Psychic debugging: When reading unfamiliar code, assume it's mostly correct

devblogs.microsoft.com/oldnewthing/20070423-00

April 23, 2007

Raymond Chen

You may be called in to study a problem in code you've never seen before or be asked to look over a proposed change to some code you've never seen before. When this happens, you have to take shortcuts in your analysis because following every function call to the bottom would not only take far too much time, but also take you so far away from the code in question that you will probably forget what you were looking for in the first place.

For example, suppose you're looking at some code that goes like this:

```
...
Gizmo *gizmo = get_gizmo_from_name(name);
if (gizmo) {
 Gizmo *parent = gizmo->get_parent();
 parent->set_height(newheight);
 ...
}
```

You might have some questions about this code.

- What if `name` is `NULL`? Is it legal to pass `NULL` to `get_gizmo_from_name`?
- What if the `gizmo` doesn't have a parent? Is there a potential `NULL` pointer dereference here?
- Are the `gizmo` and `parent` reference-counted? Did we need to do something like `gizmo->Release()` or a `parent->Release()` to keep the reference counts in balance and avoid a memory leak?

Finding the answers to these questions may take some time. For example, you might have access only to the diff and not to the entire project, or a grep for the definition of `get_gizmo_from_name` in the same directory that has the function in question doesn't turn up anything and you have to expand your search wider and wider in an attempt to find it.

This is when you invoke the "Assume it's mostly correct" heuristic. The theory behind this heuristic is that whoever wrote this code has a better understanding of how it works than you do. (This is a pretty safe bet since your knowledge of this code is approximately zero.) The problem you're looking for is probably some small detail, an edge case, a peculiar combination of circumstances. You can assume that the common case is pretty solid; if the common case were also broken, the problem would be so obvious that they wouldn't need to ask an outsider for help.

Therefore, look at the other parts of the code. For example, you might find a code fragment nearby like this one:

```
// rename the gizmo
Gizmo *gizmo = get_gizmo_from_name(oldname);
if (gizmo) {
 gizmo->set_name(newname);
}
```

That already answers two of your questions. First, you don't have to worry about checking the name against `NULL` because this code fragment doesn't check, and by the heuristic, the code is mostly correct. Therefore, `NULL` is most likely an acceptable parameter for the `get_gizmo_from_name` function. Because if it weren't, then that code would be broken too! (This is sort of the counterexample to what Mom always told you: If everybody else jumped off a bridge, then it is probably okay to jump off bridges.)

Second, this code doesn't do anything special when it's done with the `gizmo` so it's probably okay just to abandon the `gizmo` without need to do any special reference count management. Because if you had to dispose of it in a special way, then that code would be broken too!

Now, of course, this heuristic can be fooled, but if you're operating with only partial information, it's often the best you can do. Get it right often enough and people will believe that you too have psychic debugging powers.

Raymond Chen

Follow