Making Myself Better: What Craftspeople Can Teach Us about Software

SHARE

PUBLISHED MAY 25, 2020
AUTHOR WOLFGANG BANGERTH

TOPICS BETTER PLANNING, BETTER RELIABILITY, SOFTWARE ENGINEERING, DEBUGGING, TESTING

TRACK DEEP DIVE

Anticipating what could go wrong is an essential part of becoming a master craftsperson. An important part of that process is reflecting on and learning from mistakes made along the way. This blog article draws analogies to perspectives in defensive programming.

From journeyman to master craftsperson

We tend to have a romanticized notion of craftspeople and artisans as skillful creators striving for perfection, every time. That perception may of course be misplaced, but at least historically, becoming a master stonemason, master carpenter, or master metal maker was a serious undertaking in my country of origin, Germany: it required three years and one day as a journeyman. During that time, apprentices would live a rather spartan life: they would travel the country, stay with and learn from masters, and forego marriage as well as most worldly possessions other than what they could carry.

Maybe we don't need to apply quite such monk-like dedication to becoming good software engineers. But I think that the way I imagine these journeyman years to have been, still holds useful lessons. I suppose that a lot of classes in "STNM 501: Intro to Stone Masonry" would have been of the form "If you put the chisel to this place of the rock, it will flake off in that way, and you end up with the shape you are looking for." That's like writing software: translating an abstract algorithm into code. The part that I imagine as so much more interesting comes in "STNM 502: Advanced Stone Masonry": "Think about what happens if you put the chisel in that place but you accidentally hit it at the wrong angle with your hammer because you're distracted by the beautiful sunset? Then you flake off a much larger piece, and because you can't glue rock back together, the piece is ruined and you need to start from scratch." To put this differently: when we do things, we shouldn't just know how to do things right, but we should *plan for the kinds of mistakes we tend to make* on a daily basis. It is this kind of planning that makes the "every time" possible in my first sentence above.

I have kind of a long history with this notion of planning for failure — longer than I've been programming. Some 30 years ago, when I was a teenager, my mom told me about a colleague who had slipped on icy stairs, and — *because his hands had been in his pockets* — couldn't catch his fall and cracked a couple of ribs. I really can't say why this particular story stuck with me, but it's become a key piece of how I've approached programming: keep your hands out of your pockets if there is a risk that you might slip. Or, more concretely: even if you are pretty sure that your code is correct, build a safety net anyway. In software design, this would be called defensive programming, which is a form of defensive design.

It's a mindset question

It's definitely a good habit to use defensive techniques, and I'll provide a few examples of defensive programming below. But I think that the *real* lesson of the story is actually a different one: we all make mistakes. Some have consequences that are quite annoying (cracked ribs, broken software). But what I think sets good programmers apart is not that once they run into a bug, they fix it. It is that *then they go for a walk around the block*, go back in time and wonder what was on their mind when they wrote the code with the bug. *How* did it come that I wrote the code with the bug? What could I have done differently to avoid this bug in the first place? What could I have done to find the bug quicker, and what mistakes did I make that resulted in it taking this long?

The idea of this mindset is not to be annoyed at having to spend a couple of hours fixing a bug, but rather to see it as an opportunity to learn how to become better at programming, debugging, or essentially anything else. To learn something about how *you* work and how you can work around your own limitations. To see cracked ribs as an opportunity to understand that slipping on ice happens, but that it doesn't have to have grave consequences if one anticipates it and guards against it.

An example

Let me show you a simple piece of code, not untypical of the kind we all write when dealing with geometry in three-dimensional spaces:

```
class Point3d
{
  private:
    double coordinates[3];
  public:
    double &  operator[] (const unsigned int c)
    {
      return coordinates[c];
    }
};
```

Sooner or later, we will want to add points together:

```
Point3d operator+ (const Point3d &p1,
                   const Point3d &p2)
{
  Point3d x;
  for (unsigned int c=0; c<=3; ++c)
    x[c] = p1[x] + p2[c];
  return x;
}
```

I suspect you all noticed: there's a bug in this function — the loop should have run only while `c<3`, not `c<=3`. That's a nuisance. The bug is likely difficult to find because the three coordinates of the sum object were computed correctly, but I wrote past the end of the `coordinates` array. So the error would not have been visible immediately and would have manifested only when I found myself confused that some other variable did not seem to store the value it should have had. It may take an afternoon to debug this problem.

