ROBIN PALOTAI

PROGRAMMING WITHOUT ANXIETY

(SAMPLE, AS OF SEPTEMBER 8, 2019)

Contents

Preface 3

Disclaimer 3

Sources of Anxiety 4

Workplace Factors. 4

Rest of the Chapter 6

Bearing with Breakages 7

Not Breaking Things 7

Rest of the Chapter 8

Down by Debugging 10

A Wild Bug Appears!. 10

Search for a Precedent 12

Rest of the Chapter 13

Full Contents 14

Preface

Programming is a beautiful profession we love, but not one without times of stress and anxiety. Our code is mostly a dumb actor, carrying out what we prescribed. This gives us a position of power and control, yet it can feel stressful when we hit seemingly impenetrable walls. Mysterious errors whispering, "*You should be in control. But you don't seem to be.*", which is frustrating at best.

As a professional programmer I have experienced various kinds of frustration and blockage. Some are subtle, slow creepers that cause misery on the long run. [1] Others are more upfront, causing paralysis and bringing progress to a grinding halt. [2]

At times I was fruitlessly pondering over designs for days, and the end result was not necessary better than making an arbitrary design choice quickly. Often it seemed that I just can't bring myself to start working on a given task for mysterious reasons. At other times I was stressed by having to redesign and code a fundamental chunk of my team's product due to security concerns – failure would have put the team's work at risk.

Over time, after a considerable amount of introspection, I managed to overcome these issues. I would like to share my tips & tricks with my fellow developers (or those aspiring to be). I hope that this will help you recognize the causes of anxiety and take action against them, or even avoid them in the first place.

[1] Cutting corners, accumulating code debt, avoiding confrontation, or neglecting planning are quick examples.

[2] These often feel like running circles in your head, unable to break free from a track of thought.

Disclaimer

While this booklet hopes to offer relief, it is based on personal experience. If you feel anxiety you can't handle, depressed, or any other chronic anxiety symptoms, please seek a medical professional for effective and specialized help.

Sources of Anxiety

In this chapter I identify the most common causes of anxiety I perceive, and provide tips on shortcutting them.

Workplace Factors

Unclear expectations. If you don't know what your team expects from you, you can't act in your best capacity, and will likely waste effort. This is compounded when feedback cycles are long. Mitigation:

- Talk to your lead (or in lack of, teammates) about your goal for the current development cycle. [3]
- Make sure your assumptions and planned actions are OK. Do this in writing. A short email to the group mailing list might suffice, but a short shared document or wiki [4] would enable easier referencing and commenting.
- Once you have started actual development, seek feedback [5] every now and then (but don't annoy your colleagues).

[3] These come in different units, such as week, sprint, quarter, based on the team and methodology, if any.

[4] Design Doc, RFC.

[5] For example in the form of code reviews, smallchat, or internal demo.

* * *

Time pressure on its own is rarely a genuine factor of anxiety, but it can exaggerate existing problems. In such cases it becomes easy to blame the lack of time for these shortcomings. [6] Even though there is often a sense of urgency in development, the fact is that slipping schedules are more common than not, and management usually factors that in. [7] Mitigation:

- Don't work in single long stretches, use a focus-relax-focus cycle. Especially when under pressure, we tend to stick to the keyboard, which is counter-productive on the long run.

[6] Maybe there is not enough time, because unimportant features had not been pruned, or the scope of a feature not properly limited. How about an important but overly slow regression test that makes the team sit idle for hours?

[7] That said, this is not an excuse to be sloppy, but rather a reminder to not feel disappointed when progress is slower than anticipated.

- Seek to remove bottlenecks, improve processes, or cut features instead of burning extra hours.
- Push back on conflicting requests, and clearly (but politely) tell that you want to complete your current task. Be open to help if offered.
- Delegate to team members. You can find someone willing to take over more often than expected.
- If the pressure is due to tasks competing with one another, try to slot and timebox them to prevent endlessly starving a particular one. [8]

[8] For example, you need to develop new code for feature A, as well as sync with another team about feature B. You can drag on developing for days. Instead of that, allocate a time slot every day, say after lunch, for sorting non-coding tasks.

* * *

Time fragmentation prevents establishing a deep focus required to solve tasks. It can be frustrating to start on a task with a lingering thought of an interruption upcoming in 30 minutes.

- Defrag your time. Move meetings and small chores to compact blocks in your calendar, if you have the opportunity.
- Decline meetings where you can't contribute. Excuse and leave meetings early if possible. [9] At least, strive to make meetings efficient by requiring an agenda to be propagated upfront.
- Add *Do Not Disturb* time on your calendar. A block of 2 to 4 hours works best.
- Arrive early to get a jumpstart while noone else is there yet. Make your presence known, and leave earlier. [10]

[9] This works if such behavior is tolerated.

