Dr. Arne Babenhauserheide / draketo.de

Write programs you can still hack when you feel dumb

. . . so you can still understand and change them, when you can only work half an hour

> *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.* — Brian Kernighan

In the article Hyperfocus and balance, Arc Riley from PySoy talks about trying to get to the **Hyperfocus state** without endangering his health. Since I have similar needs, I am developing some strategies for that myself *(though not for my health, but because my wife and children can't be expected to let me work 8h without any interruptions in my free time)*.

Different from Arc, I try to **change my programming habits** instead of changing myself to fit to the requirements of my habits.[1]

Easy times

Let's begin with **Programming while you feel great**.

The guideline I learned from writing PnP roleplaying games is to keep the number of things to know at **7 or less** at each point (according to Miller, 1956; though the current best guess of the limitation for average humans is only 4 objects!). For a function of code I would convert that as follows:

1. You need to keep in mind the **function** you work in *(location)*, and
2. the **task** it should perform *(purpose and effect)*, and
3. the **resources** it uses *(arguments or global values/class attributes)*.

[1] Where I got bitten badly by my high-performance coding habits is the keyboard layout evolution program. I did not catch my error when the structure grew too complex (while adding stuff), and now that I do not have as much uninterrupted time as before, I cannot work on it efficiently anymore. I'm glad that this happened with a mostly finished project on whoose evolution no ones future depended. Still it is sad that this will keep me from turning it into a realtime visual layout optimizer. I can still work on its existing functionality (I kept improving it for the most importang task: the cost calculation), but adding new functionality is a huge pain.

Only 4 things left for the code of your function. *(three if you use both class attributes/global values and function arguments. Two, if you have complex custom data-structures with peculiar names or access-methods which you have to understand for doing anything. One if you also have to remember the commands of an unfamiliar[2, 3, 4] editor or VCS tool. See how fast this approaches zero even when starting with 7 things?)*

Add an if-switch, for-loop or similar and you have only 3 things left.

You need those for what the function should actually do, so better put further complexities into subfunctions.

Also ensure that each of the things you work with is easy enough. If you get the things you use down to 7 by writing functions with 20 arguments, you don't win anything. Just the resources you could use in the function will blow your mind when you try to change the function a few months later. This goes for every part of your program: The number of *functions*, the number of *function arguments*, the number of *variables*, the *lines of code per function* and even the number of *hierarchy levels* you use to reduce the other things you need to keep in mind at any given time.

Hard times

But if you want to be able to hack that code while you feel dumb (compared to those streaks of genius when you can actually hold the whole structure of your program in your head and forsee every effect of a given change before actually doing it), you need to make sure that you don't have to take all 7 things into account.

[2] This limit only applies to unfamiliar things: things you did not yet learn well enough that they work automatically. Once you know a tool well enough that you don't have to think about it anymore, it no longer counts against the 7 thing limit, since you don't need to remember it. That's strong support for writing conventional code — or at least code you'll still write similarly a decade later — and using tools which can accompany you for a long time.

[3] This is reasoning from experience. I think the actual reason why people can juggle large familiar structures is more likely that they have an established mental model which allows them to use multiple dimensions and cut the amount of bits you need for referring to the thing. See the **Absolute Judgments of Multidimensional Stimuli** section, the **recoding** section and the difference between **chunks and bits** in George A. Miller (1956). This is part of writing programs you can still hack when you feel dumb — but one which only helps those who use the same structures and one which binds you to your established coding style.

[4] Aside from being able to remember the full mental model, it is often enough to remember something close enough and then find the correct answer with assisted guessing. A typical example is narrowing down auto-completion candidates by matching on likely names until something feels right. This is how good auto-completion — or rather: guided interactive code inspection — massively expands the size of models we can work with efficiently. It depends on easily guessable naming, typically aided by experience, and it benefits from tools which can limit or order the potential candidates by the context. With good tool-support it suffices to have a general feeling about the direction to take for doing something. The guidelines in this article should help you with guessing, and should help your tool with limiting candidates to plausible choices and with ordering them by context.

Tune it down for **the times when you feel dumb** by starting with **5 things**.[5] After substracting one for the location, for the task and for the resources, you are left with only two things:

> **Two things for your function. Some Logic and calling stuff are 2 things.**

If it is an if-switch, let it be just an if-switch calling other functions.[6] Yes, it may feel much easier to do it directly here, when you are fully embedded in your code and feel great, but it will bite you when you are down. Which is exactly when you won't want to be bitten by your own code.

Loose coupling and tight cohesion

> *Programming is a constant battle against complexity. Stumble from the sweet spot of your program into any direction, and complexity raises its ugly head. But finding the sweet spot requires constant vigilance, as it shifts with the size and structure of your program and your development group.*

To find a practical way of achieving this, Django's concept of loose coupling and tight cohesion (more detailed) helped me most, because it reduces the interdependencies.

