Dirty Tricks in the Name of Quality

Ian Dees
ian.s.dees@tektronix.com

Abstract

We join software projects with grand ideas of tools, techniques, and processes we'd like to try. But we don't write code in a vacuum. Except on the rare occasions when we're starting from scratch, we're confronted with legacy code we may not understand and team members who have been quite productive for years without the silver bullets we're pushing.

How do we get a toehold on a mountain of untested code? How can we get our software to succeed despite itself? Sometimes, we have to get our hands dirty. We may have to break code to fix it again. We may have to put ungainly scaffolding in place to hold the structure together long enough to finish construction. We may have to look to seemingly unrelated languages and communities for inspiration.

This introductory-level talk is a discussion of counterintuitive actions that can help improve software quality. We'll begin with source code, zoom out to project organization, and finally consider our personal roles as contributors to quality.

Biography

Ian Dees was first bitten by the programming bug in 1986 on a Timex Sinclair 1000, and has been having a blast in his software apprenticeship ever since.

Since escaping Rice University in 1996 with engineering and German degrees, he has debugged assembler with an oscilloscope, written web applications nestled comfortably in high-level frameworks, and seen everything in between. He currently hacks embedded C++ application code, automates laboratory hardware, and writes test scripts for Tektronix, a test equipment manufacturer near Portland, Oregon.

When he's not coding for work or for friends, you're most likely to find Ian chasing his family around on bicycles, plinking away at his guitar, or puzzling at the knobs on the espresso machine while some impromptu meal simmers on the stove nearby.

Ian is the author of Scripted GUI Testing With Ruby *and co-author of* Using JRuby*, both published by the Pragmatic Programmers.*

1. Introduction

We just finished a big project at work. At times like these, it's natural to reflect on the practices that got us to this point, to see which ones are effective and which ones we should abandon. As we engaged in this sorting activity, a third category emerged: practices that helped in a surprising way. For want of a better term, I'm going to refer to these as *dirty tricks*.

What's a dirty trick? It's certainly not a universal best practice.[1] It's also not an antipattern, though. Antipatterns are near-universal worst practices; for some examples, see Moss Drake's "Sabotaging Quality" paper (which he's presenting here at PNSQC 2011).

No, dirty tricks sit somewhere between those two extremes. They're practices that may help in one specific context, and be a terrible idea in other contexts. They may only be useful as a desperate last resort, when the alternative is doing nothing and watching your software—or your career—slowly stagnate.

So, why am I bringing up these practices today, if they're so context-specific or such potentially bad ideas? There are a few reasons to talk about dirty tricks:

1. Because dirty tricks may only be valid in a narrow context, they're a good reminder to consider the context of *any* proposed tool, technology, methodology, or practice. The next time you hear someone tout a so-called "best practice," you can remember this talk and engage your skepticism gear.

2. You may actually find one or two of these techniques useful as a kind of digital dynamite if you're on a project whose progress has become dammed up with untamed code.

3. I hope to elicit a few groans of recognition and keep you entertained for the next hour or so.

This is primarily a talk on software construction. So we're going to start off with some code examples of dirty tricks. Since many of us program in C++, that's the language I'll use for the examples. From there, we'll zoom out one level, and look at ways to shoehorn testing into a reluctant project or subsystem. Finally, we'll talk about more touchy-feely topics like survival in the workplace.

In short, this is a talk on *mindfully* taking on a little technical debt, with our eyes fully open to the costs and consequences.[2]

2. Assorted Dirty Tricks

2.1. The Meta-Dirty Trick

The first and most important tip we're going to discuss, and something I actually consider to be a good practice in most contexts, is:

Don't get fired.

As I've said, the activities we'll be discussing are not best practices. They're not even recommendations. They may be a terrible idea in your context or on your project. Don't undertake them heedlessly. For that reason, this is the only tip you'll see stated in the imperative. The rest of the dirty tricks are all nouns or noun phrases, just to drive the point home.

---

[1] As if there were such a thing.

[2] For more on technical debt, see http://www.martinfowler.com/bliki/TechnicalDebt.htm.

