Cognitive Load In Software Development

Releases · zakirullin/cognitive-load (github.com) (cập nhật 5/2024)

- Cognitive Load

Cognitive Load

Lời nói đầu

Có rất nhiều từ chuyên ngành và phương pháp luyện tập tốt nhưng hãy tập trung vào những điều cơ bản nhất. Điều quan trọng là sự bối rối mà người lập trình cảm thấy khi đọc code.

Chi phí cho sự bối rối đó là thời gian vào tiền bạc. Sự bối rối này tạo ra bởi *tải nhận thức* cao. Nó không phải khái niệm trừu tượng đặc biệt mà là hạn chế cơ bản của con người.

Từ khi chúng ta dành nhiều thời gian để đọc hiểu code hơn là viết nó. Chúng ta sẽ liên tục tự hỏi chính mình rằng liệu chúng ta có đang đưa quá tải nhận thức vào trong code của mình.

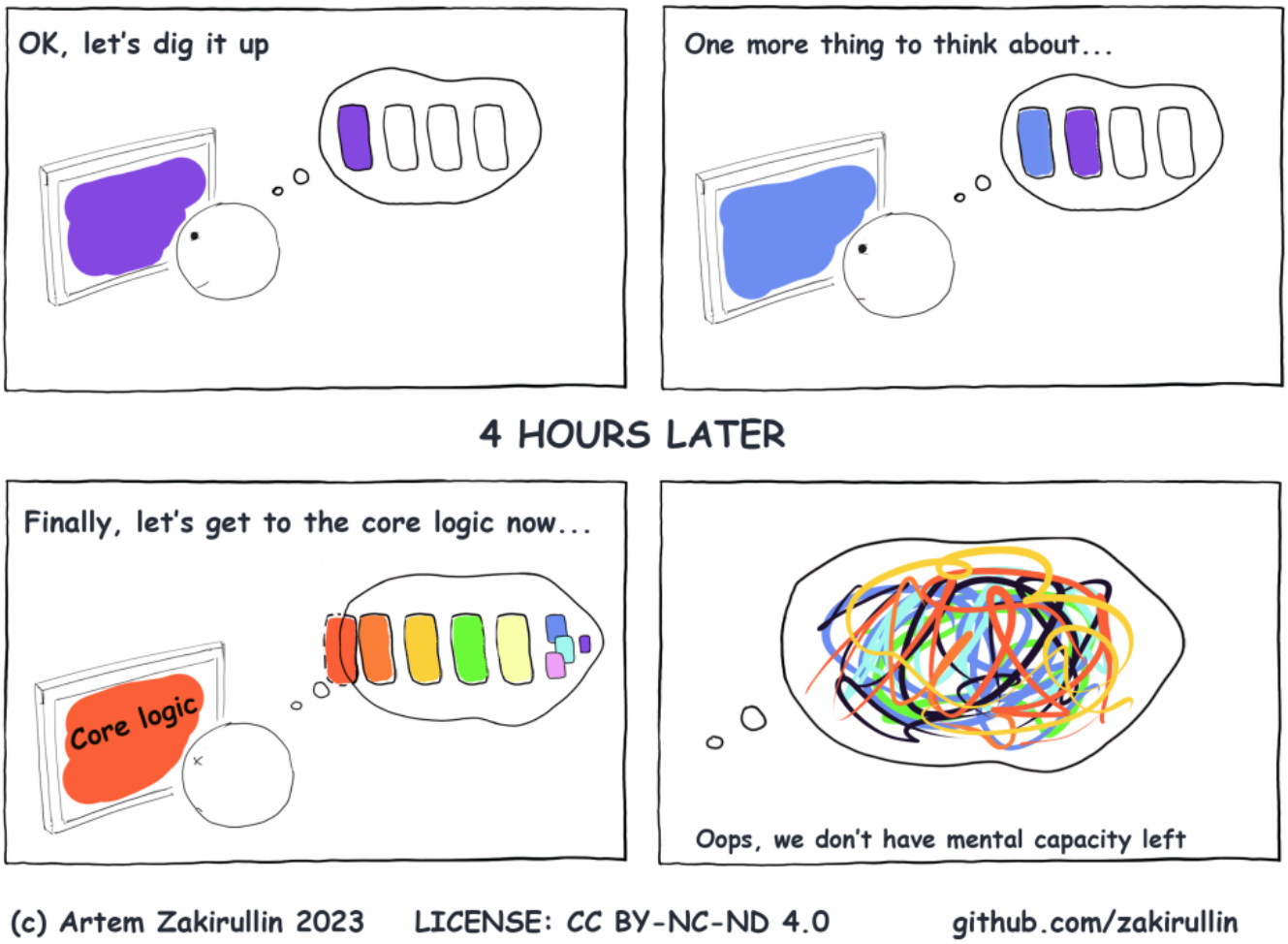
Tải nhận thức

Tải nhận thức là những gì mà lập trình viên cần nghĩ để hoàn thành một task.

Chúng ta cần cải thiện tải nhận thức trong dự án nhiều nhất có thể.

