Write code that is easy to delete, not easy to extend.

> *"Every line of code is written without reason, maintained out of weakness, and deleted by chance" Jean-Paul Sartre's Programming in ANSI C.*

Every line of code written comes at a price: maintenance. To avoid paying for a lot of code, we build reusable software. The problem with code re-use is that it gets in the way of changing your mind later on.

The more consumers of an API you have, the more code you must rewrite to introduce changes. Similarly, the more you rely on an third-party api, the more you suffer when it changes. Managing how the code fits together, or which parts depend on others, is a significant problem in large scale systems, and it gets harder as your project grows older.

> *My point today is that, if we wish to count lines of code, we should not regard them as "lines produced" but as "lines spent" EWD 1036*

If we see 'lines of code' as 'lines spent', then when we delete lines of code, we are lowering the cost of maintenance. Instead of building re-usable software, we should try to build disposable software.

I don't need to tell you that deleting code is more fun than writing it.

To write code that's easy to delete: repeat yourself to avoid creating dependencies, but don't repeat yourself to manage them. Layer your code too: build simple-to-use APIs out of simpler-to-implement but clumsy-to-use parts. Split your code: isolate the hard-to-write and the likely-to-change parts from the rest of the code, and each other. Don't hard code every choice, and maybe allow changing a few at runtime. Don't try to do all of these things at the same time, and maybe don't write so much code in the first place.

Step 0: Don’t write code

The number of lines of code doesn’t tell us much on its own, but the magnitude does 50, 500 5,000, 10,000, 25,000, etc. A million line monolith is going to be more annoying than a ten thousand line one and significantly more time, money, and effort to replace.

Although the more code you have the harder it is to get rid of, saving one line of code saves absolutely nothing on its own.

Even so, the easiest code to delete is the code you avoided writing in the first place.

Step 1: Copy-paste code

Building reusable code is something that’s easier to do in hindsight with a couple of examples of use in the code base, than foresight of ones you might want later. On the plus side, you’re probably re-using a lot of code already by just using the file-system, why worry that much? A little redundancy is healthy.

It’s good to copy-paste code a couple of times, rather than making a library function, just to get a handle on how it will be used. Once you make something a shared API, you make it harder to change.

The code that calls your function will rely on both the intentional and the unintentional behaviours of the implementation behind it. The programmers using your function will not rely on what you document, but what they observe.

It’s simpler to delete the code inside a function than it is to delete a function.

Step 2: Don’t copy paste code

When you’ve copy and pasted something enough times, maybe it’s time to pull it up to a function. This is the “save me from my standard library” stuff: the “open a config file and give me a hash table”, “delete this directory”. This includes functions without any state, or functions with a little bit of global knowledge like environment variables. The stuff that ends up in a file called “util”.

Aside: Make a `util` directory and keep different utilities in different files. A single `util` file will always grow until it is too big and yet too hard to split apart. Using a single `util` file is unhygienic.

The less specific the code is to your application or project, the easier they are to re-use and the less likely to change or be deleted. Library code like logging, or third party APIs, file handles, or processes. Other good examples of code you’re not going to delete are lists, hash tables, and other collections. Not because they often have very simple interfaces, but because they don’t grow in scope over time.

Instead of making code easy-to-delete, we are trying to keep the hard-to-delete parts as far away as possible from the easy-to-delete parts.

Step 3: Write more boilerplate

Despite writing libraries to avoid copy pasting, we often end up writing a lot more code through copy paste to use them, but we give it a different name: boilerplate. Boiler plate is a lot like copy-pasting, but you change some of the code in a different place each time, rather than the same bit over and over.

Like with copy paste, we are duplicating parts of code to avoid introducing dependencies, gain flexibility, and pay for it in verbosity.

Libraries that require boilerplate are often stuff like network protocols, wire formats, or parsing kits, stuff where it’s hard to interweave policy (what a program should do), and protocol (what a program can do) together without limiting the options. This code is hard to delete: it’s often a requirement for talking to another computer or handling different files, and the last thing we want to do is litter it with business logic.

This is not an exercise in code reuse: we're trying keep the parts that change frequently, away from the parts that are relatively static. Minimising the dependencies or responsibilities of library code, even if we have to write boilerplate to use it.

You are writing more lines of code, but you are writing those lines of code in the easy-to-delete parts.

Step 4: Don't write boilerplate

Boilerplate works best when libraries are expected to cater to all tastes, but sometimes there is just too much duplication. It's time to wrap your flexible library with one that has opinions on policy, workflow, and state. Building simple-to-use APIs is about turning your boilerplate into a library.

This isn't as uncommon as you might think: One of the most popular and beloved python http clients, `requests`, is a successful example of providing a simpler interface, powered by a more verbose-to-use library `urllib3` underneath. `requests` caters to common workflows when using http, and hides many practical details from the user. Meanwhile, `urllib3` does the pipelining, connection management, and does not hide anything from the user.

It is not so much that we are hiding detail when we wrap one library in another, but we are separating concerns: `requests` is about popular http adventures, `urllib3` is about giving you the tools to choose your own adventure.