2.2. Blunt Code

2.2.1. The Code Crowbar

The code crowbar is a blunt instrument for cracking open a project and getting a little testing in. Like a regular crowbar, this digital version is meant for one-time use—eventually, we'll discard it and install a proper entry point for our tests.

This dirty trick comes in handy with legacy code that has no unit tests. The proper way to add testing may be to change the design dramatically. But that's risky and takes time. With the crowbar, one can add a tiny bit of testing—just one or two methods in a single class, for starters—and then use those tests to drive the improved architecture.

For example, consider the following C++ class:

```
class Widget
{
public:
    // ...

private:
    int vim;
    int vigor;
    int Pizazz() { return vim * vigor; }
};
```

Let's imagine the `Pizazz()` calculation is a little less trivial and more error-prone than the simple multiplication shown here. We'd like add some tests for it, like this:

```
class WidgetTest
{
public:
    void TestPizazz()
    {
        Widget w;
        w.vim   = 2;
        w.vigor = 5;
        assert(w.Pizazz() == 10);
    }
};
```

All of the `Widget` fields and methods we're using are private, so this code won't compile. The real issue is that we're dealing with a bunch of separate things that really belong together—they're all part of the class's configuration. A permanent improvement will probably involve moving those private properties into a completely new configuration class. If this is a much larger class in a more complicated system, that task might require lots of changes to the code. Breaking up a big module with no tests is risky business.

The crudest way possible to get the above test code to compile and run is to add the following definition just before we `#include` the definition of the `Widget` class:

```
#define private public
#include "widget.h"
```

Horrible, isn't it? We shouldn't do things this drastic lightly. We can do better than this. We could *(temporarily!)* give our test code access to those internals by adding the following line to the `Widget` class:

```
    friend class WidgetTest;
```

This kind of intrusive monkeying with a class's internals does not make for long-term quality. But it gives us a toehold, so that we can move this configuration information to its own class:

```
class WidgetConfig
{
public:
    WidgetConfig(int vim, int vigor)
        : vim(vim), vigor(vigor) {}

    int Pizazz() { return vim * vigor; }

private:
    int vim;
    int vigor;
};
```

…and replace those private fields in the `Widget` class:

```
class Widget
{
public:
    // ...

private:
    WidgetConfig config;
};
```

Now, our test code doesn't need to resort to dirty tricks anymore:

```
class WidgetConfigTest
{
public:
    void TestPizazz()
    {
        WidgetConfig wc(2, 5);
        assert(wc.Pizazz() == 10);
    }
};
```

This example illustrates one property common to several dirty tricks: what we did was temporary. We started with getting the tests in by any means necessary, we did the hard work to improve our code quality, and then we cleaned up our mess.

2.2.2. Macro Abuse

Developers have an allergic reaction to C++ preprocessor macros, and with good reason. Search stackoverflow.com for the term "macro abuse" for some rather entertaining examples of macros making spaghetti code even worse. Many uses of macros can be replaced with more type-safe constructions like new-style casts, templates, and so on.

However, the preprocessor is there for a reason. It's there for you to write "code that writes code," albeit in a somewhat crude fashion. Imagine the following hypothetical code based loosely on a real-life example:

```
someComplicatedFunction(LONG_NAME_FOR_FOO, WITH_FOO_AND_BAR, BAR_VALUE);
someComplicatedFunction(LONG_NAME_FOR_QUUX, WITH_QUUX_AND_BAZ, BAR_VALUE);
```

Did you spot the probable error? If we look closely at the names of the macros, we can see that the first line deals with the names `FOO` and `BAR`. The second line looks like it was meant to deal with `QUUX` and `BAZ`. But the end of the line refers to `BAR_VALUE`, which is likely a typo or a cut-and-paste error. It was supposed to be `BAZ_VALUE`.

