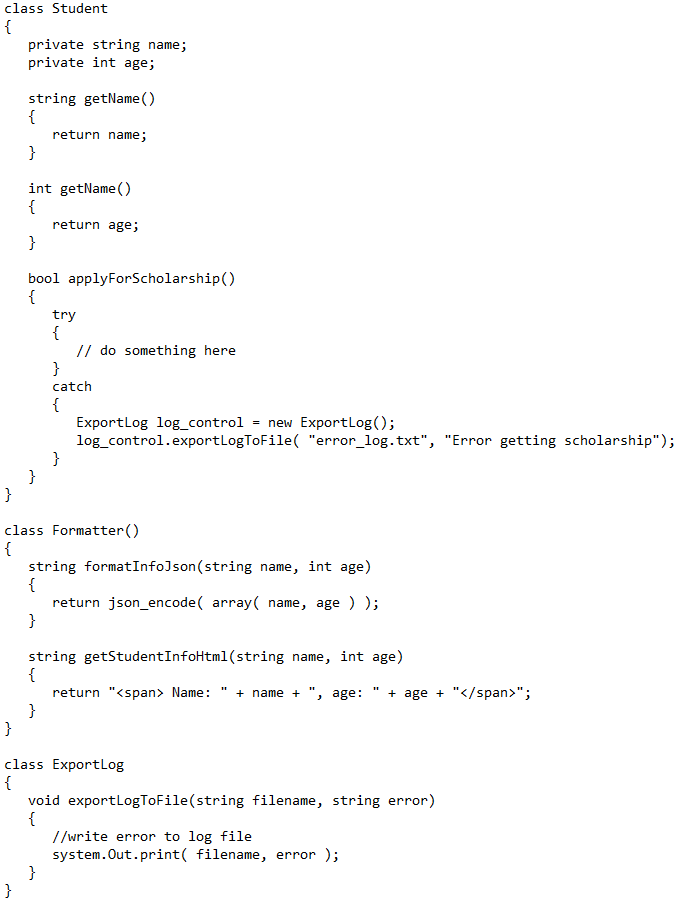
# Units test

1. TDD là gì?  
   Là một quy trình lập trình, trong đó bao gồm nhiều giai đoạn nhỏ lặp đi lặp lại, mỗi giai đoạn bao gồm các bước:   
   - Viết các bài kiểm thử  
   - Viết mã nguồn  
   - Tái cấu trúc mã nguồn   
   Vòng thực thi của TDD  
   - Viết Test :   
   + Fail bởi vì chưa có mã nguồn  
   + Thông thường các IDE sẽ báo message màu đỏ

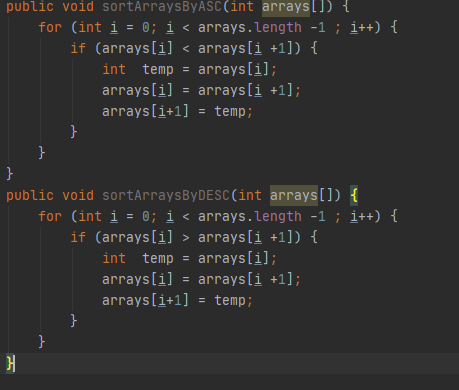
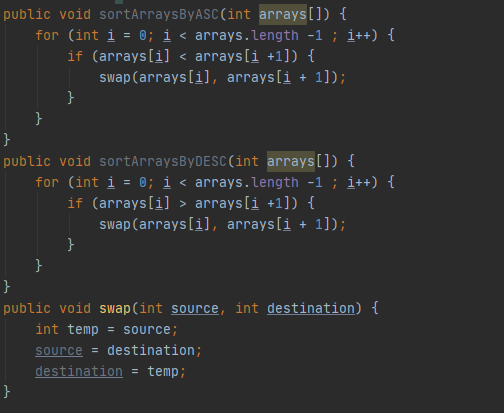
- Viết mã nguồn  
+ Viết mã nguồn để vượt qua test  
+ Thông thường các IDE sẽ báo message màu xanh  
- Tái cấu trúc mã nguồn  
+ Chỉnh sửa mã nguồn để “tốt” hơn  
+ Cần đảm bảo mã nguồn vẫn vượt qua các bài test

1. 3 Luật để làm sao để test  
   -: Không cho phép viết bất kỳ một mã chương trình nào cho tới khi nó làm một test bị fail trở nên pass  
   -: Không cho phép viết nhiều hơn một unit test mà nếu chỉ cần 1 unit test cũng đã đủ để fail. Hãy chuyển sang viết code function để pass test đó trước.  
   - :Không cho phép viết nhiều hơn 1 mã chương trình mà nó đã đủ làm một test bị fail chuyueenr sang pass.