I'm not advocating you go out and create a `/protocol/` and a `/policy/` directory, but you do want to try and keep your `util` directory free of business logic, and build simpler-to-use libraries on top of simpler-to-implement ones. You don't have to finish writing one library to start writing another atop.

It's often good to wrap third party libraries too, even if they aren't protocol-esque. You can build a library that suits your code, rather than lock in your choice across the project. Building a pleasant to use API and building an extensible API are often at odds with each other.

This split of concerns allows us to make some users happy without making things impossible for other users. Layering is easiest when you start with a good API, but writing a good API on top of a bad one is unpleasantly hard. Good APIs are designed with empathy for the programmers who will use it, and layering is realising we can't please everyone at once.

Layering is less about writing code we can delete later, but making the hard to delete code pleasant to use (without contaminating it with business logic).

Step 5: Write a big lump of code

You've copy-pasted, you've refactored, you've layered, you've composed, but the code still has to do something at the end of the day. Sometimes it's best just to give up and write a substantial amount of trashy code to hold the rest together.

Business logic is code characterised by a never ending series of edge cases and quick and dirty hacks. This is fine. I am ok with this. Other styles like 'game code', or 'founder code' are the same thing: cutting corners to save a considerable amount of time.

The reason? Sometimes it's easier to delete one big mistake than try to delete 18 smaller interleaved mistakes. A lot of programming is exploratory, and it's quicker to get it wrong a few times and iterate than think to get it right first time.

This is especially true of more fun or creative endeavours. If you're writing your first game: don't write an engine. Similarly, don't write a web framework before writing an application. Go and write a mess the first time. Unless you're psychic you won't know how to split it up.

Monorepos are a similar tradeoff: You won't know how to split up your code in advance, and frankly one large mistake is easier to deploy than 20 tightly coupled ones.

When you know what code is going to be abandoned soon, deleted, or easily replaced, you can cut a lot more corners. Especially if you make one-off client sites, event web pages. Anything where you have a template and stamp out copies, or where you fill in the gaps left by a framework.

I'm not suggesting you write the same ball of mud ten times over, perfecting your mistakes. To quote Perlis: "Everything should be built top-down, except the first time". You should be trying to make new mistakes each time, take new risks, and slowly build up through iteration.

Becoming a professional software developer is accumulating a back-catalogue of regrets and mistakes. You learn nothing from success. It is not that you know what good code looks like, but the scars of bad code are fresh in your mind.

Projects either fail or become legacy code eventually anyway. Failure happens more than success. It's quicker to write ten big balls of mud and see where it gets you than try to polish a single turd.

It's easier to delete all of the code than to delete it piecewise.

Step 6: Break your code into pieces

Big balls of mud are the easiest to build but the most expensive to maintain. What feels like a simple change ends up touching almost every part of the code base in an ad-hoc fashion. What was easy to delete as a whole is now impossible to delete piecewise.

In the same we have layered our code to separate responsibilities, from platform specific to domain specific, we need to find a means to tease apart the logic atop.

> *[Start] with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such*

> *a decision from the others. D. Parnas*

Instead of breaking code into parts with common functionality, we break code apart by what it does not share with the rest. We isolate the most frustrating parts to write, maintain, or delete away from each other.

We are not building modules around being able to re-use them, but being able to change them.

Unfortunately, some problems are more intertwined and hard to separate than others. Although the single responsibility principle suggests that 'each module should only handle one hard problem', it is more important that 'each hard problem is only handled by one module'

When a module does two things, it is usually because changing one part requires changing the other. It is often easier to have one awful component with a simple interface, than two components requiring a careful co-ordination between them.

> *I shall not today attempt further to define the kinds of material I understand to be embraced within that shorthand description ["loose coupling"], and perhaps I could never succeed in intelligibly doing so. But I know it when I see it, and the code base involved in this case is not that. SCOTUS Justice Stewart*

A system where you can delete parts without rewriting others is often called loosely coupled, but it's a lot easier to explain what one looks like rather than how to build it in the first place.

Even hardcoding a variable *once* can be loose coupling, or using a command line flag over a variable. Loose coupling is about being able to change your mind without changing too much code.

For example, Microsoft Windows has internal and external APIs for this very purpose. The external APIs are tied to the lifecycle of desktop programs, and the internal API is tied to the underlying kernel. Hiding these APIs away gives Microsoft flexibility without breaking too much software in the process.

HTTP has examples of loose coupling too: Putting a cache in front of your HTTP server. Moving your images to a CDN and just changing the links to them. Neither breaks the browser.

HTTP's error codes are another example of loose coupling: common problems across web servers have unique codes. When you get a 400 error, doing it again will get the same result. A 500 may change. As a result, HTTP clients can handle many errors on the programmers behalf.

How your software handles failure must be taken into account when decomposing it into smaller pieces. Doing so is easier said than done.

> *I have decided, reluctantly to use L^aT_eX. Making reliable distributed systems in the presence of software errors. Armstrong, 2003*

Erlang/OTP is relatively unique in how it chooses to handle failure: supervision trees. Roughly, each process in an Erlang system is started by and watched by a supervisor. When a process encounters a problem, it exits. When a process exits, it is restarted by the supervisor.