These things happen. I know I make mistakes, I fix them, I walk around the block once and *resolve to be more careful with loop indices in the future*. I feel good about having found the bug and about having thought about its root causes.

Three months later I write code that has a similar problem:

```
double dot_product (const Point3d &p1,
                    const Point3d &p2)
{
  double sum = 0;
  for (unsigned int c=0; c<=3; ++c)
    sum += p1[x] * p2[c];
  return sum;
}
```

This time, my walk around the block is not as pleasant. I have to admit to myself that I thought I had a plan last time, but that it didn't work — apparently, my brain makes these kinds of mistakes. Resolving to be more careful doesn't seem to work: I need a different strategy *that acknowledges that I make these kinds of mistakes*, that I make them frequently, and that apparently I am unable to stop myself from making them.

The way to do this in the current context is to let the program itself help me find those places where I make mistakes:

```
class Point3d
{
  ...
    double &  operator[] (const unsigned int c)
    {
      assert(c<3);
      return coordinates[c];
    }
};
```

The assertion will terminate the program if the condition `c<3` is not satisfied. For both of the wrong pieces of code above, I would have been alerted to the problem right away, and it would have been a matter of a minutes to fix them — no need for difficult debugging.

The point I'm trying to make is not that the assertion is a good way to help me debug code (though of course it is), but that I put it there because *introspection has allowed me to understand what kinds of mistakes I often make*, and how I can learn to live with it.

Defensive programming

Assertions are one component of "defensive programming". I use them extensively in all of the codes I work on. For example, the deal.II finite element library's core currently consists of a large amount of C++ code, of which about 122,700 lines contain a semicolon (a measure of the size of a code); nearly 12,800 of these lines contain an assertion. It is not uncommon to see functions such as the following one that extracts from a solution vector those values that are active on a cell to which a `CellAccessor` object points (lightly edited to avoid unnecessary details):

```
template <typename ForwardIterator>
void
CellAccessor::get_dof_values(const Vector      &values,
                             ForwardIterator    local_values_begin,
                             ForwardIterator    local_values_end) const
{
  Assert(this->dof_handler != nullptr,
         ExcMessage("The CellAccessor object is not initialized."));
  Assert(this->is_artificial() == false,
         ExcMessage("Can't ask for DoF indices on artificial cells."));
  Assert(this->is_active(),
         ExcMessage("Cell must be active."));
  Assert(local_values_end - local_values_begin == this->get_fe().dofs_per_cell,
         ExcMessage("The output range has the wrong size."));
  Assert(values.size() == this->get_dof_handler().n_dofs(),
         ExcVectorDoesNotMatch());

  const types::global_dof_index *cache =
    this->dof_handler->levels[this->present_level]
    ->get_cell_cache_start(this->present_index, this->get_fe().dofs_per_cell);
  DoFAccessorImplementation::Implementation::
    extract_subvector_to(values,
                         cache,
                         cache + this->get_fe().dofs_per_cell,
                         local_values_begin);
}
```

The actual implementation in the last few lines is unimportant here; the point simply is: the function has three input arguments (plus the `this` pointer) that could all be wrong or in undefined states, and we should check that they are valid individually and in combination — because you bet that at some point someone will write code calling this function and do so while the sun is setting beautifully, and it will be immediately followed by a night figuring why the code doesn't work.

Many other strategies make it easier for me to live with the bugs I often seem to introduce. For example, I mark as many variables as possible as `const` because it helps me avoid accidentally modifying things I assume have a fixed value. I use a naming scheme and consistent indentation style everywhere because that makes reading the code easier. If a variable counts something, it is marked as `unsigned`. As in the code above, I assert that all input variables of a function are within valid ranges.

