# Estimation

**Đừng bao giờ đánh giá thấp việc estimation hay thực hiện nó một cách qua loa!**

Estimation dài hơn thực tế thì khỏe, nhưng estimation mà ngắn hơn thực tế thì ăn cứt. "Ủa, nếu vậy thì mình cứ estimation dài dài ra?" Vấn đề ở đây là khách hàng/cấp trên có **deadline** cho mình, và trong rất nhiều trường hợp, deadline của họ ngắn hơn estimation của mình. Vậy phải làm sao? "Tính toán làm sao để estimation vừa sát ngày deadline (vd, deadline ngày 10 thì estimate sao cho hoàn thành task trong ngày 9)? Chấp nhận nó và cố gắng để hoàn thành kịp deadline?" Hay là lại ăn cứt?

NO! Trong tình huống như thế này phải tìm cách để **kéo dài deadline**. Đưa ra những lý do hợp lý; như thế nào thì tùy project (vd, hệ thống lớn nhiều thứ mơ hồ nên cần nhiều thời gian investigate hơn; cái này phải stub để test, cái kia phải test trên máy server khiến thời gian testing sẽ rất lâu, v.v.. muôn vàn lý do).

Dù sớm hay trễ thì **quan trọng nhất vẫn là làm sao để estimation cho đúng**. Dùng từ “đúng” cho nó ngầu vậy thôi chứ làm éo gì mà đúng nổi. Chắc chắn sẽ có sai số. Bản thân "estimate" đã có nghĩa là "ước lượng".

Dưới đây là **kinh nghiệm xương máu khi estimation**:

* **PHẢI chia nhỏ task** – càng nhỏ càng tốt (lý tưởng là tính bằng đơn vị 0.25h). Đừng chơi ngu mà estimate theo cảm tính, kiểu "à cái này hồi trước có làm một cái tương tự rồi nè, nhắm tầm 3 tiếng là xong thôi", hoặc là "à, cái này thư viện có sẵn rồi nè, chỉ cần gọi API là chạy được thôi". Chưa chắn đâu! Chưa phân tích nó là gì, solution là gì, thư viện chạy được không, hướng code kiểu gì, test trông ra sao … thì không nói lên được điều gì cả.
* **PHẢI** **cộng thêm buffer** thời gian cho mỗi task. Cái này là phải có nha, đừng bao giờ bỏ qua nó vì sẽ có những (dù ít hay nhiều) vấn đề phát sinh trong quá trình làm task gây gián đoạn công việc. Điển hình nhất là một vấn đề tưởng như đơn giản, nhưng nhiều giờ làm không xong.

Ngoài ra, sự mất tập trung cũng làm hao hụt khoảng thời gian có ích. Ví dụ, với một task nhỏ nhỏ tầm 2-3 tiếng thì thời gian thật sự mà đa số mọi người tập trung để thực hiện nó là khoảng 1.5-2.5 tiếng thôi.

Vậy thời gian còn lại đi đâu? Nếu dành cho việc có ích thì đó là: check và trả lời email, chat trao đổi công việc với đồng nghiệp, support/train new members, peer review, meeting (lâu lâu sẽ gặp), đi vệ sinh, và đặc biệt là các vấn đề liên quan đến process sẽ chiếm một khoảng thời gian tương đối lớn nữa. Nếu dành cho việc không có ích thì vô số: 15p ngồi nghe nhạc, 10p đọc tin tức, 5p tán dốc với đồng nghiệp, chưa kể đi cà phê … có quá nhiều thứ khiến ta mất tập trung.

# OT

**OT có tốt không**? Tùy hoàn cảnh:

* Nếu thỉnh thoảng (1 lần/tuần, mỗi lần 4 tiếng) và 200% lương thì OT sẽ tốt. Hihi!
* Nhưng nếu OT thường xuyên (3 lần/tuần, mỗi lần 3 tiếng) thì chắc chắn là không tốt. Nó lại càng không tốt khi OT kiểu thường xuyên như vậy sẽ bị đánh giá là low performance thay vì nhận được thêm tiền OT.