If we define a couple of helper macros:

```
#define DO_SOMETHING_WITH(k, v) {\
    someComplicatedFunction(LONG_NAME_FOR_ ## k,      \
                            WITH_ ## k ## _AND_ ## v, \
                            v ## _VALUE);             \
}
```

…we can avoid that particular cut-and-paste error entirely:

```
DO_SOMETHING_WITH(FOO,  BAR);
DO_SOMETHING_WITH(QUUX, BAZ);
```

You're all seasoned pros who know how the preprocessor works. And yet, I bring this example up because of all the horror stories that make us feel guilty for using macros at all. It's okay to use blunt instruments sometimes.

With this foundation of guilt-free coding established, we can build on it with a couple of related tips that lean on the preprocessor.

2.2.3. Testing? Testing!

Ideally, the code that runs in our unit tests should be the same as the code that runs in the production system. Sometimes, though, things get more tangled. The code we're testing might link to a function in a completely different subsystem that has a bunch more dependencies. Those dependencies will presumably have their own tests.

If we're in the process of writing a unit test, we don't want to be forced to drag a bunch of other subsystems in. For example, imagine we encounter the following call in the middle of the `Aim()` method of a `MindControlLaser` object:

```
if (systemBatteryLevel() < LOW_BATTERY_THRESHOLD)
{
    // logging code
}
```

The first unit test we write for the `Aim()` function is presumably concerned with making sure the aiming algorithm is correctly implemented. The logging and battery systems aren't part of this class's contract. The right thing to do in the long term is decouple the `MindControlLaser` from any specific battery implementation. There are lots of ways to do this, as we'll see later on in "The Stubmarine."

Those big refactorings will have to wait until we've got some tests in, though. Let's temporarily excise the problematic code from our unit tests:

```
#ifndef TESTING
//... system battery level stuff from above
#endif
```

We'll set the `TESTING` flag when we compile our unit tests, but leave it unset when we build the production app. Later, we can refactor the code and remove this temporary scaffolding.

2.2.4. Static Assertions

Another useful application of the preprocessor is *static assertions*. These are actually a broadly useful technique, but I'm still bringing them up here as a dirty trick because nearly all plain C implementations of the idea rely on some form of source code cuteness.

Static assertions document assumptions about the program that the compiler can verify. For instance, suppose we're working on an application to route salespeople to cities. Most parts of the system—the user interface, the storage engine, and so on—don't care what the maximum number of salespeople is that we can support. So we've got a constant defined in our program somewhere like this:

```
#define MAX_SALESPEOPLE 100
```

But let's say we have a super-fast routing algorithm that's only proven correct with a sales staff of fewer than a thousand people. We'd like to be able to document this property in the code:

```
STATIC_ASSERT(MAX_SALESPEOPLE < 1000)
```

If a well-meaning coder (perhaps even us!) comes along later and changes `MAX_SALESPEOPLE` to 2000, the compiler will complain and they'll be forced to confront the issue: either replace the algorithm with a slower one that works on larger data sets, or decide that the limit is something we can live with.

Jon Jagger has written a nice catalog of compiler tricks used to implement static assertions.[3] The most reliable of these relies on a case statement:

```
#define STATIC_ASSERT(condition) \
    switch(condition) { case 0: case condition: break; }
```

As long as the condition is something the compiler can evaluate to true or false, this code will test your assumptions at compile time.[4]

2.2.5. The Wheel, Reinvented

"No code is better than no code"—Ezra's Law[5]

It's always better to use an off-the-shelf library than to write your own, isn't it? I'd rather reach for a library or something out of a common tools folder to do just about any common task—decoding JPEGs, etc.—than spend hours writing my own buggy version. Those hours should be spent delivering features to paying customers.

And yet… there are times when it's okay to reinvent the wheel. If you have quality or legal concerns about a piece of third-party code, and if the task you're undertaking is extremely well-defined, then the risk of rolling your own solution is low.

---

[3] http://www.jaggersoft.com/pubs/CVu11_3.html

[4] The upcoming revision of the C++ standard has an official mechanism for these kinds of assertions.