[10] Sending an email first thing in the morning leaves a timestamped record of your activity.

* * *

The sense of non-creative work can feel disheartening, but in reality much of our work is mechanical or repetitive. [11] As programmers we feel an urge to automate any kind of repetition, which is generally fine, but we can be overly trigger-happy and spend more time developing (and then fixing the bugs of) automation than the time it would have taken to just do the thing manually in the first place.

- As a rule of thumb, if you have to repeat a thing at most 5–20 times, just do it manually. [12] This has the benefit that if later automation turns out to be really needed, you will already have a good sense and have run into some of the edge cases as well. [13]

[11] See, you can't design a new data structure, or reinvent UIs every week. In fact, readers of your code and your users will be grateful for this.

[12] 3–5 times for larger tasks like adding a new submodule, and 15–20 times for smaller tasks like adding a new field to the server response.

[13] Slightly off-topic but related life hack: don't presume you will be a regular, say, of a gym. Go with spot tickets, then after ten occasions, buy a weekly pass; invest into a season pass only if a few weekly passes were used up.

* * *

Failure is perceived as something terrible, though it is a common part of life and software development especially. Sooner or later you will run into at least a small, user-visible hiccup. Mitigation:

- Realize that failure is part of everyday life. Also, don't confuse the failure of a piece of your work with your personal failure, even though we often use the phrase, "it's your fault". [14]
- One of our most important cognitive ability is learning from our mistakes. Investigate the causes of the failure, and prepare a list of actionable changes that will help avoiding such failures in the future.
- Share your lesson with others, write a post-mortem document. This will help others to avoid making the same mistake.

[14] The point is, the fact that some of your work was faulty doesn't make you an incapable or unreliable person. Especially if you demonstrate integrity by admitting the mistake.

Rest of the Chapter

Buy the full book to access the rest of this chapter:

- Fear of the Unknown
  - The lack of knowledge
  - Planning
  - Lack of vision
  - When things go wrong
- Procrastination and Blockage
  - Fearing to open the closet door
  - Multitude of choices
  - Unfamiliar tasks
  - Perceived complexity
  - Progress anxiety
  - Non-coding anxiety
- Generic Anxiety Relief
  - The base of Maslow's pyramid
  - Organize your thoughts
  - Defrag your mind
  - Take a small step

Bearing with Breakages

Not Breaking Things

Adding new features is exciting, but doing without breaking existing functionality is challenging.

Minimum-effort hacks come natural to programmers. Hacks emerge since we want to spare tedious plumbing work, which we perceive as the non-creative, repetitive and dull part of coding. [15]

What happens if we think of plumbing as an unwelcome intruder, ripping us of precious development time? We strive to finish hastily. We exhibit hacker pride for pushing a round peg through the square hole. [16] We take shortcuts, which rarely pays off on the long term.

- Don't give in to instincts, spend proper effort. Hacks convolute the codebase beyond comprehension and slow debugging.
- Notice bragging that stems from hacker pride. Take a step back, question the merits and point out possible drawbacks of the method.
- Batch modifications of a given code path to spare context switches, time and effort. [17]
- Accept plumbing as part of the job. It can rarely be circumvented. Don't delay a plumbing task by trying to automate it. [18]

* * *

Non-atomic upgrades require planning and supervision.

Atomic updates can prepare the newly released version upfront, and users can switch over smoothly. [19]

During a non-atomic upgrade, a part of the system – typically a single service or database – changes. Unchanged parts keep speaking the old protocol. [20]

[15] For example, propagating a new argument through a series of function calls, spanning remote services; adding a new field to the database and making its content reach the user-facing UI; copying and adjusting configuration files for bringing up new services.

[16] *"Instead of adding a new argument, let's just allow passing zero here, and enter alternative-entry-processing mode on orders worth more than $100."*

[17] Don't overbatch. A larger change adds cognitive load on reviewers, and prevent undoing modifications selectively.

[18] An effort to replace lots of plumbing with an allegedly simpler system must be agreed upon and scheduled. It can't be a gut reaction to a single tedious plumbing task.

[19] Glitches still occur. For example, browsers have to load the new frontend.

[20] Imagine removing a request option in the server. There are likely clients around who still send that option.