Khi đọc code, bạn đặt những thứ như giá trị của biến, luồng điều khiển logic và gọi hàm vào trong đầu. Một người bình thường có thể nhớ được 4 thứ như thế trong bộ nhớ làm việc. Một khi *tải nhận thức* đạt tới giới hạn, sẽ cần tới một sợ nỗ lực đáng kể để hiểu chúng.

Giả sử chúng ta được yêu cầu sửa một dự án vô cùng xa lạ đã được hoàn thành. Chúng ta được biết rằng một lập trình viên thực sự thông minh đã phát triển nó. Rất nhiều thiết kế ngầu, thư viện đặc biệt và công nghệ thời thượng được sử dụng. Nó đồng nghĩa với việc lập trình viên trước đã taọ ra cho chúng ta một mức tải nhận thức cao.



Phần khó khăn nhất là lập trình viên trước có thể đã không phải chịu mức độ tải nhận thức cao như thế vì họ đã quen với project.

Quen thu?c và ??n gi?n

Vấn đề là sự quen thuộc không đồng nghĩa với sự đơn giản. Họ cảm thấy giống nhau – cùng một sự dễ dàng khi chuyển qua một không gian mà không tốn quá nhiều nỗ lực tư duy nhưng đó là vì những lý do rất khác nhau. Mọi "thông minh" và thủ thuật phi ngôn ngữ bạn sử dụng đều gây ra khó khăn cho những người khác. Khi học xong điều đó, họ sẽ thấy làm việc với code bớt khó khăn hơn. Vì vậy, sẽ rất khó để làm đơn giản hoá code khi mà bạn đã quá quen với code đó. Đó là lý do tại sao tôi cố để đơn giản hoá code trước khi nó trở nên quá khó để khắc chế :)))

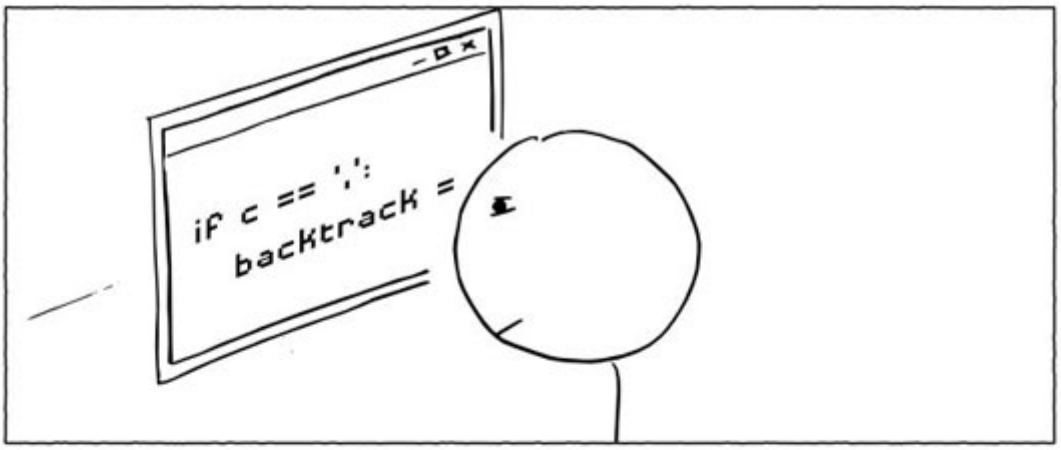
Có khả năng là những tác giả trước đõ đã taọ ra mớ hỗn độn này từng chút một chứ không phải là cùng một lúc. Nên bạn là người đầy tiên phải cố hiểu tất cả cùng một lúc.

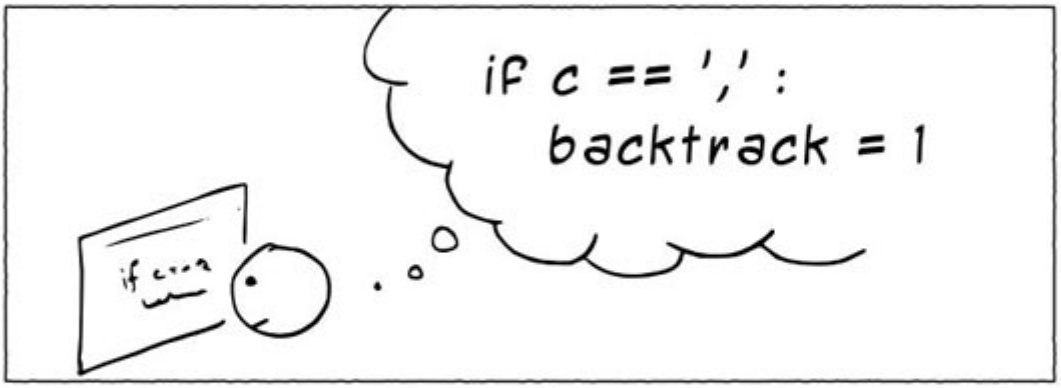
Trong lớp của tôi, tôi mô tả một thủ tục lưu trữ SQL phức tạp mà chúng ta đã làm với hàng trăm dòng điều kiện trong một mệnh đề WHERE khổng lồ. Một số hỏi tại sao người ta lại để cho mọi thứ tệ đến vậy. Tôi nói với họ:"Khi chúng chỉ là 2, 3 dòng điều kiện, việc thêm vào một dòng không làm cho nó trở nên khó hơn. Theo thời gian chúng là 20 hoặc 30 điều kiện, thêm vào một dòng vẫn không làm n trở nên phức tạp hơn!"

