29

No Defect in Behavior or Structure

2. The code that I produce will always be my best work. I will not knowingly allow code that is defective either in behavior or in structure to accumulate.

Remember, Kent Beck once said: “First, make it work. Then, make it right.” Getting the program to work is just the first and easiest step. The second and harder step is to clean the code.

Unfortunately, too many programmers think they are done once they get a program to work. Once it works, they move on to the next program, and then the next, and the next.

In their wake, they leave behind a history of tangled, unreadable code that slows down the whole development team.

They do this because they think their value is in speed. They know they are paid a lot; and so they feel that they must deliver a lot of functionality in a short amount of time.

But software is hard, and it takes a lot of time, and so they feel like they are going too slowly. They feel like they are failing. And this feeling of failure creates a pressure that causes them to try to go faster. It causes them to rush.

And so they rush to get the program working; and then they declare themselves to be done—because, in their minds, it’s already taken them too long.

The constant tapping of the project manager’s foot doesn’t help, but that’s not the real driver.

I teach lots of classes. In many of them, I give the programmers small projects to code. My goal is to give them a coding experience in which they can try out new techniques and new disciplines. I don't care if they actually finish the project.

Indeed, all the code is going to be thrown away.

And still I see people rushing. Some stay past the end of the class just hammering away at getting something utterly meaningless to work.

So, although the boss's pressure doesn't help, the real pressure comes from inside us. We consider speed of development to be a matter of our own self-worth.

Making It Right

As we saw in the last chapter, there are two values to software. There is the value of its behavior and the value of its structure. And in the last chapter, I also made the point that the value of structure is more important than the value of behavior. This is because, to have any long-term value, software systems must be able to react to changes in requirements.

Software that has a structure that makes it hard to change, makes it harder for that software to stay up-to-date with the requirements. Software with a bad structure can quickly get out-of-date.

So, in order to keep up with requirements, the structure of the software has to be clean enough to allow, and even encourage, change. Software that is easy to change can keep up with changing requirements, allowing it to remain valuable with the least amount of effort. But if you've got a software system that's hard to change, then you're going to have a devil of a time keeping that system working when the requirements change.

When are requirements most likely to change? Requirements are most volatile at the start of a project; just after the users have seen the first few features work. That's because they're getting their first look at what the system actually does, as opposed to what they thought it was going to do.

This means that the structure of the system needs to be clean at the very start, if early development is to proceed quickly. If you make a mess at the

start, then even the very first release will be slowed down by that mess.

Good structure enables good behavior and bad structure impedes good behavior. The better the structure, the better the behavior. The worse the structure, the worse the behavior. The value of the behavior depends critically on the structure. And therefore, the value of the structure is the more critical of the two values.

And that means that professional developers put a higher priority on the structure of the code than on the behavior.

Yes, first you make it work; but then, you make very sure that you continue to make it right. You keep the system structure as clean as possible throughout the lifetime of the project. From its very beginning to its very end, it must be clean.

What Is Good Structure?

Good structure makes the system easy to test, easy to change, and easy to reuse. Changes to one part of the code do not break other parts of the code. Changes to one module do not force massive recompiles and redeployments. High-level policies are kept separate and independent from low-level details.

Poor structure makes a system rigid, fragile, and immobile. These are the traditional design smells.

Rigidity is when relatively minor changes to the system cause large portions of the system to be recompiled, rebuilt, and redeployed. A system is rigid when the effort to integrate a change is much greater than the change itself.

Fragility is when minor changes to the behavior of a system force many corresponding changes in a large number of modules. This creates a high risk that a small change in behavior will silently break some other behaviors in the system that then get inadvertently deployed. When this happens, managers and customers will come to believe that the team has lost control over the software and don’t know what they are doing.

Immobility is when a module in an existing system contains a behavior you want to use in a new system but is so tangled up within the existing system that you can't extract it for use in the new system.

These problems are all problems of structure and not behavior. The system may pass all its tests and meet all its functional requirements. And yet such a system may be close to worthless because it is too difficult to manipulate.

There's a certain irony in the fact that so many systems that correctly implement valuable behaviors end up with a structure that is so poor that it negates that value and becomes worthless.

And *worthless* is not too strong a word. In Chapter 1, I told you the tale of the grand redesign in the sky. This is when developers tell management that the only way to make progress is to redesign the whole system from scratch; those developers have assessed the current system to be worthless.