- Upgrade in multiple steps. First, the server must tolerate the presence of new options (or absence of the removed). Gradually switch clients to use the new protocol.
- Don't delete support code for the old behavior. Rolling back should be a configuration reload away. [21]
- Stateful components can rarely be swapped atomically. Keep writing both in parallel. Once the new system is up to date, switch over to reading from it.
- If everything goes without a glitch, remove support of the old protocol. Stop writing the old database.
- When planning, add buffer time for the migration steps.
- Don't complicate life if a service can tolerate maintenance downtime.

[21] In contrast to a sequence of merging, rebuilding and re-releasing the old version.

* * *

Coupled data representation becomes a problem once the representations start to diverge. Early during feature development, the database-, server- and client-side representation of a data object match closely. We are tempted to share this single object, instead of making copies and writing boring one-to-one transformation code.

These different formats unavoidably diverge. [22] Additional fields occur on the same object, with field comments striving to clarify usage. [23]

- Notice signs of coupling, for example field comments trying to clarify conditions of presence.
- Separate representation of different tiers from the beginning. Accumulated dependencies tax late separation.

[22] The database representation gets a sequence id added; server logic denormalizes extra data into the object to save client roundtrips; the frontend receives the color to display the item with.

[23] *"required when writing to the database"* or *"only present in the frontend"* .

Rest of the Chapter

Buy the full book to access the rest of this chapter:

- Your are not a Java Programmer
  - Slotting ourselves
  - Language slotting

  - Role slotting
- Depending on Open-Source
  - The One Who Depends On It Fixes It
  - Your dependency is the transitive closure
  - Living on the edge
  - False Promises
  - Compatible Licence

Down by Debugging

A Wild Bug Appears!

How do we get to know about an error? More importantly, how is it possible that we know about the existence of an error, but have no clue about its cause? Let's examine the ways how errors come to our attention!

User bug reports are frequent messengers. The user describes his perception, which is not a direct mapping to software concepts. [24] Information is often partial, missing some preconditions to reproduce the error.

[24] "Then I submitted the form" might mean that the user clicked the `Submit` button, but actually failed to notice a small red text notifying about missing required data. For a programmer, submission clearly didn't happen.

- Verify existence of the error. Try to reproduce yourself, or look for supporting facts in log files or database state. If you didn't succeed, ask for more information.
- What is the exact version of the software used (if applicable)?
- What are specific steps to reproduce?
- Does the error happen all the time? When did it happen last time?
- Are there specific conditions observed for the error to happen?
- What is the larger context – what was the user trying to achieve?

* * *

Test failures are reliable indicators, and if ran periodically, they pinpoint a relatively small set of changes where the error could have been introduced.

- Find the last source code state where the tests passed. If such information is not available in some central dashboard, find it manually. [25].

[25] Use a binary search on the versions, or specific tools like `git bisect`.

- Have a look at the test case, and browse through relevant files. Start with files in the stack trace.
- Look for recent changes in the source code of those files. If the changes are hard to grok, just read through the commit messages – maybe they reveal a relevant change.
- It makes sense to limit your investigation to files of your project. If that didn't bring a suspect, inspect repository-wide changes. Check for non-source changes as well, such as data files, binary artifacts or version updates.

* * *

Observed oddities should not be overlooked. An odd line in the log file can be the tip of an iceberg. Some examples to look for:

- Glitchy behavior. For example a button that needs to be clicked twice occasionally, or a tool that segfaults the first time it runs a given day.
- Orthogonal sources of information conflicting. For example the count of some entries in a log file, and a variable counting such occurrences differing wildly.
- A behavior sequence that, according to our knowledge about the code logic, should not happen.

We often catch oddities when we are not looking for them – more precisely, when we are looking for something else, say a different bug.

- However distracting, don't ignore oddities. If can't see to them instantly, open a quick issue about the observed behavior and come back later.

* * *

Random crashes are hard to examine after the fact. A random crash can either be easier or harder to investigate than a random glitch: The immediate condition triggering the crash just happened, so we should find the most relevant information at the end of the log files. [26] On the other hand, a sudden crash might not leave opportunity for recent logs or other diagnostics to be written.

[26] Not always. A slow but steady memory leak builds up over the course of the whole execution. A null-pointer dereferenced might have been set a long time ago.

Search for a Precedent

We warm up our fingers, getting ready to dive deep in code. But let's take a step back, and carry out due diligence on the observation.

> *Movie Blue Streak. Thieves in front of a safe.*
>
> *Miles:* What's the first thing you do?
> *Eddie:* Drill the lock.
> *Miles:* No! You got to check to see if it's open.

The issue tracker might already contain our bug. Search it for specific error messages. Also pick three sets of keywords, each relating to how you would report the bug yourself. [27]Then search using these keywords.