Không có “lực đơn giản hoá” nào tác động lên code base của bạn ngoài lựa chọn của chính bạn. Việc đơn giản hoá cần rất nhiều nỗ lực, mọi người thường quá vội vàng.

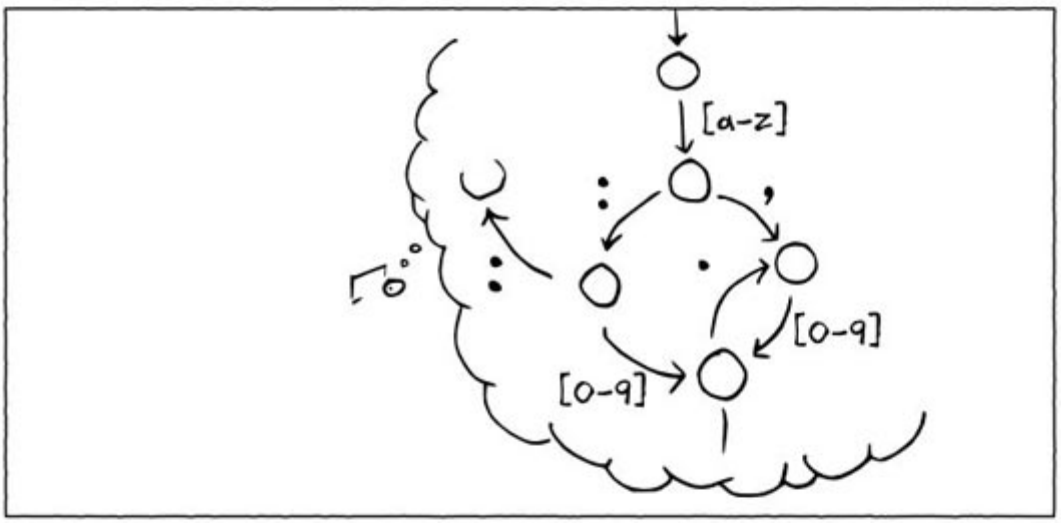
“*Khi có người mới tham gia vào dự án hay thử mức độ bối rối của họ. Nếu họ mất hơn 40ph cho một dòng thì đó là chỗ bạn cần cải thiện.*” by Dan North

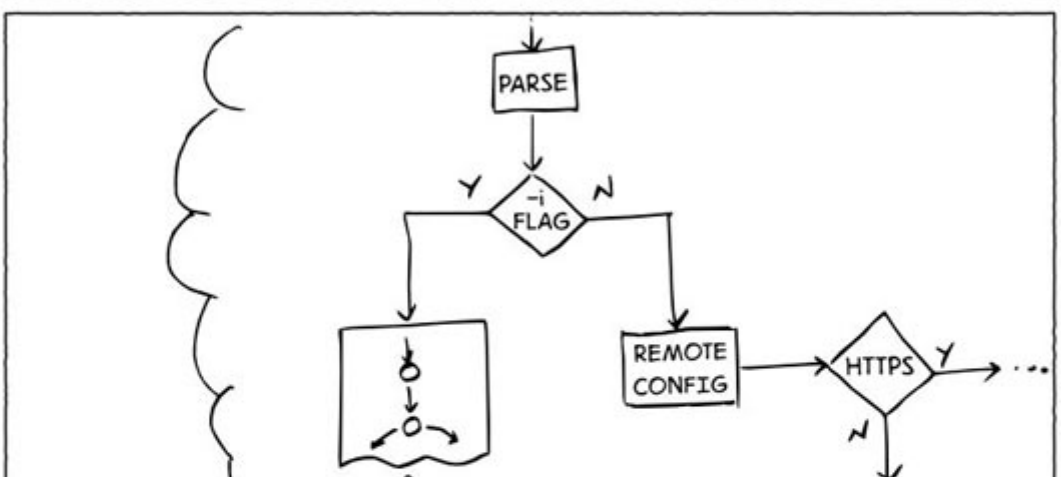
Tải nhận thức và chỗ đứt đoạn


if c == ',':
backtrack =


if c == ',' :
backtrack = 1


...so if the current character is a comma, we set the back-tracking flag...


[a-z]
:
,
:
[0-9]
[0-9]


PARSE
Y
-i
FLAG
N
REMOTE
CONFIG
HTTPS
Y
N

Các loại tải nhận thức

Bản chất – gây ra bởi độ khó của task. Nó không thể được cải thiện. Nó là trọng tâm của phát triển phần mềm.

Ngoại cảnh – tạo ra bởi cách trình bày thông tin. Gây ra bởi những thứ không liên quan trực tiếp đến task như thói quen của dev, Có thể cải thiện. Chúng ta sẽ tập trung vào loại tải nhận thức này.

image-1-1024x374.png

Chúng ta sẽ đề cập đến các mức độ tải nhận thức sau:

- Lv1: bộ nhớ làm việc mới, tải nhận thức bằng 0.
- Lv2: Có sự kiện trong bộ nhớ, tải nhận thức tăng lên.
- Lv3: Bộ nhớ quá tải, nhiều hơn 4 sự kiện.