[5] http://www.longbeard.org/2008/07/13/Ezras-Law.html

Test frameworks are a good example of things you might want to write yourself. They're pretty easy to do. They're not impossible to get right. Heck, they're practically a rite of passage in some programming languages.[6] If you're concerned about getting Legal to approve an open source license, or just don't want to put up with someone else's idea of what goes in `main()`, give it a shot.

How easy is this? A test "framework" can be as simple as an agreement to use the built-in C `assert()` macro and name your test methods `Test...()`. For a more useful example of a minimalist framework, see John Brewer's MinUnit exercise.[7]

2.3. Test Practices

2.3.1. The Stubmarine

A moment ago, we looked at an example that, in just a few lines, entangled the otherwise simple `Aim()` function of a `MindControlLaser` class with a specific battery system:

```
if (systemBatteryLevel() < LOW_BATTERY_THRESHOLD)
{
    // logging code
}
```

For the moment, we slapped an `#ifndef TESTING` around those lines. But now, let's dig in and see what this code tells us about our design. Why does the `Aim()` function need to check the battery level? Is it just something someone left in for debugging purposes? Is it to provide a cue for the operator? Is it part of the contract of this class that it checks the battery before aiming?

Once we've answered those questions, we'll know more about what our tests need to do. If the battery check was just something we had in there for debugging, an cheap way to make this function more testable is to turn the battery level function into a pointer:

```
if (batteryLevelFunc &&
    (*batteryLevelFunc)() < LOW_BATTERY_THRESHOLD)
{
    // logging code
}
```

The testing code can leave this function set to `NULL`, while the development version of the app can point it at the real battery function—or even selectively enable/disable the check at runtime.

If it's actually part of the specification for `MindControlLaser` that the `Aim()` function does something specific with a low battery level, we'd like to be able to test that. To do so, we can provide a stub function that returns canned values:

---

[6] Such as Ruby. And SQL, apparently; see http://chrisoldwood.blogspot.com/2011/04/you-write-your-sql-unit-tests-in-sql.html

[7] http://www.jera.com/techinfo/jtns/jtn002.html

```
double fakeLowBatteryLevel()
{
    return LOW_BATTERY_THRESHOLD / 2.0;
}

void testAimWithLowBattery()
{
    MindControlLaser l;
    l.batteryLevelFunc = &fakeLowBatteryLevel;
    l.Aim();
    assert(l.WarningLightOn());
}
```

Of course, if the thing you're replacing is a C++ class, you've got even more options—such as having your real battery and fake battery implement the same interface.

For more sophisticated behavior (e.g., recording whether or not a function gets called), you may look into a more full-featured mocking library like Google Mock.[8]

2.3.2. Cut and Paste

An engineer confronted with a crooked picture will "buy a CAD system and spend the next six months designing a solar-powered, self-adjusting picture frame while often stating aloud [the] belief that the inventor of the nail was a total moron."[9]

Sometimes the right thing to do when faced with a problem is just fix it quickly. Earlier in "Macro Abuse," we saw how cut and paste led to a subtle error, and how doing tricks with the programming language helped us avoid the problem. Now just for fun, we're going to talk about the opposite side of the coin.

There are a few times when a simple cut and paste is easier, faster, and even more reliable than whipping up a Rube Goldberg contraption to write our code for us. One of these cases you'll frequently encounter in the C++ world is makefiles.

When we first introduce testing into a project, we may find ourselves needing two radically different groups of build settings: one for the real program, and one for the tests. These settings may include compiler flags, linker settings, and perhaps even different compilers outright.

If the differences are big enough, the crude "everything's a global constant" approach of makefiles may require you to write a bunch of spaghetti in your build rules. The simplest solution here may be two makefiles. It's either that, or try to use the weak abstraction mechanisms offered by most build systems. The cure may be worse than the disease.

2.3.3. Wranings and Erors *[sic]*

This next item is barely even a dirty trick at all. It's helpful in more situations than our other dirty tricks, and its use isn't even frowned on all that often. The practice: telling the compiler to complain more loudly.

