Modularity

Contents

Modularity concepts

Coupling = strength of dependencies between different modules. Loose/weak coupling is good, strong coupling is bad.

Cohesion = within one module, how well do its various responsibilities mesh (fit together)? Its goals are all aligned, not disparate. **Separation of concerns.**

Some researchers work on **quantifying** these concepts. But no way of quantifying has caught on universally. Not clear that the proposed metrics correlate with code quality.

Top-down vs bottom-up

In software development, **top-down** refers to planning out modules and their responsibilities from a *bird's-eye view*. In contrast, **bottom-up** means coding (or experimenting with) some fragments of functionality that we think we'll need.

Most projects proceed with **a combination of both.** If you do *too much* top-down before starting to code, it's easy to overlook critical details or constraints that only become apparent on the ground. If you do *too much* bottom-up without planning, you can spend time building and debugging modules that may not actually be needed, or are somehow misaligned with the project goals.

Pub-Sub

Widely-used pattern for decoupling modules that share some data.

Pub-Sub = Publish/Subscribe. Depends on a Message Broker who actually sends published data to the subscribers.

See `pubsub/` in repo on gitlab[1] for an example with a message broker in Javascript.

[1] gitlab.liu.edu/cs164s19/cs164pub/tree/master/pubsub

Decoupling producers and consumers

Example: pipes in a command-line, such as:

```
find DIR | grep .png | wc -l
```

Demo with Python generators – see `pygen/` in repo on gitlab. This calculates

$$\sum_{i=0}^{4}\sum_{j=0}^{3} i^2 j$$

but producing the pairs of numbers is separate from calculating the expression which is separate from summing them.

```python
def produce_pairs():
    for i in range(5):
        for j in range(4):
            print("PRODUCING", i, j)
            yield (i,j)

def calc_pair(gen):
    for i,j in gen:
        n = i*i*j
        print("CALCULATING", n)
        yield n

if __name__ == "__main__":
    print(sum(calc_pair(produce_pairs())))
```

The pipeline is specified right-to-left by function calls in that `__main__` block, and the print statements help us trace how it jumps between producer and consumer.

```
% python demo.py
PRODUCING 0 0
CALCULATING 0
PRODUCING 0 1
CALCULATING 0
[...]
PRODUCING 3 3
CALCULATING 27
PRODUCING 4 0
CALCULATING 0
PRODUCING 4 1
CALCULATING 16
PRODUCING 4 2
CALCULATING 32
PRODUCING 4 3
CALCULATING 48
180
```

Just to check that answer with a little algebra:

$$= \sum_{i=0}^{4}(0i^2 + 1i^2 + 2i^2 + 3i^2)$$

$$= \sum_{i=0}^{4}(i^2 + 2i^2 + 3i^2)$$

$$= \sum_{i=0}^{4}(1 + 2 + 3)i^2$$

$$= \sum_{i=0}^{4} 6i^2$$

$$= 6\left(\sum_{i=0}^{4} i^2\right)$$

$$= 6(0 + 1 + 4 + 9 + 16)$$

$$= 6(30)$$

$$= 180$$

Your notes & questions

According to the article, when McConnells said "keep it simple" is there a certain way to follow up with that or just break your code down into pieces,and develop the most efficient way?

> KISS = Keep It Simple & Stupid
>
> Do the simplest thing that could possibly work.
>
> There are well-known **patterns** of abstraction that can be applied. There are also **anti-patterns**.

CS 164 Check-in 4 In the "Part 1" article, we know that we need to focus on the long term health and maintainability of our code and to achieve this we need to reduce technical debt, so what that incurs Technical Debt, just fixing bugs or cleaning up the code?

In the "Part 2" article, we have to spend significant effort understanding and building software to solve complex problems. Any piece of non-trivial software quickly becomes too big to comprehend as a single unit of code. So, how do we use “abstraction” to manage complex code?

I like the quote at the end of the second post:

> Neither hierarchies nor abstractions reduce the total number of details in a program — they might actually increase the total number. Their benefit arises from organizing details in such a way that fewer details have to be considered at any particular time.

This shows the inherent trade-off that comes with abstraction: forgoing optimal solutions in favor of understandable ones. Of course, in most cases where abstraction is used, this trade-off is well worth it (at least I would hope so) since it can make problems substantially easier to think about, approach and solve, thus, as this post suggests, retain software development velocity over time. This even comes down to the very programming language with which you choose to build your software: software that needs an emphasis on speed and memory efficiency will most likely be written in a lower level language while software that has much more essential complexity and has to be maintained and worked on frequently will most likely be written in a higher level language.

Sometimes abstraction can have some rather disastrous effects. Abstraction often leads to even more abstraction and when you’re already in several layers, having to change something buried under all that abstraction becomes daunting. If anything, this further emphasizes the importance of paying off technical debt. Another issue that can arise abstraction is hiding rather important details. When you can hide really complex operations behind very simple and innocent looking lines of code, it becomes incredibly easy to trick yourself and anyone else working on the same code into thinking that your program is well optimized and efficient when in reality you just let a ridiculous amount of unnecessary overhead go over your head.

It is suggested that the best way to minimize technical debt is to constantly maintain the source code and remove bugs before they can interfere with new features. However, how can we determine when the source code is of feasible quality to *start* adding modules for new features? Unit testing can only do so much with so little interaction. Do we just start adding features and then do modifying?

Technical debt with interest rate that is not sustainable because if something isn’t fixed earlier than will continue being an issue.

Fix things immediately to maintain quality.

Simplicity, Dependencies, Hierarchy, Abstraction? Don't understand the bit about abstraction. The article mentions hiding unnecessary details but I don't know what is expected of the complexity that is needed most of the time.

How often should we be looking to prevent Dependencies interworking within a work file?