Điều kiện phức tạp

```
if val > someConstant // Lv1
&& (condition2 || condition3) // Lv2, prev cond should be true, one of c2 or c3 has be true
&& (condition4 && !condition5) { // Lv3, we are messed up here
…
}
```

Sử dụng các biến trung gian với tên có ý nghĩa:

```
isValid = var > someConstant
isAllowed = condition2 || condition3
isSecure = condition4 && !condition5
// Lv1, we don't need to remember the conditions, there are descriptive variables
if isValid && isAllowed && isSecure {
…
}
```

If lồng nhau

```
if isValid { // Lv1, okay nested code applies to valid input only
if isSecure { // Lv2, we do stuff for valid and secure input only
stuff // Lv3
}
}
```

So sánh với early returns:

```
if !isValid
return

if !isSecure
return

// Lv1, we don't really care about earlier returns, if we are here then all good

stuff // Lv2
```

Chúng ta chỉ cần tập trung vào “con đường hanh phúc” từ đó giải phóng bộ nhớ bởi đủ thứ điều kiện tiên quyết.

Ác mộng kế thừa

Chúng ta được yêu cầu để thay đổi một số thứ cho admin: Lv1

```
AdminController extends UserController extends GuestController extends BaseController
```

Oh, chúng ta sẽ cần xem xét phần lớn chức năng đang nằm trong `BaseController`: Lv1+
Cơ chế vai trò nằm trong `GuestController`: Lv2
Một vài thứ bị thay đổi trong `UserController`: Lv2+
Cuối cùng chúng ta ở đây `AdminController` Lv2++

Oh, từ đã, đây là `SuperuserController` kế thừa `AdminController`. Bằng cách sửa đổi `AdminController` chúng ta có thể phá vỡ mọi thứ trong lớp kế thừa vậy nên hãy tìm hiểu `SuperuserController` trước: Lv3

Ưu tiên sử dụng kết hợp hơn là kế thừa. (tham khảo)

Quá nhiều phương thức, lớp hoặc module nhỏ

Phương thức, lớp và module có thể thay đổi cho nhau trong nội dung này.

Thần chú kiểu như “phương thức nne ngắn hơn 10 dòng code” hoặc “class nên nhỏ” hoá ra đều có sai lầm.

Module sâu – giao diện đơn giản, chức năng phức tạp.
Module nông – giao diện tương đối phức tạp so với chức năng nó cung cấp.

image-2-1024x517.png

Có quá nhiều module nông có thể khiến dự án trở nên khó hiểu. Chúng ta không chỉ phải nhớ chức năng của từng module mà còn phải nhớ tất cả các tương các giữa chúng. Để hiểu được module

nông chúng ta cần hiểu được tất cả các tương tác giữa chúng.

Việc ẩn giấu thông tin là vô cũng quan trọng và chúng ta không giấu nhiều thông tin vào trong module nông.

Có 2 dự án. Cả 2 đều bao gồm khoảng 5000 dòng code. Dự án 1 có khoảng 20 class nông, trong khi lớp thứ 2 chỉ có khoảng 7 class sâu. Không dự án nào được phát triển trong vòng 1,5 năm trở lại.

Khi làm việc lại, tôi nhận thấy việc hiểu được hơn 80 class nông là vô cùng khó khăn. Tôi sẽ phải xây dựng lại gần như toàn bộ tải nhận thức trước khi có thể tiếp tục làm việc. Mặt khác tôi có thể nắm bắt nhanh hơn dự án thứ 2 với vài class sâu có giao diện đơn giản.

"*Các thành phần tốt nhất cung cấp những chức năng mạnh mẽ cùng với giao diện đơn giản.*" by **John K. Ousterhout**

```
open(path, flags, permissions)
read(fd, buffer, count)
write(fd, buffer, count)
lseek(fd, offset, referencePosition)
close(fd)
```

Một triển khai hiện đại của giao diện này bao gồm hàng trăm hoặc hàng ngàn dòng mã. Rất nhiều sự phức tạp bị ấn giấu. Tuy nhiên rất dễ để sử dụng vì nó có giao diện đơn giản.

> *Đây là ví dụ về module sâu lấy từ cuốn A Philosophy of Software Design của John K. Ousterhout. Cuốn sách này không chỉ đề cập đến bản chất phức tạp của phát triển phầm mềm mà còn có diễn giải sâu sắc về bài báo có ảnh hưởng của Parnas On the Criteria To Be Used in Decomposing Systems into Modules. Cả 2 đều là những bài nên đọc. Các bài liên quan khác: It's probably time to stop recommending Clean Code, Small Functions considered Harmful, Linear code is more readable.*

Nếu bạn nghĩ chúng ta đang ủng hộ những đối tượng cồng kềnh của chúa với quá nhiều trách nhiệm thì có thể bạn đã nhầm.