This technique takes several different forms, depending on the programming language. C++ has the `-Werror` compiler flag to treat all warnings as errors you must fix before continuing. Perl has `Test::NoWarnings` and `use warnings`, among other things.

---

[8] http://code.google.com/p/googlemock

[9] http://www.netfunny.com/rhf/jokes/96/Oct/engineers.html

Why is this important? In a big build, it's easy for a compiler warning to slip by unnoticed in the log (or be buried in your IDE). More often than not, the warnings I've seen have indicated real bugs in the code. There have been a few false positives over the years, but these all had easy workarounds.

I'm a big fan of this approach, but still consider it a slightly dirty trick because it affects your teammates as well. If someone's legitimate programming technique happens to be caught by the compiler, they're going to question the value of all this extra checking—and justifiably so. Fortunately, time heals most objections, as more developers see the stricter settings catch a bug that otherwise would have made it into production.

2.3.4. Manualization

Manualization means making something manual. I'm using it here to mean the opposite of automation. More specifically, I'm talking about setting up something manual that has some of the same desirable properties as an automated process.

Consider the practice of continuous integration, where a build server is constantly pulling the latest updates from source control and rebuilding the project. These often run an automated test suite as well.

A large, complex product may have a suite of automated tests that takes hours to run. With source code checkins happening every minute, such a long turnaround time cancels a few of the benefits of continuous integration. We may throw more hardware at the problem, or run the tests less often, or run only a certain subset of the tests on the continuous integration server. But there's another possibility as well. We can make integration and testing a side effect of the normal build.

If you have a large enough team of developers, you can perform something akin to manual continuous integration. Several times throughout the day, people can pull the latest code from one another and build it on their own. They can test by hand on their own systems. They may not even consciously be testing all the features that have changed. But in the act of working on their own specific task, they may notice when something unrelated has gone wrong.

Eric Raymond has said on multiple occasions, "Given enough eyeballs, all bugs are shallow." We might offer a corollary: "Given a few eyeballs less, many bugs are shallow."

2.3.5. Inspiration from Elsewhere

Inspiration can come from surprising places.

C++ programmers have been doing automated unit tests of their code for decades. There are libraries, IDEs, seminars, and books that help with this task. And yet… a lot of developers haven't tried incorporating this practice yet. They may be working in domains where these kinds of unit tests are less helpful. Or they may just not have seen enough value in the practice to push them past the initial stage of the learning curve.

There are other programming communities, though, where having code-level tests is practically a given for any project. There are both cultural and technological reasons for this. If every library you download has a test directory and the language itself comes with a test framework, new developers feel more encouraged to write tests for their own code.

A related practice is refactoring: changing the design of an existing program piece by piece, without breaking it. The Smalltalk folks made this practice famous, but you don't have to be a Smalltalk developer to use the techniques. I finally began to gain a toehold of understanding in this subject when the book *Refactoring: Ruby Edition* came out. Seeing the examples in the language I use for fun (Ruby) made it much easier to apply the concepts in the language I use at work (C++).

2.4. Social Beings

2.4.1. TPS Reports With No Cover

Ever seen the movie *Office Space*? The main character, a programmer, gets told off by manager after manager for failing to use the new, presumably improved, cover sheet for his TPS report. The episode was meant to point out a dysfunctional organization. But there's a second lesson in there as well. Sometimes the "official" format for documentation is a good thing. At other times it slows you down.

In my talk at PNSQC last year, I told the story of a brainstorming session with our team's software lead, where we typed straight into our text editors and generated a preliminary PDF of the requirements for a particular feature. That PDF was the first thing we checked into source control. Sure, the requirements weren't duly marked, numbered, and annotated—yet.

Think of it as a kind of "documentation debt." We knew we were eventually going to convert the document into the official format with the required template. But if we'd started there, we'd never have gotten off the ground.

2.4.2. Playing Hooky

More work time equals more work done—sounds logical, right? And yet, we need to recharge. We need to be able to change directions mentally. Spending all of our time, up to the last minute, on a single urgent project brings diminishing returns.

