Extracted from:

Practices of an Agile Developer

Working in the Real World

You might get the impression that experienced woodworkers never make mistakes. I can assure you that isn't true. Pros simply know how to salvage their goofs.

► Jeff Miller, furniture maker and author

Chapter 7

Agile Debugging

Even on the most talented agile projects, things will go wrong. Bugs, errors, defects, mistakes—whatever you want to call them, they will happen.

The real problem with debugging is that it is not amenable to a time box. You can time box a design meeting and decide to go with the best idea at the end of some fixed time. But with a debugging session, an hour, a day, or a week may come and go and find you no closer to finding and fixing the problem.

You really can't afford that sort of open-ended exposure on a project. So, we have some techniques that might help, from keeping track of previous solutions to providing more helpful clues in the event of a problem.

To reuse your knowledge and effort better, it can help to *Keep a Solutions Log*, and we'll see how on the following page. When the compiler warns you that something is amiss, you need to assume that *Warnings Are Really Errors* and address them right away (that's on page 132).

It can be very hard—even impossible—to track down problems in the middle of a entire system. You have a much better chance at finding the problem when you *Attack Problems in Isolation*, as we'll see on page 136. When something does go wrong, don't hide the truth. Unlike some government cover-up, you'll want to *Report All Exceptions*, as described on page 139. Finally, when you do report that something has gone awry, you have to be considerate of users, and *Provide Useful Error Messages*. We'll see why on page 141.

33 Keep a Solutions Log

"Do you often get that déjà vu feeling during development? Do you often get that déjà vu feeling during development? That's OK. You figured it out once. You can figure it out again."

Facing problems (and solving them) is a way of life for developers. When a problem arises, you want to solve it quickly. If a similar problem occurs again, you want to remember what you did the first time and fix it more quickly the next time. Unfortunately, sometimes you'll see a problem that looks the same as something you've seen before but can't remember the fix. This happens to us all the time.

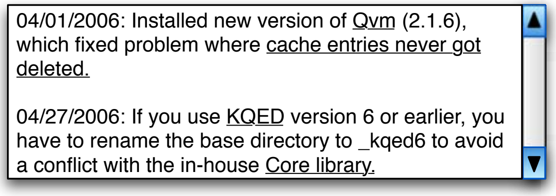
Can't you just search the Web for an answer? After all, the Internet has grown to be this incredible resource, and you might as well put that to good use. Certainly searching the Web for an answer is better than wasting time in isolated efforts. However, it can be *very* time-consuming. Sometimes you find the answers you're looking for; other times, you end up reading a lot of opinions and ideas instead of real solutions. It might be comforting to see how many other developers have had the same problem, but what you need is a solution.

Don't get burned twice

To be more productive than that, maintain a log of problems faced and solutions found. When a problem appears, instead of saying, "Man, I've seen this before, but I have no clue how I fixed it," you can quickly look up the solution you've used in the past. Engineers have done this for years: they call them *daylogs*.

You can choose any format that suits your needs. Here are some items that you might want to include in your entries:

- Date of the problem
- Short description of the problem or issue
- Detailed description of the solution
- References to articles, and URLs, that have more details or related information
- Any code segments, settings, and snapshots of dialogs that may be part of the solution or help you further understand the details



04/01/2006: Installed new version of Qvm (2.1.6), which fixed problem where cache entries never got deleted.

04/27/2006: If you use KQED version 6 or earlier, you have to rename the base directory to _kqed6 to avoid a conflict with the in-house Core library.

Figure 7.1: Example of a solutions log entry, with hyperlinks

Keep the log in a computer-searchable format. That way you can perform a keyword search to look up the details quickly. Figure 7.1 shows a simple example, with hyperlinks to more information.

When you face a problem and you can't find the solution in your log, remember to update your log with the new details as soon as you do figure out a solution.

Even better than maintaining a log is sharing it with others. Make it part of your shared network drive so others can use it. Or create a Wiki, and encourage other developers to use it and update it.