Quá nhiều mircoservice nông

Chúng ta có thể ứng dụng nguyên tắc về các class, module, method vừa nói ở trên vào thiết kế mircroservice. Quá nhiều mircroservice nông sẽ không mang lại hiệu quả gì. Dich vụ chúng ta hướng đến không nông đến thế. Một trong những điều tồi tệ và khó khăn nhất đó là nguyên khối phân tán (distributed monolith). Thường là kết quả của việc tách ra thành quá nhiều microservice nông.

Tôi từng tư vấn cho một công ty khởi nghiệp, nơi mà 3 dev giới thiệu với tôi những 17 mircroservice. Họ đã chậm tiến độ đến 10 tháng và biến mất cho đến tận sát ngày phát hành. Với mỗi yêu cầu mới họ sẽ phải chỉnh sửa hơn 4 microservice. Việc chuẩn đoán trở nên khó khăn do tích hợp của những microservice này. Cả gánh nặng về thời gian và tải nhận thức đều không thể chấp nhận được.

Đây có phải là cách đúng khi tiếp cận sự không chắc chắn của một hệ thống mới? Rất khó để chỉ ra logic đúng ngay từ đầu và với việc đưa ra quá nhiều microservice chúng ta làm chúng càng tồi tệ hơn. Team chỉ bào chữa kiểu:"Các công ty FAANG (Facebook, Amazon, Apple, Netflix, Google) đã ứng dụng microservice rất thành công"

Một khối được chế tạo tốt với những module thực sự cô lập thường tiện lợi và linh động hơn những mircroservice lộn xộn. Chỉ khi nhu cầu triển khai riêng biệt thực sự quan trọng (mở rông team dev) bạn mới cần xem xét thêm lớp mạng vào giữa các module.

Ngôn ngữ giàu tính năng

Chúng ta cảm thấy phấn khích mỗi khi ngôn ngữ yêu thích cho ra tính năng mới. Chúng ta dành thời gian để học và rồi build code trên đó.

Nếu có nhiều tính năng, chúng ta sẽ dành ra nửa giờ với một vài dòng code để sử dụng tính năng này hoặc tính năng khác. Và đó là sự lãng phí thời gian. **Nhưng còn tệ hơn nữa là khi bạn sử dụng lại nó, bạn vẫn sẽ phải suy nghĩ lại từ đầu.**

Bạn không chỉ phải hiểu chương trình phức tạp đó mà bạn còn phải hiểu tại sao lập trình viên lại tiếp cận vấn đề từ những tính năng đã có đó.

> *"Giảm tải nhận thức bằng cách giới hạn các lựa chọn" – by Rob Pike.*

Các tính năng của ngôn ngữ là rất OK nếu như chúng trực giao với nhau.

Suy nghĩ của một kỹ sư với hơn 20 năm kinh nghiệm C++

Hôm nọ, tôi đã xem RSS reader của mình và nhận ra tôi có khoảng 300 bài viết chưa đọc với thẻ C++. Tôi đã không đọc bài viết nào về ngôn ngữ này từ mùa hè năm ngoái và tôi cảm thấy thật tuyệt!

Tôi đã sử dụng C++ 20 năm, gần 2/3 cuộc đời. Hầu hết kinh nghiệm của tôi nằm ở việc giải quyết những góc tối của ngôn ngữ (chẳng hạn như những hành vi không xác định). Nó không phải là kinh nghiệm có thể tái sử dụng (ở những ngôn ngữ khác) và thật đáng sợ khi vứt bỏ chúng ngay bây giờ.

Bạn có thể tưởng tượng, kí tự `||` có nghĩa khác nhau trong`((!P<T> || !Q<T>))`và trong `(!(P<T> || Q<T>))` trong vế đầu tiên nó là sự phân tách ràng buộc trong vế thứ 2 nó là toán tử OR bình thường.

Trước C++20, nếu bạn muốn cấp phát bộ nhớ cho một kiểu dữ liệu đơn giản và sao chép dữ liệu vào đó bằng `memcpy`, thì điều này không khởi tạo vòng đời của một đối tượng. Về cơ bản, đối tượng đó sẽ không hoạt động chính xác. May mắn là trong C++20, vấn đề này đã được khắc phục. Tuy nhiên, việc bổ sung các tính năng mới cũng dẫn đến việc ngôn ngữ trở nên phức tạp hơn, đòi hỏi người lập trình phải học hỏi và ghi nhớ nhiều hơn, gây ra “gánh nặng nhận thức” lớn hơn.

Gánh nặng nhận thức không ngừng gia tăng, ngay cả khi vấn đề đã được khắc phục. Là một lập trình viên chuyên nghiệp, tôi cần phải biết những gì đã được sửa, khi nào được sửa và nó như thế nào trước đây. Chắc chắn rồi, C++ hỗ trợ tốt cho các phiên bản cũ [của code], nhưng điều đó cũng có nghĩa là bạn sẽ phải đối mặt với những thứ cũ kỹ đó. Ví dụ, tháng trước một đồng nghiệp đã hỏi tôi về một hành vi lạ trong C++03.

