Deleting code

Created 22 December 2002, last updated 6 November 2012

This document is also available in Russian.

There's plenty of information out there about how to write code. Here's some advice on how to delete code.

The best way to delete code

This may seem obvious, but I guess it isn't, because of the variety of other ways developers have of deleting code. Here's how to delete code:

> Select a section of code in your editor, hit the backspace key, and be done with it.

Most developers don't like getting rid of stuff. They want to keep chunks of code around in case they need them again. They worked hard to write that chunk of code. They debugged it, it works. They don't want to just throw it away.

These developers want to keep their old code around, and they do it by disabling it in some way: commenting it out, conditionalizing it, or just not calling it anymore.

To those developers, I say, "Use the source (control), Luke". A source code control system (like Git, Mercurial, or Subversion), means you never have to worry that something is gone forever. Your repository will have the old code if you need it again.

If you don't have a source control system (!?!?!) or just don't want to be bothered digging back through the history, then copy the chunk of code to a separate file some place, and save it away. But don't leave it where it doesn't belong: in your source code.

What's the big deal?

If you have a chunk of code you don't need any more, there's one big reason to delete it for real rather than leaving it in a disabled state: to reduce noise and uncertainty. Some of the worst enemies a developer has are noise or uncertainty in their code, because they prevent working with it effectively in the future.

A chunk of code in a disabled state just causes uncertainty. It puts questions in other developers' minds:

- Why did the code used to be this way?
- Why is this new way better?
- Are we going to switch back to the old way?
- How will we decide?

If the answer to one of these questions is important for people to know, then write a comment spelling it out. Don't leave your co-workers guessing.

Commenting out code

It's very easy to comment out a line or two (or twenty!) lines of code:

```
//  OldWayStepOne(fooey);
//  OldWayStepTwo(gooey);
    NewWay(fooey, gooey);
```

This is bad. Comments should be used to provide people with information they need when reading or writing the code. They should be used to help the future developers who will be working with the code. These comments don't do that. In fact, they do just the opposite. In addition to removing the old code from being compiled, these comments add confusion, uncertainty, and doubt into the code.

Future developers looking at this code know that it used to work the OldWay, and they know that now it works the NewWay, but they don't know why the OldWay has been kept around:

- Maybe NewWay is just an experiment? If so, what's better about it? How and when will the final decision be made to keep it?
- Maybe OldWay is better, but there was something wrong with it? If so, what was wrong with it? Was it something wrong with the OldWay code, or they way we're calling it? When will it be fixed?
- Maybe the design has changed, and OldWay is doing unnecessary work?

Any commented-out code is an implicit question: Why is this still here? There are reasons to keep a piece of commented-out code. Changes get made that you know will be reversed soon. Changes get made that the developer is uncertain of. It's OK to keep the code, but say why you're keeping it. Comments are for people, and a line of code in a comment doesn't tell anyone anything.

> Don't comment out a piece of code without saying why (in the comment).

Isn't this better?:

```
//  OldWay did a better job, but is too inefficient until MumbleFrabbitz
//  is overhauled, so we'll use NewWay until the M4 milestone.
//    OldWayStepOne(fooey);
//    OldWayStepTwo(gooey);
    NewWay(fooey, gooey);
```

Now, who knows if MumbleFrabbitz will really be overhauled for the M4 milestone? Maybe it won't be. That's OK; who knows what the future will bring? But at least this way the developers will know why the code is being kept around. With the information about why the change was made, and why the old code is still there, the developers will know when they can finally fully commit to the NewWay, or when they can switch back to the better solution.

Conditional compilation

Developers who want to comment out large chunks will use conditional compilation instead (if the language supports it). In C++:

```
#if 0
    OldWayStepOne(fooey);
    ...
    OldWayStepTwenty(hooey);
#endif
```

In Python:

```
if 0:
    OldWayStepOne(fooey)
    ...
    OldWayStepTwenty(hooey)
```

This is no better than commenting out the code: it's just more convenient for whoever is doing the removing. In fact, in some ways it is worse than commenting out the code. Some IDEs don't syntax-color this code as a comment, so it's easy for other developers to read this code and not realize it has been disabled.