Ngoài chuyện mất thêm thời gian cho bản thân và gia đình hay chuyện tiền nong thì OT nhiều sẽ gây mệt mỏi, căng thẳng hệ thần kinh. Kết quả sau nhiều ngày OT liên tục sẽ là một bộ não thiếu minh mẫn hay mắc những sai lầm ngớ ngẩn và một tinh thần đầy tiêu cực. Đây là bài học xương máu mà tôi đã thực sự trải qua và nó "mệt vl".

**Nguyên nhân OT**

* Estimation bị sai: Dù có tính toán chi tiết đến đâu thì khả năng estimation bị sai là hoàn toàn có thể.
* Làm việc quá thiếu tập trung: Dù nguyên nhân khách quan (vị trí làm việc ồn ào, nhiều vấn đề không liên quan đến task bị phát sinh ...) hay chủ quan (không chú tâm vào công việc - nghe nhạc, đọc tin tức, lướt Facebook quá nhiều) thì cũng **phải HẠN CHẾ tình trạng thiếu tập trung** hết mức có thể.

Rất khó để KHÔNG HỀ bị mất tập trung, nhưng hạn chế nó thì hoàn toàn nằm trong khả năng. Chuyên nghiệp trong công việc và kỷ luật với bản thân giống như câu hói "**làm hết sức, chơi hết mình**" - hãy luôn ghi nhớ phương châm này. Đã làm thì phải làm cho tập trung, cho tới nơi tới chốn, cho nhanh gọn lẹ để về nhà làm những việc thú vị hơn. Ngồi nhây nhây, làm mấy việc tào lao để khi sát giờ G mới chạy cong đít, bug từa lưa, OT kéo dài thì chỉ có "tèo" sớm thôi. Lại một bài học xương máu của bản thân tôi!

**Kim chỉ nang**

Cái nghề lập trình dùng nhiều chất xám này quan trọng là ở chỗ trong cùng một khoảng thời gian, mình làm nó có hiệu quả hay không. Chứ KHÔNG phải là cứ làm ngày làm đêm là sẽ xong việc nhanh hơn. Mệt thì nghỉ ngơi; gặp vấn đề bế tắc không có hướng giải quyết thì dừng lại, đi tới đi lui hay chơi vài set ping pong để relax nhẹ rồi hãy quay lại suy nghĩ vấn đề theo một hướng đi khác. Như vậy sẽ hiệu quả hơn là cứ đâm đầu vào làm giờ này qua giờ khác trong tâm thế nóng vội, lo lắng. Càng làm như vậy lại càng OT, và như đã nói, OT càng nhiều đầu óc càng thiếu minh mẫn, và kết quả cuối cùng là dù có OT vẫn không hoàn thành công việc.

À, một thứ rất quan trọng nữa. Để relax những khi bế tắc có nhiều cách, vận động nhẹ cơ thể là tốt nhất. Nhưng cắm mắt vào màn hình máy tính hay điện thoại (để xem YouTube hay để đọc tin tức) thì KHÔNG NÊN. Thật ra, cái này tùy người, nhưng với bản thân tôi, cắm mắt liên tục vào màn hình không giúp tôi relax một tí nào; có thể nguyên nhân là do việc này làm mắt tôi mỏi hơn nên gây áp lực lên thần kinh. Nếu tôi muốn nghe nhạc, tôi sẽ nhắm mắt lại và tận hưởng nó!

# Problem Solving

Là một programmer, problem solving là một skill bắt buộc phải trao dồi qua nhiều năm tháng bên cạnh những khía cạnh về ngôn ngữ, framework hay technique. Mọi thứ đều phải có phương pháp và problem solving không phải ngoại lệ. Nó không thuộc về yếu tố bẩm sinh, nó là sự luyện tập!

**Vậy làm sao để luyện tập và master problem solving?**

**1. Keep calm**: Yes, problem solving là sản phẩm của sự **điềm tĩnh, cẩn thận, rõ ràng và thư giãn**. Nó không phải là sự ép buộc não bộ phải suy nghĩ hết mức; điều này chỉ dẫn đến căng thẳng. Không thể chập dật, vội vàng, lủng củng mà giải quyết một vấn đề logic tốt được.