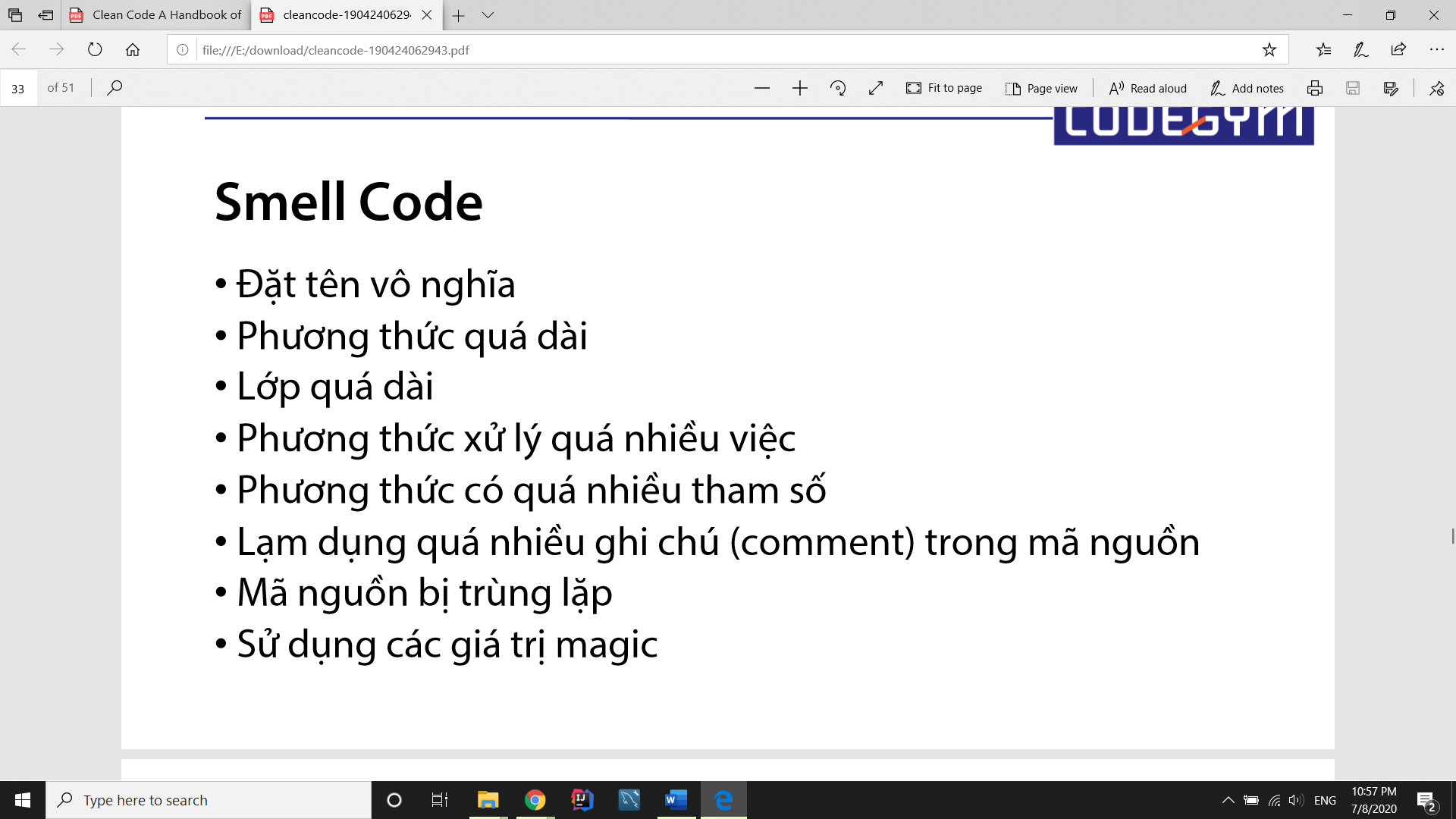
# Chap 10: Classes

1. Class Organization (Tổ chức của Class)  
   Class nên bắt đầu với một danh sách các biến (variables)
2. Classes Should Be Small
3. The Reposibility Principle  
     
   Tất nhiên, đoạn code trên có thể chạy đúng yêu cầu của phần mềm. Thế nhưng! **Dường như lớp Student đang ôm đồm quá nhiều việc**: cung cấp thông tin cá nhân, định dạng cấu trúc của thông tin cá nhân, ghi log, apply xin học bổng ..v.v.. Điều này vi phạm quy tắc đơn nhiệm. Về mặt logic xử lí, có một số câu hỏi mà chúng ta có thể thắc mắc: học sinh thì có nhiệm vụ gì mà phải đi định dạng dữ liệu nhỉ? Sau này nếu muốn định dạng ghi log từ file thành DB thì như thế nào? Đổi định dạng format thông tin log thì sẽ sửa code làm sao? …  
     
   
4. Organizing for Change

# CHAP 12: EMERGENCE

1. Simple Design Rule 1: Runs All The Tests  
   Chạy tất cả các bài kiểm thử
2. Simple Design Rule 2: No Duplication ( Không trùng lặp)  
     
     
   
3. Simple Design Rule 3: Expressive (  
   Code cần thể hiện rõ ý định của người viết : Người khác sẽ mất ít thời gian để hiểu nó , điều này giúp giảm thời gian và chi phí bảo trì.
4. Simple Design Rule 4: Minimal Classes and methods

# SMELL CODE



# 