31.3 Modular Design

A tool for managing complexity.

Modular design is a powerful tool for managing complexity because it divides the project complexity into manageable pieces. One way to implement modular design is to create helper methods or interfaces. This way, the programmer can individually handle each component of complexity rather than having to always keep track of the details of every piece of code that they write.

Modules should be simple:

In an ideal world, every module is totally independent from one another. Unfortunately, this is not possible because code from each module needs to call other modules. However, we can still try to minimize these dependencies between modules! In other words, we want to minimize how many *things* you need to know about a given module in order to use it. This is exactly what we mean when we talk about the difference between *implementation* versus *interface*. A good module will not require the user to know the specific implementation in order to use it. Rather, it should be sufficient to just know the interface of the module. Changing the implementation of a module should not affect the interface.

John Ousterhout once said: "The best modules are those whose interfaces are much simpler than their implementation." This is a good rule to swear by, and putting it into practice will save a lot of headache.

One other technique of minimizing complexity is to restrict what the user can do. If a user does not *need* to interact with an instance variable, then don't give them access to it.

Interface rules:

Interfaces have a further set of rules. These rules are divided into *formal* and *informal* rules. The difference between the two is that informal rules are not enforced by the compiler.

Formal rules are the list of method signatures. If a method is not implemented in a class that implements the interface, then the compiler will give an error.

Some examples of informal rules are:

- If your iterator class does not call hasNext() on its own (for some reason) and instead requires the user to call it.
- Any exceptions that are thrown.
- Any runtime specifications.

Be especially wary of informal rules! They are hard to keep track of.

Modules should be deep

Another cool idea is that Modules should be deep. Their simple interfaces but powerful functionality. We do this a lot like thats the 61B story! A set for example is a deep module that has power functioanlity and simple interfaces. So Red Black BSTs is very deep. I can add, contain, and delete, and there is nothing informal I need to know, it's all under the hood. Powerful functionality means that all operations are efficient. Tree balancing is maintained using sophisticated yet subtle rules. They are tricky and we hide them under the surface. The most important way to keep modules deep is by practicing information hiding.

Information Hiding

That is, make your variables private, don't let anyone see what's inside the module as much as you can. Embed all the cleverness inside the modules. So that will keep your interfaces simple. And also it would keep it easy to modify your system. If I made a mistake, I can go fix that without thinking about it in another context. The opposite of hiding information is leaking information.

Leaking Information

This occurs when design decisions are reflected across multiple modules.

- Any change to one module requires a change to all modules

- Information leakage is one of the most important red flags in software design
- One of the best skills you can learn as a software designed is a high level of sensitivity to information leakage

Temporal Decomposotion

One of the biggest causes of information leaking is "temporal decomposition," especially in BYOW. The structure of your system very much reflects the order in which events occur.

For example, student often do the following in BYOW:

- Game is started with an input string, so call interactWithInputString()
  - Parse the String and find the seed by extracting N#####S (example code that contains the seed.)[1]
  - Generate the world
  - Process each character using move(World, char)
  - etc.
- Game is started with no input String, so call interactWithKeyboard
  - Display a menu and collect the seed (number we are using to generate the world).[1]
  - Generate the world
  - Until done, call moveWithKeyboard(World)
  - etc.

[1] Because the temporal discussion of when you worked on the project and the temporal decomposition of when these things happen, you don't really recognize that they should be sharing code that collects and extracts the seed for example.

Summary

- Buld classes that provide functionality needed in many places in your code.
- Create deep modules, classes with simple interfaces that do complicated things
- avoid over-reliance on temporal decomposition where your decomposition is driven primarily by the order in which things occur.
  -

It’s OK to use some temporal decomposition, but try to fix any information leakage that occurs!

- Be strategic, not tactical.
- Most importantly: Hide information from yourself when unneeded!

Previous
31.2 Sources of Complexity

Next
31.4 Teamwork

Last updated 1 year ago