What else is there to do during office hours besides work? How about going to the occasional conference? Since we're all attending PNSQC as we speak, I'm assuming you're all receptive to the idea that a few days spent outside the lab once in a while are a good thing.

By slacking off and going to hang out with people who inspire us, we learn more about our craft. We sharpen ideas. We gain renewed energy for attacking the great problems that await us back in the office.

2.4.3. The Heist

We spend most of our workday chained to one main computer. The machine is sufficient for day-to-day coding, and then…. Suddenly we need a server for a quick experiment. Or a machine that can run OmniFocus. Or a computer with PhotoShop installed. Or a laptop we can try four different Linux distros on for compatibility testing.

Any office that's big enough will have dark corners with unused computers lying around. Maybe it's a hand-me-down from someone who just got an upgrade, or a VM someone spun up in the server farm weeks ago for testing.

We shouldn't let good hardware go to waste. If we need computing power, we should feel free to ask for it and exercise it. Notice I said, "ask." This is one of those rare times in the talk when I'll use the imperative voice. Please don't steal a machine someone was about to use, or make people wonder where missing hardware went.

It's okay to fancy ourselves as scrappy, resourceful underdogs, as long as we're above board about the whole thing. We communicate, communicate, and communicate again. We find out who owns the box, and let them know how long we'll be using it. We leave a big giant Post-It™ note in its absence. We give it back as soon as anyone needs it, or when our experiment comes to an end.

2.4.4. Job Transformers

How many of you here work at a fairly large company? A lot of us, it looks like. There are a few things big organizations can do to help a software developer. Even the most cynical cubicle dweller can appreciate the ability of a large shop to get lab space and expensive hardware.

Of course, we all know the downsides of working in cubeville: bureaucracy, heavyweight process, lack of tool choice, and lack of autonomy, just to name a few. These aren't unique to big companies, and there are a few shops that seem to be able to escape these pitfalls as they scale. But the stereotype exists for a reason.

The tech news sites I visit to learn about technological developments also feature heavy coverage of small startup companies. I sometimes read about scrappy two-person shops with no formal process, no required documentation format, and so on. I confess to occasional feelings of envy.

What can we do about these feelings?

We can shrug and get back to work. We can leave and join (or found) a startup. Or we can transform the job we have into the job we want.

How do we do this? There's no universal answer, and I'm certainly not presuming to give career advice to a room full of people more experienced than me. But I can at least tell you about what I'm doing. Maybe it's foolish. Maybe the other shoe will drop any day now and I'm going to have to stop doing this stuff. Until then, I'm going to keep using my favorite coping mechanism: experimentation.

Any given day presents us with little pockets of downtime: a meeting ends early, or a software build takes extra time. We can use those pockets to conduct experiments. We might download and try out a new programming language, set up a continuous integration server, or add a few tests to a subsystem that sorely needs it.

Or we might think a little bigger. We could learn our way around illustration software and use it to draw a user interface for an imaginary product. We could buy a cup of coffee for an industrial designer and ask about what's going on in their area of expertise.

Most of these little experiments won't result in anything tangible being made. Even so, they do us a service by challenging us. Every once in a while, an experiment might lead to a good idea. This is the part where we show it off to whatever receptive audience we can find: our teammates, our managers, etc.

Developing an idea and communicating it well enough to be adopted means we might get to spend time on it in a more official way, not just eking something out between compiles. In other words, we've just transformed a tiny piece of our career.

3. Conclusion

Over the past hour, we've discussed several different software practices. Some were code-level techniques, while others had more to do with project organization or personal development. None are things anyone would universally recommend.

So, why even bring them up? Because there are times when a blunt tool— a sledgehammer or a stick of dynamite—is exactly what you need to get a piece of recalcitrant software unblocked.

You can go your whole career making great software without doing any of the things we've discussed today. The real lesson here is: if you're stuck with a big puzzle in the form of legacy code, start somewhere. Start with something small you can do right now.

Don't get so paralyzed by analyzing new processes, new ways of working, or big changes that you end up not starting anything. What's the one thing you can do today that will take the most pain out of building software?