Maintain a log of problems and their solutions. *Part of fixing a problem is retaining details of the solution so you can find and apply it later.*

What It Feels Like

Your solutions log feels like part of your brain. You can find details on particular issues and also get guidance on similar but different issues.

Keeping Your Balance

- You still need to spend more time solving problems than documenting them. Keep it light and simple; it doesn't have to be publication quality.

- Finding previous solutions is critical; use plenty of keywords that will help you find an entry when needed.
- If a web search doesn't find *anyone* else with the same problem, perhaps you're using something incorrectly.
- Keep track of the specific version of the application, framework or platform where the problem occurred. The same problem can manifest itself differently on different platforms/versions.
- Record *why* the team made an important decision. That's the sort of detail that's hard to remember six to nine months later, when the decision needs to be revisited and recriminations fill the air.

34 Warnings Are Really Errors

“Compiler warnings are just for the overly cautious and pedantic. They’re just warnings after all. If they were serious, they’d be errors, and you couldn’t compile. So just ignore them, and let ’er rip.”

When your program has a compilation error, the compiler or build tool refuses to produce an executable. You don’t have a choice—you have to fix the error before moving on.

Warnings, unfortunately, are not like that. You can run the program that generates compiler warnings if you want. What happens if you ignore warnings and continue to develop your code? You’re sitting on a ticking time bomb, one that will probably go off at the worst possible moment.

Some warnings are benign by-products of a fussy compiler (or interpreter), but others are not. For instance, a warning about a variable not being used in the code is probably benign but may also allude to the use of some other incorrect variable.

At a recent client site, Venkat found more than 300 warnings in an application in production. One of the warnings that was being ignored by the developers said this:

```
Assignment in conditional expression is always constant;
did you mean to use == instead of = ?
```

The offending code was something like this:

```
if (theTextBox.Visible = true)
...
```

In other words, that **if** will always evaluate as true, regardless of the hapless theTextBox variable. It’s scary to see genuine errors such as this slip through as warnings and be ignored.

Consider the following C# code:

```
public class Base
{
  public virtual void foo()
  {
    Console.WriteLine("Base.foo");
  }
}
```

```
public class Derived : Base
{
  public virtual void foo()
  {
    Console.WriteLine("Derived.foo");
  }
}

class Test
{
  static void Main(string[] args)
  {
    Derived d = new Derived();
    Base b = d;
    d.foo();
    b.foo();
  }
}
```

When you compile this code using the default Visual Studio 2003 project settings, you'll see the message "Build: 1 succeeded, 0 failed, 0 skipped" at the bottom of the Output window. When you run the program, you'll get this output:

```
Derived.foo
Base.foo
```

But this isn't what you'd expect. You should see both the calls to foo() end up in the Derived class. What went wrong? If you examine the Output window closely, you'll find a warning message:

```
Warning. Derived.foo hides inherited member Base.foo
To make the current member override that implementation,
add the override keyword. Otherwise, you'd add the new keyword.
```

This was clearly an *error*—the code should use **override** instead of **virtual** in the Derived class's foo() method.[1] Imagine systematically ignoring warnings like this in your code. The behavior of your code becomes unpredictable, and its quality plummets.

You might argue that good unit tests will find these problems. Yes, they will help (and you should certainly use good unit tests). But if the compiler can detect this kind of problem, why not let it? It'll save you both some time and some headaches.

[1]And this is an insidious trap for former C++ programmers; the program would work as expected in C++.

Find a way to tell your compiler to treat warnings as errors. If your compiler allows you to fine-tune warning reporting levels, turn that knob all the way up so no warnings are ignored. GCC compilers support the -Werror flag, for example, and in Visual Studio, you can change the project settings to treat warnings as errors.

That is the least you should do on a project. Unfortunately, if you go that route, you will have to do it on each project you create. It'd be nice to enable that more or less globally.