**2. Chia để trị**: Điều này đặc biệt đúng trong lập trình. Bạn không thể một phát mà nhìn ra được solution cho một vấn đề lớn được. Bạn bắt buộc phải chia nhỏ nó ra - càng nhỏ càng tốt. Bằng cách giải quyết từng vấn đề nhỏ lẻ, bạn sẽ thấy cái vấn đề to tát gì đó thật ra rất đơn giản. Nhưng làm sao để "chia để trị" cho hiệu quả? Đọc tiếp ba điều dưới đây!

**3. Từ output đến input**: Khi nhận được một task, đầu tiên và mấu chốt là phải nắm được mục đích cuối cùng của cái task này là gì; hay nói cách khác, **output của task là gì**. Khi biết được bước đi cuối cùng, ta mới biết mình cần thực hiện những bước gì khác để đến được bước này; bước nào quan trọng cần hoàn thành trước, bước nào có thể làm sau.

Ví dụ, ta cần implement một method với chức năng ghi log. Đầu tiên, ta phải nắm được cấu trúc nội dung của log trông như thế nào, nó được ghi ở thời điểm nào của chương trình và trong điều kiện gì. Nói cụ thể hơn, method này return cái gì (WHAT), sau đó là nó được gọi ở đâu (WHERE) và khi nào (WHEN).

Một sai lầm mà chung ta hay mắc phải là vội vàng đi tìm câu trả lời cho câu hỏi HOW (làm như thế nào để ghi log) mà chưa có câu trả lời thỏa đáng cho WHAT, WHERE and WHEN. Kết quả là output không tốt (nhẹ thì không tối ưu hay thừa đầu thiếu đuôi; nặng thì không đúng với requirement).

Một video rất hay về vấn đề này: <https://www.youtube.com/watch?v=i1T84mh3iqI>

**4. Từ tổng quan đến chi tiết**: Luôn nhìn vấn đề từ ngoài vào trong. Lấy lại ví dụ method ghi log ở mục 3, trong bước implement định nghĩa của method, thay vì bắt tay ngay vào coding để get info của log, ta nên **viết ra một skeleton những bước cần thực hiện trước**. Điều này giúp ta chắc chắn rằng các bước đi của mình là hợp lý và không có bước nào thừa thải. Ví dụ:

// Get info A

// Get info B

// Get info C

// Append info A, B and C to the log string

// Write the log string to file (viết trước tiên theo đúng phương pháp tư duy ngược)

Sau khi nắm được cái tổng quan, ta bắt đầu thực hiện các chi tiết trong từng bước để dần trả lời đầy đủ cho câu hỏi HOW.

**5. Từ idea đến code**: Một ví dụ điển hình cho tình huống này là các logic liên quan đến điều kiện hoặc vòng lặp. Dù logic đơn giản hay phức tạp, trước tiên hãy luôn trình bày nó bằng lời (thể hiện qua comment trong code), rồi hãy bắt đầu gõ code. Bằng cách này, idea được trình bày rõ ràng ngay từ ban đầu; nhiệm vụ chính của ta là thể hiện những idea này qua những dòng code. Tuyệt đối không vội vàng code ngay lập tức, nghĩ đến đâu gõ đến đó để rồi logic sai bét (nhẹ thì khó hiểu, dài dòng; nặng thì thiếu đầu thừa đuôi, bug từa lưa).

**6. Hiểu vấn đề, đừng cố ghi nhớ chúng**: Thay vì cố gắng ghi nhớ mọi thứ, chúng ta hãy thật sự HIỂU về chúng. Việc ghi nhớ một thuật toán hay design pattern không giúp ích nhiều; thay vào đó, cố gắng hiểu được chúng sẽ rất cần thiết, từ đó ta biết cách áp dụng chúng. Thực tế cho thấy, hiểu rõ vấn đề giúp ghi nhớ tốt hơn rất nhiều so với chỉ ghi nhớ mà không hiểu gì cả.

# Late Delivery

Being late is a part of our job, though we always try to avoid it at all cost. Sometimes we just blow our estimates and wind up late.