Trước đây có 20 cách khởi tạo [object]. Ký hiệu khởi tạo thống nhất đã được thêm vào. Bây giờ chúng ta có 21 cách khởi tạo. Nhân tiện, còn ai nhớ quy tắc chọn hàm dựng (constructor) từ danh sách khởi tạo không? Có gì đó liên quan đến chuyển đổi ngầm với ít mất mát thông tin nhất, nhưng nếu giá trị được biết tĩnh [ngay từ khi biên dịch], thì…

Sự tăng tải nhận thức này không phải do bản chất công việc hay bản chất của lập trình. Nó phức tạp dần theo thời gian, việc sửa lỗi, thêm tính năng làm cho ngôn ngữ trở nên phức tạp hơn mức cần thiết.

Tôi đặt ra một số quy tắc như nếu dòng mã đó không rõ ràng và tôi phải nhớ tài liệu chuẩn thì tôi sẽ không viết theo cách đó nữa. Nhân tiện thì tài liệu đó có 1500 trang.

Nhưng không có nghĩa tôi đang cố đổ lỗi cho C++. Tôi yêu ngôn ngữ này. Chỉ là tôi đang mệt mỏi thôi.

Logic doanh nghiệp và mã trạng thái HTTP

Ở backend chúng ta trả về:

`401 khi token h?t h?n`

`403 khi không ?? quy?n truy câp`
`418 khi user b? ban`

Frontend sử dụng backend API để triển khai tính năng đăng nhập. Họ sẽ phải tạm thời tải nhận thức vào trong não:

```
401 khi token h?t h?n
403 khi không ?? quy?n truy cập
418 khi user b? ban
```

Các frontend dev sẽ tạo ra các biến (hoặc hàm) kiểu như `isTokenExpired(status)` để các thế hệ dev sau không phải tạo lại

Rồi sau đó QA tới và:”Này tôi có trạng thái 403. Nó là token hết hạn hay người dùng không đủ quyền vậy?”. QA không thể nhảy thẳng vào test bởi vì họ phải tạo lại tải nhận thức mà backend đã tại ra.

Tại sao lại phải giữ những mã này trong bộ nhớ làm việc ngắn hạn? Tốt hơn hết nên loại bỏ hết các mã này trả về trực tiếp mô tả kiểu: `“code”: “jwt_has_expired”`

Luật tương tự cũng được áp dụng cho tất cả các số hiệu trạng thái (trong database hoặc bất kỳ đâu). Chúng ta đâu còn ở trong thời kỳ máy tính 640K để tối ưu bộ nhớ.

> *Mọi người dành thời gian tranh cãi giữa 401 và 403, đưa ra lựa chọn dựa trên mức độ hiểu biết của họ. Nhưng cuối cùng nó chẳng có ý nghĩa gì cả. Chúng ta có thể chia các lỗi này thành phía người dùng hoặc phía server thế nhưng ngoài việc đó thì mọi thứ khá mơ hồ. Với việc tuân theo “RESTful API” và sử dụng tất cả các loại động từ trạng thái HTTP thì đơn giản là không có tiêu chuẩn nào cả. Tài liệu hợp lệ duy nất về vấn đề này là một bài báo được Roy Fielding xuất bản vào năm 2000 và nó không đề cập gì đến các động từ và trạng thái. Mọi người chỉ sử dụng một vài trạng thái HTTP cơ bản và chỉ POST mà vẫn có thể làm tốt.*

Lạm dụng nguyên tắc DRY

Đừng lặp lại, đó là 1 trong những nguyên tắc đầu tiên của kỹ sư phần mềm. Nó ăn sâu vào chúng ta đến mức khiến chúng ta không thể ngồi yên khi có thêm vài dòng code. Mặc dù đây là một nguyên tắc tốt và cơ bản nhất thế nhưng khi lạm dụng n vẫn gây ra vấn đề với tải nhận thức khiến chúng ta không thể kiểm soát.

Ngày nay, mọi người xây dựng phần mềm dựa trên những thành phần được phân tách hợp lý. Thông thường chúng được phân phối giữa nhiều codebase đại diện cho những dịch vụ riêng biệt. Khi cố gắng loại bỏ sự lặp lại nào bạn có thể sẽ tạo ra sự ràng buộc giữa các thành phần. Kết quả có thể tạo ra sự thay đổi ở nhiều phần khác mà chúng ta không thể lường trước được. Nó có thể cản trở việc thay thế, thay đổi các thành phần một cách độc lập mà không ảnh hưởng đến hệ thống.

Sự thật thì vấn đề cũng xảy ra cả trong 1 module duy nhất. Có thể bạn đã tạo ra tạo ra những function chung quá sớm dựa vào những điểm chung mà có thể sẽ không tồn tại lâu. Điều này có thể dẫn đến sự trừu tượng không cần thiết dẫn đến khó bảo trì và mở rộng

> *“Một chút sao chép sẽ tốt hơn một chút rằng buộc”* – Rob Pike

Vì chúng tôi không muốn phát minh lại bánh xe nên chúng tôi sử dụng những thư viện lớn và nặng cho những chức năng nhỏ mà chúng tôi có thể dễ dàng tự viết? Nó tại nên những phụ thuộc không cần thiết và làm code trở nên cồng kềnh. Cần đưa ra quyết định sáng suốt khi nào nên thêm những thư viện cồng kềnh khi nào nên tự viết những đoạn mã nhỏ gọn cho những công việc đơn giản.

