Spaghetti code

If somebody tells you that you write good spaghetti code, it's not a compliment. They are telling you that your code has the same "clean logical structure as a plate of spaghetti" (Richard Conway, 1978).

Spaghetti code can be the result when a developer jumps into coding without giving much thought to how the program should flow. The finished product may work as intended, but there could be problems later because the structure and flow are not understood.

As new code is added and older code is copied and pasted into new areas, your previously functioning program evolves into a tangled mess of randomly placed files, folders, and functions. It's almost impossible to add anything new without breaking something. Spaghetti code can be as difficult to figure out as it is to trace each spaghetti noodle on your plate from one end to the other.

It's kind of like adding a new room to an existing building. You wouldn't start building with random materials without considering the architecture, materials, color schemes, design, and flow of the existing building. A random approach would likely result in a room that clashes with the existing building and looks like it just doesn't belong.

Golden hammer

In psychology, a cognitive bias refers to a person's understanding of the world around them based on their own beliefs and experience. The golden hammer is a cognitive bias based on the belief that a single tool can be used to solve all of your programming problems. You've used a specific, well-designed, and architecturally sound piece of code to solve problems in previous projects. Surely it will work again for your current project, right?

Not always.

The idea is that you shouldn't rely too heavily on one solution because one size never really fits all. In our room addition example, a hammer is a very useful tool, but would you try to use it to saw a piece of wood?

You may be able to force your golden hammer anti-pattern into your coding where it doesn't quite fit right—and you might even be able to get it to work—but your program can become unreliable and unstable as you add new features later. And if you're not careful, you could end up with another plate of spaghetti code.

Dead Code

A boat anchor anti-pattern happens when somebody leaves a piece of code in the codebase not because it belongs there but because it might be needed later. The reasoning is that when the code is needed later, it will be easy to turn it on and get it running. It won't be turned on, so what kind of damage can it do?

Like a boat anchor, this type of anti-pattern weighs down your project and can keep it from moving forward quickly. Developers might get caught up in reading through and trying to debug code that won't even be turned on in this iteration. All this extra, unneeded code bloats the codebase and slows down your build times. And if you inadvertently turn on one or more of these boat anchor anti-patterns, it could cause problems such as breaking the build and adding technical risk or debt.

Dead code

Dead code is any section of the source code that might get executed, but its results aren't used by the program. The code is unnecessary and wastes processing resources.

For example, many years ago a technical writer was working on documenting solutions for error codes thrown from networking software. He was surprised to find out that many of programmers didn't know:

- What the error codes meant
- Why the server would throw the error
- Which piece of code triggered those errors

The code was essentially dead and needed to be removed. But the engineers were reluctant to remove it because they were afraid it would introduce new bugs or break the code.

Dead code anti-patterns are heavy, do nothing for the program, slow down development, increase build times, and are difficult to maintain.

God object

When you have an object or class that is doing too much and is responsible for too many things, it might be considered a God object. Assigning too much responsibility goes against the single responsibility principle of object-oriented design. Every object and class in your code should be responsible for a single part of the software's functionality.

For example, a customer ID object that is responsible for the user ID, first name, last name, list of items to purchase, total amount spent, transaction ID, and so on, might be a God object. It makes sense for the customer ID object to take care of the user ID, first name, and last name, but try separating and modularizing the code by creating a separate object to handle the transaction details.

Copy and paste programming

Sometimes copying and pasting code from other sources into your code can cause unintended problems. Just because these code samples worked for other developers on problems similar to yours, doesn't mean they can simply be dropped into your code and work without incident.

If you test the code and ensure that it works, that it can work with your project's architecture, then you might want to go ahead and use it in your code. On the other hand, if you don't test it and add it because it worked for other people, you run the risk of introducing bugs and other problems. The only way to fix it is to hunt down and delete every instance where the code was pasted. Or, you could revert to a version of the software from before you introduced the copied and pasted anti-patterns.