The trick to managing lateness is early detection and transparency. The worst case scenario occurs when you continue to tell everyone, up to the very end, that you will be on time—and then let them all down.

Don’t do this. Instead, regularly measure your progress against your goal, and come up with three fact-based end dates: best case, nominal case, and worst case. Be as honest as you can about all these dates.

**Do not incorporate hope into your estimates**

If the deadline is in ten days, and your nominal estimate is 12, you are not going to make it. Make sure that the team and the customers understand the situation, and don’t let up until there is a fall-back plan. Do not hope that you can get it all done in ten days! Hope is the project killer. Hope destroys schedules and ruins reputations. Hope will get you into deep trouble. So, don’t let anyone else have hope.

**Do not be rushing**

What if your manager sits you down and asks you to try to make the deadline? Hold to your estimates!

Your original estimates are more accurate than any changes you make while your boss is confronting you. Tell your boss that you’ve already considered the options (because you have) and that the only way to improve the schedule is to reduce scope. Do not be tempted to rush!

Woe to the poor developer who buckles under pressure and agrees to try to make the deadline. That developer will start taking shortcuts and working extra hours in the vain hope of working a miracle. This is a recipe for disaster because it gives you, your team and your customers false hope. It allows everyone to avoid facing the issue and delays the necessary tough decisions.

There is no way to rush. You can’t make yourself code faster. You can’t make yourself solve problems faster. If you try, you’ll just slow yourself down and make a mess that slows everyone else down, too. So you must answer your boss, your team, and your stakeholders by depriving them of hope.

# Practice

**Since your practice time is your own time, don’t have to use the same languages or platforms that you’re working in your company**. Pick any language you like and keep your polyglot skills sharp.

One way to stay ahead of the curve is to do what lawyers and doctors do: Take on some pro-bono work by contributing to an open-source project. There are lots of them out there, and there is probably no better way to increase your repertoire of skills than to actually work on something that someone else cares about.

If you work in a .NET shop, practice a little Java or Ruby at home. If you’re a Java programmer, contribute to a Rails project. If you write a lot of C++ for your employer, find a Python project and contribute to it.

# Meeting

## When declining?

**You don’t have to attend every meeting to which you are invited**. Indeed, it is unprofessional to go to too many meetings. You need to use your time wisely. So be very careful about which meetings you attend and which you politely refuse.

The person inviting you to a meeting is not responsible for managing your time. Only you can do that. So when you receive a meeting invitation, don’t accept unless it is a meeting for which your participation is immediately and significantly necessary to the job you are doing now.

## When leaving?

Meetings don’t always go as planned. Sometimes you find yourself sitting in a meeting that you would have declined had you known more. Sometimes new topics get added, or somebody’s pet peeve dominates the discussion. Over the years I’ve developed a simple rule: **When the meeting gets boring, leave**.

Again, you have an obligation to manage your time well. If you find yourself stuck in a meeting that is not a good use of your time, you need to find a way to politely exit that meeting.

## What stand-up meeting?

These meetings are part of the Agile cannon. Their name comes from the fact that the participants are expected to stand during the meeting. Each participant takes a turn to answer three questions:

1. What did I do yesterday?
2. What am I going to do today?
3. What’s in my way?

That’s all. Each question should require no more than twenty seconds, so each participant should require no more than one minute. Even in a group of ten people this meeting should be over well before ten minutes has elapsed.

# Focus-Manna

## What Is Focus-Manna?

Programming is an intellectual exercise that requires extended periods of concentration and focus. Focus is a scarce resource, rather like manna. **After you have expended your focus-manna, you have to recharge by doing unfocused activities for an hour or more**.

I don’t know what this focus-manna is, but I have a feeling that it is a physical substance that affects alterness and attention. Whatever it may be, you can feel when it’s there, and you can feel when it’s gone.

Professional developers learn to manage their time to take advantage of their focus-manna. **We write code when our focus-manna is high; and we do other, less productive things when it’s not**.

Focus-manna is also a decaying resource. If you don’t use it when it’s there, you are likely to lose it. That’s one of the reasons that meetings can be so devastating. If you spend all your focus-manna in a meeting, you won’t have any left for coding.