> **The effects of any given change should be contained in the part of the code you work in - and in one type of code.**

As web framework, Django seperates the templates, the URI definitions, the program code and the database access from each other. (see how these are already 4 categories, hitting the limit of our mind again?)

For a game on the other hand, you might want to seperate story, game logic, presentation (what you see on the screen) and input/user actions. Also people who write a scenario

[5] See how I actually don't get below 5 here? A good TODO list which shows you the task so you can forget it while coding *might* get you down to 4. But don't bet on it. Not knowing where you are or where you want to go are recipes for desaster... And if you make your functions *too small*, the collection of functions gets more complex, or the object hierarchy too deep, adding complexity at other places and making it *harder to change the structure* (refactor) when requirements change. Well, no one said creating well-structured programs would be easy. You need to find the right compromise for you.

[6] Keeping functions simple does not mean that they must be extremely short. If you have a library which provides many tools that get used for things like labelling axes in a plot, and you don't get much repetition between different functions, then having a function of 20 to 30 lines can be simpler than building an abstraction which only works at the current state of the code but will likely break when you add the next function. This is inherent, function-local complexity: you cannot reduce it with structure. Therefore the sweet spot of simplicity for some tasks is using medium-sized functions. If you find yourself repeating exactly the same code multiple times, however, you likely missed the sweet spot and should investigate shortening the functions by extracting the common tasks, or restructuring the function to separate semantically different tasks.

or level should only have to work in one type of code, neatly confined in one file or a small set of files which reside in the same place.

And for a scientific program, data input, task definition, processing and data output might be separated.

Remember that this separation does not only mean that you put those parts of the code into different files, but that they are loosely coupled:[7]

> **They only use lean and clearly defined interfaces and don't need to know much about each other.**

Conclusions

This strategy does not only make your program easier to adapt (because the parts you need to change for implementing a given feature are smaller). If you apply it not only to the bigger structure, but to every part of the program, it's main advantage is that any part of the code can be understood without having to understand other parts.

And you can still understand and hack your code, when your child is sick, your wife is overworked, you slept 3 hours the night before - and can only work for half an hour straight, because it's evening and you don't want to be a creep (but this change has to be finished nontheless).

Note that finding a design which accomplishes this is far more complex than it sounds. If people can read your code and say *"oh, that's easy. I can hack that"* (and manage to do so), then you did it right.

> **Designing a simple structure to solve a complex task is far harder than designing a complex structure to solve that task.**

[7] In all your structures, do keep program performance in mind. If your structure imposes high performance penalties, you will have to break it more and more as you push it beyond the limits you deemed reasonable at the beginning. And then it adds complexity instead of reducing it. When programming, you always have two audiences. One are **humans**: your program must be easy to understand and change. If it is not, it will rot. The other is **the machine**: your program must be sufficiently efficient to execute. If it is not, that will bite you when you push it where it was never meant to go. *And you will.* If it grows somewhat successful and you get any competition, even if it is much worse, you cannot afford a rewrite. The full rewrite is the number one strategic mistake you should never make. So while you keep one eye on easy structures for humans, keep the other eye on performance for the machine.

And being able to hack your program while you feel dumb (and maybe even hold it in your head) is worth investing some of your genius-time[8, 9] into your design (and repeating that whenever your code grows too hairy).

PS (7 years later): This only applies to the version of your code that stays in your codebase. During short-term experiments these rules do not apply, because there you still have the newly written code in your head. But take pains to clean it up before it takes on a life of its own. The last point in time for that is when you realize that you're no longer sure how it works (then you know that you already missed the point of refactoring, but you can at least save your colleagues and your future self from stumbling even worse than you do at that moment). That way you also always have some leeway in short-term complexity that you can use during future experimentation. Also don't make your code too simple: If you find that you're bored while coding or that you spend more time fighting the structures you built than solving the actual problems, you took these principles too far, because you're no longer getting full benefits from your brain. Well chosen local complexity reduces global complexity and the required work per change.

List of Links

[8] How to find your genius time? That's a tautology: Your genius time is when you can hold your program in your mind. If I could tell you when your genius time occurs, or even how to trigger it, I could make lots of money by consulting about every tech company in existence. A good starting point is reading about "flow", known in many other creative activities (some starting points). Reaching the flow often includes spending time outside the flow, so best write programs you can still hack when you feel dumb.

[9] And in all this reduction of local complexity, **keep in mind that there is no silver bullet (Brooks, 1986)**. Just take care that you **design** your code against the limits of the humans who work with it, and only in the second place against the limits of the tools you use — you can change the tools, but you cannot easily change the humans; often you cannot change the humans at all. In the best case you can make your tools fit and expand the limits of humans. But remember also that your code must run **well enough** on the machine. And you often do not know what "well enough" means. I know that this is not a simple answer. If that irks you, **keep in mind that there is no silver bullet (Brooks, 1986)**, and this text isn't one either. It's just a step on the way — I hope it is useful to you.

!