In Visual Studio, for instance, you can modify the project templates (see *.NET Gotchas* [Sub05] for details) so any project you create on your machine will have the option set, and in the current version of Eclipse, you can change these settings under Window → Preferences → Java → Compiler → Errors/Warnings. If you're using other languages or IDEs, take time to find how you can treat warnings as errors in them.

While you're modifying settings, set those same flags in the continuous integration tool that you use on your build machine. (For details on continuous integration, see Practice 21, *Different Makes a Difference*, on page 87.) This small change can have a huge impact on the quality of the code that your team is checking into the source control system.

You want to get all of this set up right as you start the project; suddenly turning warnings on partway through a project may be too overwhelming to handle.

Just because your compiler treats warnings lightly doesn't mean you should.

Treat warnings as errors. *Checking in code with warnings is just as bad as checking in code with errors or code that fails its tests. No checked-in code should produce any warnings from the build tools.*

What It Feels Like

Warnings feel like...well, warnings. They are warning you about something, and that gets your attention.

Keeping Your Balance

- Although we've been talking about compiled languages here, interpreted languages usually have a flag that enables run-time warnings. Use that flag, and capture the output so you can identify—end eliminate—the warnings.
- Some warnings can't be stopped because of compiler bugs or problems with third-party tools or code. If it can't be helped, don't waste further time on it. But this shouldn't happen very often.
- You can usually instruct the compiler to specifically suppress unavoidable warnings so you don't have to wade through them to find genuine warnings and errors.
- Deprecated methods have been deprecated for a reason. Stop using them. At a minimum, schedule an iteration where they (and their attendant warning messages) can be removed.
- If you mark methods you've written as deprecated, document what current users should do instead and when the deprecated methods will be removed altogether.

35 Attack Problems in Isolation

"Stepping line by line through a massive code base is pretty scary. But the only way to debug a significant problem is to look at the entire system. All at once. After all, you don't know where the problem may be, and that's the only way to find it."

One of the positive side effects of unit testing (Chapter 5, *Agile Feedback*, on page 76) is that it forces you to layer your code. To make your code testable, you have to untangle it from its surroundings. If your code depends on other modules, you'll use mock objects to isolate it from those other modules. In addition to making your code robust, it makes it easier to locate problems as they arise.

Otherwise, you may have problems figuring out where to even start. You might start by using a debugger, stepping through the code and trying to isolate the problem. You may have to go through a few forms or dialogs before you can get to the interesting part, and that makes it hard to reach the problem area. You may find yourself struggling with the entire system at this point, and that just increases stress and reduces productivity.

Large systems are complicated—many factors are involved in the way they execute. While working with the entire system, it's hard to separate the details that have an effect on your particular problem from the ones that don't.

The answer is clear: don't try to work with the whole system at once. Separate the component or module you're having problems with from the rest of the code base for serious debugging. If you have unit tests, you're there already. Otherwise, you'll have to get creative.

For instance, in the middle of a time-critical project (aren't they all?), Fred and George found themselves facing a major data corruption problem. It took a lot of work to find what was wrong, because their team didn't separate the database-related code from the rest of the application. They had no way to report the problem to the vendor—they certainly couldn't email the entire source code base to them!

So, they developed a small prototype that exhibited similar symptoms. They sent this to the vendor as an example and asked for their expert opinion. Working with the prototype helped them understand the issues more clearly.

Plus, if they *weren't* able to reproduce the problem in the prototype, it would have shown them examples of code that actually worked and would have helped them isolate the problem.

Prototype to isolate

The first step in identifying complex problems is to isolate them. You wouldn't try to fix an airplane engine in midair, so why would you diagnose a hard problem in a part or component of your application while it's working inside the entire application? It's easier to fix engines when they're out of the aircraft and on the workbench. Similarly, it's easier to fix problems in code if you can isolate the module causing the problem.

But many applications are written in a way that makes isolation difficult. Application components or parts may be intertwined with each other; try to extract one, and all the rest come along too.[2] In these cases, you may be better off spending some time ripping out the code that is of concern and creating a test bed on which to work.