- Don't forget to search for closed issues as well. [28]
- For generic errors, StackOverflow or other public sites could help. [29]

[27] For example, if the bug was about a glitchy stop button on the UI, search for *"UI stop button glitch"*, *"UI button doesn't respond"* and *"UI concurrency race"*.

[28] It happened to me that the bug was already fixed on an older release branch.

[29] Take care not to leak internal information by pasting telltale names in search boxes.

* * *

Is it a bug? Before proceeding further, check:

- Do you use any experimental, unofficial or unreleased versions of libraries or other components? These might not be production-ready. Moreover, you won't get support, or not with priority.
- Do you **newly** invoke a program with nonstandard arguments, or call complicated methods? Double-check your usage.
- Do you rely on undocumented behavior?
- Does the bug manifest only on your computer? If so, be wary and check for environment differences.
- Is the program built from the right sources? Right branch? Are modifications picked up?
- Does the program pick up the right configuration?
- Does it link against the right shared libraries? [30] Does it link against multiple versions of the same library along the dependency tree?

[30] Use `ldd` to find out.

* * *

Folklore might reveal insight. Once you are confident it is a real bug:

- Ask around in the team.
- Ask owners of the suspected piece of code.
- If you are in a hurry, open an issue with your initial findings and suspicions against the team owning the suspected code, and start debugging yourself as well. [31]

[31] If it is not urgent, work on other tasks until the team comes back with a reply.

Rest of the Chapter

Buy the full book to access the rest of this chapter:

- Online or Offline?
  - Online debugging
  - Offline debugging
- Preparations
  - Expectations
  - Collect logs
  - Clean logs
  - Grab an A3 sized blank paper
- Where is the bug?
  - Reproduce and reduce
  - Backward Method
    - The Backward Method
  - Forward Method
    - The Forward Method
    - Pruning noise
    - Parallels
    - Explore the parameter space
    - Alternative implementations
  - Endgame
    - When stuck
    - When found the bug
- Preventing bugs

Full Contents

Thank you for reading through this book sample!

If you find the content useful, please subscribe at `https://treetide.com/book/programming-without-anxiety` to get relevant news and progress updates.

You can also prepurchase at `https://pay.paddle.com/checkout/547112` and access content as it gets written!

Full table of contents:

- Preface
  - Disclaimer
- Sources of Anxiety
  - Workplace Factors
    - Unclear expectations
    - Time pressure
    - Time fragmentation
    - The sense of non-creative work
    - Failure
  - Fear of the Unknown
    - The lack of knowledge
    - Planning
    - Lack of vision
    - When things go wrong
  - Procrastination and Blockage
    - Fearing to open the closet door
    - Multitude of choices
    - Unfamiliar tasks
    - Perceived complexity

    * Progress anxiety
    * Non-coding anxiety
  – Generic Anxiety Relief
    * The base of Maslow's pyramid
    * Organize your thoughts
    * Defrag your mind
    * Take a small step
- Bearing with Breakages
  – Not Breaking Things
    * Minimum-effort hacks
    * Non-atomic upgrades
    * Coupled data representation
  – Your are not a Java Programmer
    * Slotting ourselves
    * Language slotting
    * Role slotting
  – Depending on Open-Source
    * The One Who Depends On It Fixes It
    * Your dependency is the transitive closure
    * Living on the edge
    * False Promises
    * Compatible Licence
- Down by Debugging
  – A Wild Bug Appears!
    * User bug reports
    * Test failures
    * Observed oddities
    * Random crashes
  – Search for a Precedent
    * The issue tracker
    * Is it a bug?
    * Folklore
  – Online or Offline?
    * Online debugging
    * Offline debugging
  – Preparations

        * Expectations
        * Collect logs
        * Clean logs
        * Grab an A3 sized blank paper
    - Where is the bug?
        * Reproduce and reduce
        * Backward Method
            · The Backward Method
        * Forward Method
            · The Forward Method
            · Pruning noise
            · Parallels
            · Explore the parameter space
            · Alternative implementations
        * Endgame
            · When stuck
            · When found the bug
    - Preventing bugs
- Closing words
- Everyday Productivity (TBD)
    - Workspace Organization
    - Interrupts and Regaining Productivity
    - Dealing with Feature Requests
    - Work-Rest Balance
    - Coping with Coding
    - Peek at Production
    - Gathering Stats
- In a Team Setting (TBD)
    - Giving and Receiving Feedback
    - Interfacing with Other Teams
    - Effective Use of Meetings
    - Drinking Coffee
    - Tech Leading
    - Project Frustrations
- Outside Work (TBD)

  - Hobby Projects
  - Family and Chores
  - Programmer Personality
- Appendix: Mastering the Command Line (TBD)