The same rule applies as for commenting out code:

> Don't conditionalize away code without explaining why.

If you must use the C preprocessor to remove code, "#if 0" is really the best way to do it, since it is at least clear that the code should never be compiled.

At Lotus, the source code for Notes include many lines of code removed with "#ifdef LATER", under the (correct) assumption that there was no preprocessor symbol called "LATER". This is a very weak form of documentation; it indicates that the code isn't ready to be compiled yet, but that it will be later. But when? A running joke among the developers was that we should define “LATER” and see what happened!

By using never-defined symbols to remove code, you leave doubt in developers minds as to what the symbols mean. Maybe there’s a configuration of the code called “LATER” that has to be taken into account.

Uncalled code

Let’s say you have a great class, and it has many methods. One day you discover that you no longer are calling a particular method. Do you leave it in or take it out?

There’s no single answer to the question, because it depends on the class and the method. The answer depends on whether you think the method might be called again in the future. A coarse answer could be: if the class is part of the framework, then leave it, if it is part of the application, then remove it. (I’ll have to write another piece about framework vs. application).

Leaving pointers

One compromise that you might consider is to remove a large chunk of unused code, but leave behind a pointer to where it could be found if it were needed. I’ve used comments like this before:

```
//  (There used to be another algorithm here that used hashing, that
//  was faster, but had race conditions.  If you want it, it's in
//  commit 771de15b or earlier of ThingMap.java in the repo.)
```

It’s small, it’s unobtrusive, but it gives a little history, and a place to go looking for more information.

Accidental droppings

Sometimes, while writing code, you really are unsure about whether to keep or delete a line of code, and you want to try compiling or running the code before you decide what to do. You comment out the line. A number of files get changed, and by the time you are ready to check in the code, you’ve forgotten where all those temporary removals are. You check in the code, and you’ve left accidental droppings all over the place.

> Always use a distinctive marker in your commented-out lines of code, so you can quickly find them all when it’s time to clean up and check in.

A simple convention like this:

```
//- OldWayImUnsureOf(zooey);
```

makes all the difference. By using “//-” to comment out the line, you’ve left a marker that you can easily find when you are getting ready to check in your code.

You can use it for larger chunks as well:

```
#if 0 //- I don't think I need this with the new FooBar
    OldWayStepOne(fooey);
    ...
    OldWayStepTwenty(hooey);
#endif
```

Keep things tidy

While deleting code, it is all too easy to leave phantom stubs behind. Try hard to trim these properly. For example, when getting rid of OldWay here:

```
if (bDoThing) {
    OldWay();
}
```

Don’t just take out the line calling OldWay. Get rid of the empty if as well. Then if bDoThing was only tested here, also get rid of it. Examine the code that set bDoThing. Is it now obsolete? Get rid of it. Be merciless. Keep the code tidy. Make sure it makes sense with no dead-end off ramps that can only be understood by knowing what used to be there.

It is tempting to leave this code in, because it will be difficult to understand whether it is all still needed or not. But if you leave the empty if clause, some other developer will come along later, and see it, realize it can’t be right, and have to investigate. It will take them longer to understand the empty if than it would take you to remove it.

Don’t worry, be happy

I know it seems drastic to just chop out code that you sweated over. Don’t worry: it will be OK. There’s a reason you wanted to disable it or whatever. The source control system will still have a copy if you need to go back to it. Look at it this way: what are the chances you need to go back and get it, compared to the certainty that you’ll have to be looking at those stupid commented-out lines for the rest of the project’s life?

Go ahead, delete that old code. You won’t miss it.

See also

- Erroneously empty code paths, about mistakes in defensive coding.

- Fix error handling first, about ensuring that your error handling code is always running its best.
- My blog, where other similar topics are discussed.