Lạm dụng nguyên tắc này có thể dẫn đến sự phụ thuộc giản tiếp (hoặc sự phụ thuộc không cần thiết), trừu tượng hoá vội vàng và các giải pháp lớn, chung chung phức tạp về bảo trì, gây ra gánh nặng tải nhận thức.

Rằng buộc chặt chẽ với một framework

Framework phát triển với tốc độ của riêng chúng, đa phần thì không phù hợp với vòng đời dự án của chúng ta.

Quá phụ thuộc vào một framework khiến chúng ta buộc phải học framework đó trước (hoặc một phiên bản cụ thể của framework). Mặc dù framework cho phép chúng ta khởi chạy MVP chỉ trong vài ngày nhưng về lâu dài chúng làm tăng độ phức tạp và tải nhận thức không cần thiết.

Tệ hơn, ở một số điểm framework có thể trở thành cản trở khi phải đối mặt với những yêu cầu mới không phù hợp với kiến trúc. Từ đó mọi người từ bỏ việc sử dụng framework và làm việc trên một phiên bản tuỳ chỉnh. Hãy tưởng tượng người mới đến sẽ phải chịu bao nhiêu tải nhận thức (để học về framework tuỳ chỉnh này) trước khi có thể làm việc.

Điều này không có nghĩa là chúng tôi ủng hộ việc phát triển lại từ đầu!

Chúng ta có thể viết code theo hướng phi phụ thuộc framework. Logic nghiệp vụ không nên nằm trong một framework nào, đúng hơn nó nên sử dụng các thành phần của framework. Đưa framework ra bên ngoài core logic của bạn. Sử dụng framework như một thư viện. Điều này cho phép người mới đến tiếp cận dự án ngay từ những ngày đầu tiên mà không cần tìm hiểu lại toàn bộ những phần phức tạp của framework.

Kiến trúc lục giác/củ hành

Có một sự hứng thú kỹ thuật nhất định về tất cả những thứ này.

Bản thân tôi là người ủng hộ kiến trúc củ hành (Onion Architecture) trong nhiều năm qua. Tôi sử dụng nó và khuyến khích các đội nhóm sử dụng nó. Sự phức tạp của các dự án tăng lên, chỉ riêng số lượng file cũng đã tăng gấp đôi. Cảm giác như chúng tôi đang viết rất nhiều glue code để ghép nối các lớp với nhau. Với các yêu cầu thay đổi liên tục, chúng tôi phải thực hiện thay đổi trên nhiều

lớp trừu tượng khác nhau, mọi thứ trở nên trẻ nhạt và mất thời gian.

Trong quá trình fix bug chúng tôi cần lần theo từng hàm để tìm ra vấn đề. Việc tách rời các lớp của kiến trúc này đòi hỏi phải theo dõi một lượng khổng lồ các dấu vết để xác định nơi xảy ra lỗi.

Kiến trúc này có vẻ trực quan nhưng mỗi khi chúng tôi thử áp dụng vào dự án thì nó gây hại nhiều hơn là có lợi. Cuối cùng, chúng tôi từ bỏ tất cả để ủng hộ kiến trúc đảo ngược phụ thuộc (dependency inversion) cũ kỹ. Không cần tìm hiểu về thuật ngữ port/adapter, không có các lớp trừu tượng không cần thiết, và không quá tải nhận thức.

Đừng thêm các lớp trừu tượng vì lợi ích của kiến trúc. Thêm chúng khi nào bạn cần một điểm mở rộng hợp lý cho lý do thực tế. Các lớp trừu tượng không miễn phí, chúng cần phải lưu trong bộ nhớ làm việc của chúng ta.

> “Uầy đm say cà phê vl. Buổi trưa ngồi đọc buồn ngủ quá. Biết là uống cà phê sẽ say vl mà vẫn cố chấp để giờ say vl.” – độc thoại nội tâm của ThanhDV

Mặc dù kiến trúc theo lớp (layered architectures) đã thúc đẩy một sự chuyển đổi quan trọng từ các ứng dụng tập trung vào cơ sở dữ liệu truyền thống sang một cách tiếp cận độc lập với cơ sở hạ tầng, trong đó logic kinh doanh cốt lõi không phụ thuộc vào bất kỳ yếu tố bên ngoài nào, nhưng ý tưởng này không hề mới mẻ.

Những kiến trúc này không phải là cơ bản, chúng là hậu quả do chủ quan và thiên vị của những nguyên tắc cơ bản. Tại sao lại dựa vào những cách giải thích chủ quan đó??? Thay vào đó hãy tuân theo những nguyên tắc cơ bản: nguyên tắc đảo ngược phụ thuộc (dependency inversion principle), cách ly (isolation), nguồn duy nhất của sự thật (single source of truth), thực sự bất biến (true invariant), độ phức tạp (complexity),tải nhận thức (cognitive load) và ẩn giấu thông tin (information hiding).