When managers agree to let the developers redesign the system, it simply means that they have agreed with the developers' assessment that the current system is worthless.

What is it that causes these design smells that lead to worthless systems? Source code and data dependencies! And how do we fix those dependencies? Dependency management!

And how do we manage dependencies? We use principles of design, like the SOLID[1] principles, to keep the structure of our systems free of the design smells that would make them worthless.

---

1. These principles are described in [Clean Arch] and [PPP02].

Since the value of structure is greater than the value of behavior, and since the value of structure depends upon good dependency management, and since good dependency management derives from good principles, it follows that the overall value of the system depends upon proper application of those principles.

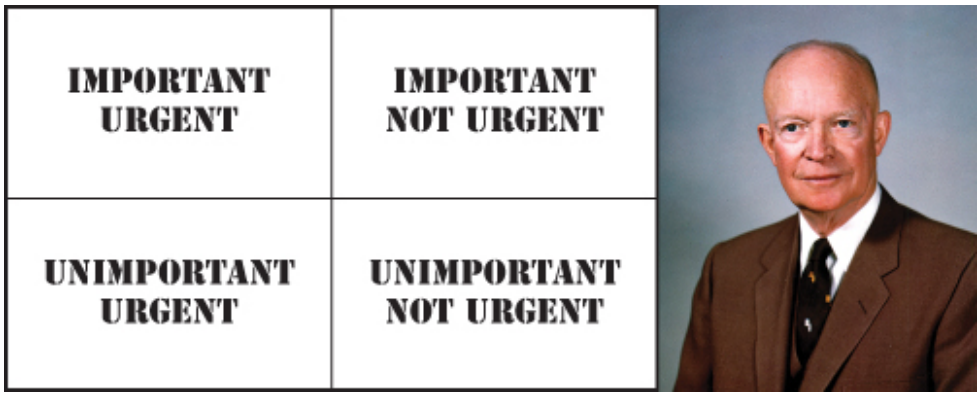
Eisenhower's Matrix

General Dwight D. Eisenhower once said: "I have two kinds of problems, the urgent and the important. The urgent are not important, and the important are never urgent."

There is a deep truth to this statement. A deep truth about engineering. We might even call this "the engineer's motto":

The greater the urgency, the less the relevance.

Here is Eisenhower's decision matrix. Urgency is on the vertical axis, and importance is on the horizontal axis. The four possibilities are urgent and important, urgent and unimportant, important but not urgent, and neither important nor urgent.



Urgency is about time. Importance is not. Things that are important are long term. Things that are urgent are short term. Structure is long term; therefore, it is important. Behavior is short term; therefore, it is merely urgent.

So structure, the important stuff, comes first. Behavior is secondary.

Your boss might not agree with that priority; but that's because it's not your boss's job to worry about structure. It's yours. Your boss simply expects that you will keep the structure clean while you implement the urgent behaviors.

This seems to violate Beck's law: "First, make it work. Then, make it right." Now I'm saying that structure is higher priority than behavior. It's a chicken-and-egg dilemma, isn't it?

The reason we make it work first is because structure has to support behavior; so we implement behavior first, then we give it the right structure. But structure is more important than behavior. We give it a higher priority. We deal with structural issues before we deal with behavioral issues.

We can square this circle by breaking the problem down into tiny units. Let's start with user stories. You get a story to work; and then you get its structure right. And you don't work on the next story until that structure is right. The structure of the current story is higher priority than the behavior of the next story.

Except stories are too big. We need to go smaller. Not stories … tests. Tests are the perfect size.

First you get a test to pass, then you fix the structure of the code that passes that test, before you get the next test to pass.

This entire discussion has been the moral foundation for the Red-Green-Refactor cycle of a good testing discipline like test-driven development (TDD).

It is that cycle that helps us prevent harm to behavior and harm to structure. It is that cycle that allows us to prioritize structure over behavior. And that's why we consider our testing discipline to be a design technique as opposed to a testing technique.

Programmers Are Stakeholders

Remember this. We have a stake in the success of the software. We, programmers, are stakeholders too.

Did you ever think about it that way? Did you ever view yourself as one of the stakeholders of the project?

But, of course, you are. The success of the project has a direct impact upon your career and reputation. So, yes, you are a stakeholder.

And as a stakeholder, you have a say in the way the system is developed and structured. I mean, it's your butt on the line too.