Attacking a problem in isolation has a number of advantages: by isolating the problem from the rest of the application, you are able to focus directly on just the issues that are relevant to the problem. You can change as much as you need to get to the bottom of the problem—you aren't dealing with the live application. You get to the problem quicker because you're working with the minimal amount of relevant code.

Isolating problems is not just something you do after the application ships. Isolation can help us when prototyping, debugging, and testing.

Attack problems in isolation. *Separate a problem area from its surroundings when working on it, especially in a large application.*

What It Feels Like

When faced with a problem that you have to isolate, it feels like searching for a needle in a tea cup, not a needle in a haystack.

[2]This is affectionately known as the "Big Ball of Mud" design antipattern.

Keeping Your Balance

- If you separate code from its environment and the problem goes away, you've helped to isolate the problem.
- On the other hand, if you separate code from its environment and the problem *doesn't* go away, you've still helped to isolate the problem.
- It can be useful to *binary chop* through a problem. That is, divide the problem space in half, and see which half contains the problem. Then divide that half in half again, and repeat.
- Before attacking your problem, consult your log (see Practice 33, *Keep a Solutions Log*, on page 129).

36

Report All Exceptions

"Protect your caller from weird exceptions. It's your job to handle it. Wrap everything you call, and send your own exception up instead—or just swallow it."

Part of any programming job is to think through how things should work. But it's much more profitable to think about what happens when things *don't* work—when things don't go as planned.

Perhaps you're calling some code that might throw an exception; in your own code you can try to handle and recover from that failure. It's great if you can recover and continue with the processing without your user being aware of any problem. If you can't recover, it's great to let the user of your code know exactly what went wrong.

But that doesn't always happen. Venkat found himself quite frustrated with a popular open-source library (which will remain unnamed here). When he invoked a method that was supposed to create an object, he received a null reference instead. The code was small, isolated, and simple enough, so not a whole lot could've been messed up at the code level. Still, he had no clue what went wrong.

Fortunately it was open source, so he downloaded the source code and examined the method in question. It in turn called another method, and that method determined that some necessary components were missing on his system. This low-level method threw an exception containing information to that effect. Unfortunately, the top-level method quietly suppressed that exception with an empty **catch** block and returned a null instead. The code Venkat had written had no way of knowing what had happened; only by reading the library code could he understand the problem and finally get the missing component installed.

Checked exceptions, such as those in Java, force you to catch or propagate exceptions. Unfortunately, some developers, maybe temporarily, catch and ignore exceptions just to keep the compiler from complaining. This is dangerous—temporary fixes are often forgotten and end up in production code. You must handle all exceptions and recover from the failures if you can. If you can't handle it yourself, propagate it to your method's caller so it can take a stab at handling it (or gracefully communicate the information about the problem to users; see Practice 37, *Provide Useful Error Messages*, on the next page).

Sounds pretty obvious, doesn't it? Well, maybe it's not as obvious as you think. A story in the news not long ago talked about a major failure of a large airline reservations system. The system crashed, grounding airplanes, stranding thousands of passengers, and snarling the entire air transportation system for days. The cause? *A single unchecked SQL exception in an application server.*

Maybe you'd enjoy the fame of being mentioned on CNN, but probably not like that.

Handle or propagate all exceptions. *Don't suppress them, even temporarily. Write your code with the expectation that things will fail.*

What It Feels Like

You feel you can rely on getting an exception when something bad happens. There are no empty exception handlers.

Keeping Your Balance

- Determining who is responsible for handling an exception is part of design.
- Not all situations are exceptional.
- Report an exception that has meaning in the context of this code. A NullPointerException is pretty but just as useless as the **null** object described earlier.
- If the code writes a running debug log, issue a log message when an exception is caught or thrown; this will make tracking them down much easier.
- Checked exceptions can be onerous to work with. No one wants to call a method that throws thirty-one different checked exceptions. That's a design error: fix it, don't patch over it.
- Propagate what you can't handle.