- *Subjective, biased consequences (hệ quả mang tính chủ quan và thiên kiến): Cách thức tổ chức các lớp trong kiến trúc theo lớp có thể khác nhau tùy thuộc vào người thiết kế và bối cảnh cụ thể.*
- *Dependency inversion principle (nguyên tắc đảo phụ thuộc): Nguyên tắc này nói rằng các lớp cấp cao không nên phụ thuộc vào các lớp cấp thấp, mà cả hai đều nên phụ thuộc vào các abstraction (sự trừu tượng).*
- *Isolation (cô lập): Các thành phần của phần mềm nên được thiết kế sao cho chúng ít phụ thuộc lẫn nhau nhất có thể để dễ dàng bảo trì và thay đổi.*
- *Single source of truth (nguồn duy nhất của sự thật): Dữ liệu chỉ nên được lưu trữ và quản lý ở một vị trí duy nhất trong hệ thống để đảm bảo tính nhất quán.*
- *True invariant (bất biến thực sự): Một đặc tính của hệ thống luôn luôn đúng bất kể trạng thái của hệ thống.*
- *Complexity (độ phức tạp): Giữ cho code đơn giản và dễ hiểu để dễ dàng bảo trì và tránh lỗi.*
- *Cognitive load (tải nhận thức): Giảm thiểu sự phức tạp của code để dễ dàng cho lập trình viên tương lai hiểu và làm việc với nó.*

- *Information hiding (ẩn giấu thông tin): Chỉ nên cho phép các thành phần của phần mềm truy cập vào thông tin cần thiết cho chúng hoạt động, giúp bảo mật và giảm thiểu sự phụ thuộc.*

DDD

Thiết kế DDD (Domain-Driven Design – Thiết kế hướng đối tượng theo vấn đề) có một số điểm mạnh mặc dù nó thường bị hiểu sai. Mọi người nói rằng họ viết code theo DDD điều này hơi lạ vì **DDD tập trung vào vấn đề không phải giải pháp**.

Ngôn ngữ phổ biến, miền (domain), ngữ cảnh giới hạn (bounded context), tổng hợp (aggregate), bão sự kiện (event storming) đều liên quan đến vấn đề. Chúng được thiết kế để giúp chúng ta học hỏi những hiểu biết sâu sắc về miền và xác định ranh giới. DDD cho phép dev, chuyên gia và doanh nghiệp giao tiếp hiệu quả bằng ngôn ngữ thống nhất. Thay vì tập trung vào các khía cạnh về vấn đề của DDD chúng ta thường nhấn mạnh đến cấu trúc thư mục cụ thể, dich vụ, kho lưu trữ, các kỹ thuật khác liên quan đến giải pháp.

- *Miền (domain): Là lĩnh vực kinh doanh cụ thể mà phần mềm sẽ giải quyết vấn đề.*

Rất có khả năng chúng ta hiểu DDD là độc nhất và mang tính của quan. Nếu chúng ta xây dựng code dựa trên hiểu biết này nghĩa là nếu chúng ta tạo ra quá nhiều gánh nặng nhận thức không cần thiết thì các nhà phát triển tương lai đều sẽ gặp rắc rối lớn.

Học từ những người khổng lồ

Hãy nhìn vào thiết kế tổng thể của một trong những công ty công nghệ lớn nhất:

- **Rõ ràng (Clarity):** Mục đích và lý do của code phải rõ ràng với người đọc.
- **Đơn giản (Simplicity):** Code đạt được mục tiêu của nó theo cách đơn giản nhất có thể.
- **Súc tích (Concision):** Code dễ dàng phân biệt các chi tiết quan trọng, và cách đặt tên và cấu trúc hướng dẫn người đọc hiểu các chi tiết này.
- **Bảo trì (Maintainability):** Code dễ dàng cho lập trình viên tương lai sửa đổi một cách chính xác.
- **Kiên định (Consistency):** Code tuân theo phong cách chung của toàn bộ codebase.

Từ ngữ diêm dúa (fancy buzzword) không tuân theo các nguyên tắc này và chỉ tạo ra thêm những gánh nặng nhận thức không cần thiết.

image-3.png

Code Complexity vs. Experience from @flaviocopes

> *“Việc debug khó gấp đôi so với việc viết code ngay từ đầu. Do đó nếu bạn viết code càng “thông minh” thì bạn lại càng không đủ “thông minh” để debug chúng code đó.” – Brian Kernighan*

Tổng kết

Bản chất phức tạp và nhiều mặt của tải nhận thức trong việc hiểu và giải quyết vấn đề đòi hỏi một cách tiếp cận chăm chỉ và có chiến lược để vượt qua những khó khăn và tối ưu hoá khả năng suy nghĩ.

Bạn cảm thấy sao? Chúng tôi vừa tạo ra một tải nhận thức không cần thiết cho bạn đấy. Đừng làm thế với đồng nghiệp của bạn.

image-4-1024x374.png

Chúng ta nên giảm thiểu bất kỳ gánh nặng nhận thức nào vượt quá mức cần thiết cho công việc chúng ta đang làm.

Code đơn giản, dễ hiểu là một hướng đi tốt.