But you are more than just a stakeholder. You are an engineer. You were hired because you know how to build software systems and how to structure those systems so that they last.

And with that knowledge comes the responsibility to produce the best product you can.

So, not only do you have the right as a stakeholder, but you have the duty as an engineer to make sure that the systems you produce do no harm either through bad behavior or through bad structure.

A lot of programmers don't want that kind of responsibility. They'd rather just be told what to do. And that's a travesty and a shame. It's wholly unprofessional. Programmers who feel that way should be paid minimum wage, because that's what their work output is worth.

If you don't take responsibility for the structure of the system, who else will? Your boss?

Does your boss know the SOLID principles? How about design patterns? How about object-oriented design, and the practice of dependency inversion? Does your boss know the discipline of TDD? Do they know what a self-shunt is, or a test-specific subclass, or a humble object? Does your boss understand that things that change together should be grouped together and that things that change for different reasons should be separated?

Does your boss understand structure? Or is your boss's understanding limited to behavior?

Structure matters. If you aren't going to care for it, who will? If not you, then who?

What if your boss specifically tells you to ignore structure and focus entirely on behavior? You refuse. You are a stakeholder. You have rights too. And you are an engineer; you have responsibilities that your boss cannot override.

Perhaps you think that will get you fired. Probably not. Most managers expect to have to fight for things they need and believe in. And they respect

those who are willing to do the same.

Oh, there will be a struggle; even a confrontation. And it won't be comfortable. But you are a stakeholder and an engineer. You can't just back down and acquiesce. That's not professional.

Most programmers do not enjoy confrontation. But dealing with confrontational managers is a skill we have to learn. We have to learn how to fight for what we know is right. Because taking responsibility for the things that matter, and fighting for those things, is how a professional behaves.

Do Your Best

This promise of The Oath is about doing your best.

Clearly, this is a perfectly reasonable promise for a programmer to make. Of course you are going to do your best, and of course you will not knowingly release code that is harmful.

And of course this is not black and white. There are times when structure must bend to schedule. For example, if, in order to make a trade show, you have to put in some quick-and-dirty fix, then so be it.

The promise doesn't even prevent you from shipping code to customers with less-than-perfect structure. If the structure is close but not quite right, and customers are expecting the release tomorrow, then so be it.

On the other hand, the promise does mean that you will address those issues of behavior and structure before you add more behavior. You will not pile more and more behavior on top of known bad structure. You will not allow those defects to accumulate.

What if your boss tells you to anyway? Here's how that conversation should go.

Boss: I want this new feature added by tonight.

Programmer: I'm sorry, but I can't do that; I've got some structural cleanup to do before I can add that new feature.

Boss: Do the cleanup tomorrow. Get the feature done by tonight.

Programmer: That's what I did for the last feature, and now I have an even bigger mess to clean up. I really have to finish that cleanup before I start on anything new.

Boss: I don't think you understand. This is business. Either we have a business or we don't have a business. And if we can't get features done, we don't have a business. Now get the feature done.

Programmer: I understand. Really, I do. And I agree. We have to be able to get features done. But if I don't clean up the structural problems that have accumulated over the last few days, we're going to slow down and get even fewer features done.

Boss: Y'know, I used to like you. I used to say, that Danny, he's pretty nice. But now I don't think so. You're not being nice at all. Maybe you shouldn't be working with me. Maybe I should fire you.

Programmer: Well, sir, that's your right. But I'm pretty sure you want features done quickly and done right. And I'm telling you that if I don't do this cleanup tonight, then we're going to start slowing down. And we'll deliver fewer and fewer features.

Look, sir, I want to go fast, just like you do. You hired me because I know how to do that. You have to let me do my job. You have to let me do what I know is best.

Boss: You really think everything will slow down if you don't do this cleanup tonight?

Programmer: I know it will. I've seen it before. And so have you.

Boss: And it has to be tonight?

Programmer: I don't feel safe letting the mess get any worse.

Boss: You can give me the feature tomorrow?

Programmer: Yes, and it'll be a lot easier to do once the structure is cleaned up.

Boss: OK. Tomorrow. No later. Now get to it.

Programmer: OK. I'll get right on it.

Boss: (aside) I like that kid. He's got guts. He's got gumption. He didn't back down even when I threatened to fire him. He's gonna go far, trust me. But don't tell him I said so.