Other approaches that fall into the same category are of course that all software needs to have good tests. Every experienced programmer knows that complex software that isn't tested isn't just in the category where it *may* not work, but in fact in the category where it *does* not work. For software that continues to be developed (nearly all software), that also implies that testing must be part of the development process, for example as part of a Continuous Integration framework. Some would advocate that tests should also check every assertion in the code to ensure that they trigger when they are needed; in my view, that may be going too far, but I appreciate the spirit of anticipating that one may also make mistakes writing assertions.

Summary

The point I want to make here is not that defensive programming is good (though it unambiguously is), but what made people think about it in the first place: all of these techniques ultimately came about because of unpleasant walks around the block coming to terms with the fact that we make mistakes and apparently can't seem to stop ourselves from making mistakes.

This *process* of *reflecting about what we do and how we do it* is an important component at getting better at what we do. We can do this by walking around the block after each debugging session. Others keep a journal in which they write their insights about how they work, possibly illustrating their thoughts with code snippets or screenshots from their own work. I require students in some of my classes to do this, and indeed there is research that says that this kind of "reflective writing" leads to deeper learning of processes and practice. At the end of the day, what's important is *forcing* ourselves to be introspective and learn how we, as software engineers, operate.

Author bio

Wolfgang Bangerth is a professor of mathematics and (by courtesy) geosciences at Colorado State University. In 1997, he founded and is now one of the Principal Developers of the deal.II project that provides finite element functionality from laptops to supercomputers. In 2011, he also co-founded and has since been a Principal Developer of ASPECT, the Advanced Simulator for Problems in Earth ConvecTion, a software package for the simulation of convection in the Earth mantle and the dynamics of Earth's crust.

Despite his apparent focus on safety nets, he tremendously enjoyed skydiving for a few years. He's into hiking and climbing the exposed ridges going up and between Colorado's highest peaks these days. (He does, however, carry a satellite communicator on these trips.)

Before joining Colorado State University, he was on the faculty of Texas A&M University and a postdoc at the University of Texas at Austin. He received his PhD from the University of Heidelberg, Germany, in 2002.

Comment

0 Comments Wolfgang Bangerth

W Start the discussion…

Share Best Newest Oldest

Be the first to comment.

Subscribe Privacy Do Not Sell My Data

More on Software Engineering, Debugging, and Testing

[Growing Resilient Scientific Software Ecosystems: Introducing the Software Gardening Almanack]

Deep Dive

PUBLISHED APR 16, 2025

BY DAVE BUNTEN, WILL DAVIDSON, AND GREGORY P. WAY

[Come for Syntax, Stay for Speed, and Understand Bugs in Julia Programs]

Experience

PUBLISHED FEB 10, 2024

BY AKOND RAHMAN

[Developing Coding Standards and Practices for Sustainable Software Development]

Deep Dive and How To

PUBLISHED FEB 13, 2025

BY MANOJ K. BHARDWAJ

Reflecting on Our Community: The SC24 BoF on Scientific Software and the People Who Make it Happen: Building Communities of Practice

Community

PUBLISHED MAR 23, 2025

BY DAVID E BERNHOLDT, JEFFREY C. CARVER, IAN A. COSDEN, ANSHU DUBEY, WERONIKA FILINGER, SANDRA GESING, MOZHGAN KABIRI CHIMEH, LAUREN MILECHIN, MARION WEINZIERL, HARSHITHA MENON, AND ADDI MALVIYA THAKUR

Celebrating the Fifth Anniversary of the Correctness Workshop: Looking Back and Looking Forward

Community

PUBLISHED FEB 22, 2022

BY IGNACIO LAGUNA AND CINDY RUBIO-GONZÁLEZ

- BSSw Fellowship Program
- Policies
- Site Contributors

- Contact BSSw
- Receive Our Email Digest
- Follow Our RSS Feed