### *What mistake did the original code make?*

The original code in Bug.cs had a major flaw, it let you change the status of a bug to **anything**, even if it made no sense. Like, you could take a bug that was already marked as **Closed** and bring it back to **Open** like nothing ever happened. That completely breaks the workflow and goes against how real-world bug tracking works. Once something’s closed, that’s supposed to be the end of the story. But the code didn’t enforce that. It also let you set the same status over and over again, which doesn’t do anything and could confuse the system or the user.

Basically, it was too forgiving, and in a system like this, that kind of flexibility causes chaos.

### How did your tests help you uncover the issue?

Writing tests is like shining a flashlight into a dark room. At first glance, the code *looked* fine, no errors, everything ran. But once I started writing unit tests and simulating real status changes, it exposed the cracks fast.

I wrote tests for:

* Valid transitions like **Open → InProgress** and **InProgress → Closed**
* Invalid transitions like **Closed → Open** or setting the same status again

The moment those tests hit the bad transitions, they either passed when they shouldn’t, or they didn’t throw any errors at all. That’s how I knew the logic was broken, the tests made it undeniable.

From there, I refactored UpdateStatus() to strictly follow the proper lifecycle, and had it throw InvalidOperationException when someone tried to go outside the rules. Then I re-ran the tests, and boom, green lights all across the board.

That’s the power of Test-Driven Development. It doesn’t just help you write better code, iit **forces you to think like a user and an engineer at the same time.**

I did step outside the original scope of the assignment. But not just to do extra for the sake of it. I did it because the assignment, while clear, left some real-world gaps that I knew would come back to bite me if I didn’t handle them right.

For example, the instructions had me focus on the Bug.cs file, which is where the core logic lives. But in practice, that logic gets called through the BugService, and *that’s* where a lot of the interaction actually happens. So instead of just patching things in one spot, I chose to reinforce the entire workflow, testing the full path, from service to model, because in the real world, users don’t talk to models, they talk to services. I wanted to make sure the logic held up all the way through.

### *Challenges I Ran Into*

One challenge was dealing with how the original property Status had a private setter. That meant I couldn’t just assign values in my tests like I normally would, I had to simulate valid transitions step by step using the UpdateStatus() method. That forced me to think carefully about the lifecycle of a bug and how to chain transitions without breaking the rules.

Another challenge was when I added exception handling for invalid transitions. My tests started failing, but not because the code was wrong. It was because the tests **weren’t ready for the exceptions I was now throwing on purpose**. So I had to refactor my tests to expect and assert those exceptions. That was actually a really cool learning moment, because it showed how writing strong tests doesn’t just check your logic, it makes your logic stronger.

### *Bottom Line*

I stepped outside the box to do this the right way, not just the easy way. I treated it like production-level code, not just homework. And yeah, I hit some bumps, but I’m glad I did. That’s where the real growth happened.

### *🧨 Examples of Intentional Exceptions Thrown*

#### **🔁 InvalidOperationException — Same Status Reassignment**

if (Status == newStatus)

throw new InvalidOperationException("Bug is already in the requested status.");

✅ **Why I threw it**:  
 If a bug is already marked as Open, there's no reason someone should be able to mark it Open again. That kind of redundancy can clutter logs, mislead users, or trigger unintended side effects. I wanted to shut that down early.

#### **❌ InvalidOperationException — Trying to Change a Closed Bug**

if (Status == BugStatus.Closed)

throw new InvalidOperationException("Cannot change status once it is Closed.");

✅ **Why I threw it**:  
 Closed is supposed to be a final state. Once a bug is closed, it’s done — archived, resolved, whatever the process is. Reopening it through a status change like Closed → Pending or Closed → Open breaks the logic. If a bug needs to be reopened, that should go through a separate approval flow — not a silent status change.

#### **⛔ InvalidOperationException — Invalid Lifecycle Transitions**

if (Status == BugStatus.Open && newStatus != BugStatus.InProgress)

throw new InvalidOperationException("Open bugs can only move to InProgress.");

if (Status == BugStatus.InProgress && newStatus != BugStatus.Pending && newStatus != BugStatus.Closed)

throw new InvalidOperationException("InProgress bugs can only move to Pending or Closed.");

if (Status == BugStatus.Pending && newStatus != BugStatus.InProgress && newStatus != BugStatus.Closed)

throw new InvalidOperationException("Pending bugs can only move to InProgress or Closed.");

**Why I threw them**:  
 These guard rails exist to make sure bugs follow a **logical lifecycle**. I didn’t want people to send a bug from Open straight to Closed or from Pending back to Open. This code enforces the intended workflow so the whole team can trust the system.