(These supervisors are started by a bootstrap process, and when a supervisor encounters a fault, it is restarted by the bootstrap process)

The key idea is that it is quicker to fail-fast and restart than it is to handle errors. Error handling like this may seem counter-intuitive, gaining reliability by giving up when errors happen, but turning things off-and-on again has a knack for suppressing transient faults.

Error handling, and recovery are best done at the outer layers of your code base. This is known as the end-to-end principle. The end-to-end principle argues that it is easier to handle failure at the far ends of a connection than anywhere in the middle. If you have any handling inside, you still have to do the final top level check. If every layer atop must handle errors, so why bother handling them on the inside?

Error handling is one of the many ways in which a system can be tightly bound together. There are many other examples of tight coupling, but it is a little unfair to single one out as being badly designed. Except for IMAP.

In IMAP almost every each operation is a snowflake, with unique options and handling. Error handling is painful: errors can come halfway through the result of another operation.

Instead of UUIDs, IMAP generates unique tokens to identify each message. These can change halfway through the result of an operation too. Many operations are not atomic. It took more than 25 years to get a way to move email from one folder to another that reliably works. There is a special UTF-7 encoding, and a unique base64 encoding too.

I am not making any of this up.

By comparison, both file systems and databases make much better examples of remote storage. With a file system, you have a fixed set of operations, but a multitude of objects you can operate on.

Although SQL may seem like a much broader interface than a filesystem, it follows the same pattern. A number of operations on sets, and a multitude of rows to operate on. Although you can’t always swap out one database for another, it is easier to find something that works with SQL over any homebrew query language.

Other examples of loose coupling are other systems with middleware, or filters and pipelines. For example, Twitter’s Finagle uses a common API for services, and this allows generic timeout handling, retry mechanisms, and authentication checks to be added effortlessly to client and server code.

(I’m sure if I didn’t mention the UNIX pipeline here someone would complain at me)

First we layered our code, but now some of those layers share an interface: a common set of behaviours and operations with a variety of implementations. Good examples of loose coupling are often examples of uniform interfaces.

A healthy code base doesn’t have to be perfectly modular. The modular bit makes it way more fun to write code, in the same way that Lego bricks are fun because they all fit together. A healthy code base has some verbosity, some redundancy, and just enough distance between the moving parts so you won’t trap your hands inside.

Code that is loosely coupled isn’t necessarily easy-to-delete, but it is much easier to replace, and much easier to change too.

Step 7: Keep writing code

Being able to write new code without dealing with old code makes it far easier to experiment with new ideas. It isn’t so much that you should write microservices and not monoliths, but your system should be capable of supporting one or two experiments atop while you work out what you’re doing.

Feature flags are one way to change your mind later. Although feature flags are seen as ways to experiment with features, they allow you to deploy changes without re-deploying your software.

Google Chrome is a spectacular example of the benefits they bring. They found that the hardest part of keeping a regular release cycle, was the time it took to merge long lived feature branches in.

By being able to turn the new code on-and-off without recompiling, larger changes could be broken down into smaller merges without impacting existing code. With new features appearing earlier in the same code base, it made it more obvious when long running feature developement would impact other parts of the code.

A feature flag isn’t just a command line switch, it’s a way of decoupling feature releases from merging branches, and decoupling feature releases from deploying code. Being able to change your mind at runtime becomes increasingly important when it can take hours, days, or weeks to roll out new software. Ask any SRE: Any system that can wake you up at night is one worth being able to control at runtime.

It isn't so much that you're iterating, but you have a feedback loop. It is not so much you are building modules to re-use, but isolating components for change. Handling change is not just developing new features but getting rid of old ones too. Writing extensible code is hoping that in three months time, you got everything right. Writing code you can delete is working on the opposite assumption.

The strategies i've talked about — layering, isolation, common interfaces, composition — are not about writing good software, but how to build software that can change over time.

> *The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. […] Hence plan to throw one away; you will, anyhow. Fred Brooks*

You don't need to throw it all away but you will need to delete some of it. Good code isn't about getting it right the first time. Good code is just legacy code that doesn't get in the way.

Good code is easy to delete.

Acknowledgments

Thank you to all of my proof readers for your time, patience, and effort.

Further Reading

Layering/Decomposition

On the Criteria To Be Used in Decomposing Systems into Modules, D.L. Parnas.

How To Design A Good API and Why it Matters, J. Bloch.

The Little Manual of API Design, J. Blanchette.

Python for Humans, K. Reitz.

Common Interfaces

The Design of the MH Mail System, a Rand technical report.

The Styx Architecture for Distributed Systems

Your Server as a Function, M. Eriksen.

Feedback loops/Operations lifecycle

Chrome Release Cycle, A. Laforge.

Why Do Computers Stop and What Can Be Done About It?, J. Gray.

How Complex Systems Fail, R. I. Cook.

The technical is social before it is technical.

All Late Projects Are the Same, Software Engineering: An Idea Whose Time Has Come and Gone?, T. DeMarco.

Epigrams in Programming, A. Perlis.

How Do Committees Invent?, M.E. Conway.

The Tyranny of Structurelessness, J. Freeman