## Recharging

Focus-manna can be partially recharged by de-focussing. **A good long walk, a conversation with friends, a time of just looking out a window can all help to pump the focus-manna back up**.

Some people meditate. Other people grab a power nap. Others will listen to a podcast or thumb through a magazine.

I have found that once the manna is gone, you can’t force the focus. You can still write code, but you’ll almost certainly have to rewrite it the next day, or live with a rotting mass for weeks or months. So it’s better to take thirty, or even sixty minutes to de-focus.

## Muscle Focus

There is something peculiar about doing physical disciplines such as martial arts, tai-chi or yoga. Even though these activities require significant focus, it is a different kind of focus from coding. It’s not intellectual, it’s muscle. And somehow **muscle focus helps to recharge mental focus**. It’s more than a simple recharge though.

My chosen form of physical focus is bike riding. I’ll ride for an hour or two, sometimes covering twenty or thirty miles. I ride on a trail that parallels the Des Plaines river, so I don’t have to deal with cars. While I ride I listen to podcasts about astronomy or politics. Sometimes I just listen to my favorite music. And sometimes I just turn the headphones off and listen to nature.

Some people take the time to work with their hands. Perhaps they enjoy carpentry, or building models, or gardening. Whatever the activity, there is something about activities that focus on muscles that enhances the ability to work with your mind.

## Tomatoes Technique

One very effective way that I’ve used to manage my time and focus is to use the well-known Pomodoro Technique, otherwise knows as *Tomatoes Technique*. The basic idea is very simple. You set a kitchen timer (traditionally shaped like a tomato) for 25 minutes. While that timer is running, you let nothing interfere with what you are doing. If the phone rings you answer and politely ask if you can call back within 25 minutes. If someone stops in to ask you a question you politely ask if you can get back to them within 25 minutes. Regardless of the interruption, you simply defer it until the timer dings. After all, few interruptions are so horribly urgent that they can’t wait 25 minutes!

When the tomato timer dings, you stop what you are doing immediately. You deal with any interruptions that occurred during the tomato. Then you take a break of five minutes or so. Then you set the timer for another 25 minutes and start the next tomato.

Every fourth tomato you take a longer break of 30 minutes or so.

There is quite a bit written about this technique, and I urge you to read it. However, the description above should provide you with the gist of the technique.

Using this technique your time is divided into tomato and non-tomato time. Tomato time is productive. It is within tomatoes that you get real work done. Time outside of tomatoes is either distractions, meetings, breaks, or other time that is not spent working on your tasks.

How many tomatoes can you get done in a day? On a good day you might get 12 or even 14 tomatoes done. On a bad day, you might only get 2 or 3 done. If you count them, and chart them, you’ll get a pretty quick feel for how much of your day you spend productive and how much you spend dealing with “stuff.”

Some people get so comfortable with the technique that they estimate their tasks in tomatoes and then measure their weekly tomato velocity. But this is just icing on the cake. The real benefit of the Pomodoro Technique is that 25-minute window of productive time that you aggressively defend against all interruptions.

# Pressure

## Do Not Rush

The pressure sometimes comes despite all your best intentions and preventions. Sometimes the project just takes longer than anyone thought it would. Sometimes the initial design is wrong and must be reworked. Sometimes you lose a valued team member or customer. Sometimes you make a commitment that you just can’t keep. Then what?

**Sleepless nights won’t help you get done any faster**. Sitting and fretting won’t help either. And the worst thing you could do is to rush! Rushing will only drive you deeper into the hole. **Instead, slow down. Think the problem through. Plot a course to the best possible outcome, and then drive towards that outcome at a reasonable and steady pace**.

## Get Help

Pair! When the heat is on, find an associate who is willing to pair program with you. You will get done faster, with fewer defects. Your pair partner will help you hold on to your disciplines and keep you from panicking. Your partner will spot things that you miss, will have helpful ideas, and will pick up the slack when you lose focus.

By the same token, when you see someone else who’s under pressure, offer to pair with them. Help them out of the hole they are in.