**Software Development Is…an Art**

A great programmer needs to be, in part, a great artist. But is programming *really* an art? This is a debate that has long been held in software development circles. Some people think that programming is an engineering discipline, some an art form, some sit in-between, considering it a craft (I did call my first book *Code Craft,* after all).

Knuth is probably the most famous proponent of software as art, naming his famous series of books *The Art of Computer Programming.* He said this: *Some programs are elegant, some are exquisite, some are sparkling. My claim is that is it possible to write grand programs, noble programs, truly magnificent ones!* Stirring stuff.

There’s more to code than bits and bytes, more than brackets and braces. There’s structure and elegance. There’s poise and balance. There is a sense of taste and aesthetics.

**Key**

A programmer needs good taste and a sense of aesthetics to write exceptional code.

There are many parts of the software development process akin to the creation of a work of art. The process is:

Creative

It requires imagination. The software must be skilfully constructed and precisely designed. Programmers must have a vision for the code they are about to create, and a plan of how they will make it. Sometimes that involves a great deal of ingenuity.

Aesthetic

Good code is hallmarked by elegance, beauty, and balance. It stands within the framework of certain cultural idioms. We consider the code’s form alongside its function.

Mechanical

As any artist, we work in our particular medium with our particular tools, processes, and techniques. We work under commission for generous benefactors.

Team-based

Many forms of art are not single-person endeavours. Not every art form sees an artist sitting alone in their studio slaving day and night until their masterpiece is complete. Consider master sculptors with their apprentices. Consider the orchestra, each member held together by the conductor. Consider a musical composer, writing a piece which will then be interpreted by the performer(s). Or the architect designing a building that will be erected by a team of builders.

In many respects, the skill set of an artist is similar to that of a programmer.

Michelangelo was the archetypal renaissance man: a painter, sculptor, architect, poet, and engineer. Perhaps he would have made an incredible programmer. When asked about how he created one of his most famous works, the statue of David, he said: *I looked into the stone and saw him there, and just chipped away everything else*.

Is that what you do? Do you reduce and remove the complexities of the problem space, chipping them all away until you reach the beautiful code you were aiming for?

Here are a few questions to ask yourself on the theme of software as art:

* Do I consider the creative aspects of software development, or do I treat it as a mechanistic activity?
* Should I develop a keener sense of elegance and aesthetics in my code? Should I look beyond what’s functional and solves the immediate problem?
* Do I think that my idea of “beautiful” code is the One True Opinion? Should I consider artistry as a team pursuit?

# Software Development Is…a Science

We talk about computer science. So there must be something vaguely scientific going on somewhere, mustn’t there? It’s probably fair to say that in most development organisations there is much less science and far more plumbing happening.

The archetypal scientist is, of course, Albert Einstein. He was not only a genius, but also one of the most quotable people there has ever been (which helps authors considerably). He said this: Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius—and a lot of courage—to move in the opposite direction.

That is really profound; inappropriate complexity is a real killer in most software projects.

Einstein was also an aesthete. He appreciated elegance and beauty in his theories, and aimed to reduce things to a coherent whole. He said: I am enough of an artist to draw freely upon my imagination. Imagination is more important than knowledge. Knowledge is limited. Imagination encircles the world.

See, I told you he was quotable.

So if software development is like a science, what does that mean? It is (or, rather, should be):

Rigourous

We look for bug-free code that works, all the time, every time. It must work with all sets of valid input, and respond appropriately to invalid input. Good software must be accurate, proven, measured, tested, and verified.

How do we achieve this? Good testing is key. We look for unit tests, integration tests, and system tests. Preferably automated to remove the risk of human error. We also look for experiential testing.

Systematic

Software development is not a hit-and-miss affair. You can’t aim to create a well-structured large computer system by randomly accreting blobs of code until it appears to work. You need to plan, design, budget, and systematically construct.

It is an intellectual, logical, rational process; bringing order and understanding out of the chaos of the problem space and the design alternatives.

Insightful

Software development requires intellectual effort and astute analytical powers. This is especially apparent when tracking down tricky bugs. Like scientists, we form hypotheses, and apply something akin to scientific method (form a hypothesis, work out experiments, run experiments, and validate the theory).

###### Key

Good software development is not cowboy coding, throwing down the first code you can think of. It is a deliberate, considered, accurate endeavour.

Based on that, ask yourself:

* Is my software always totally correct and completely accurate? How do I prove this? How can I make this explicit, now and in the future?
* Do I strive to bring order out of chaos? Do I collapse complexity in my code until there are a few, small, unified parts?
* Do I approach problems methodically and thoughtfully, or do I rush headlong into them in an unstructured way?

# Software Development Is…a Sport

Most sports require great skill and effort: tenacity, training, discipline, teamwork, coaching, and self-consciousness. Likewise, software development involves:

Teamwork

It requires the concert of many people, with different skills, working in harmony.

Discipline

Each team member must be committed to the team, and willing give their best. This requires dedication, hard work, and a lot of training.

You can’t get good at soccer by sitting on a couch and watching soccer training videos. In fact, if you do it with a few beers and a tub of popcorn, you’re likely to get worse at soccer! You have to actually do it, get out there on the pitch with people, practise your skills, and then you’ll improve. You must train—have someone tell you how to improve.

The team must practise together, and work out how to function as a whole.

Rules

We’re playing to (developing to) a set of rules, and a particular team culture. This is embodied in our development processes and procedures, as well as the rites and rituals of the software team and their tool workflows (consider how you collaborate around things like the source control system).

The teamwork analogy is clearest with a sport like soccer. You work in a group of closely functioning people, playing a game by a set of well-defined rules.

Have you seen a team of seven-year-olds playing soccer? There’s one small guy left back standing in the goal mouth, and every other kid is running around the pitch maniacally chasing the ball. There’s no passing. There’s no communication. There’s no awareness of the other team members. Just a pack of children converging on a small moving sphere.

Contrast that to a high-quality premier league team. They operate in a much more cohesive way. Everyone knows their responsibility, and the team works cohesively together. There is a shared vision that they work towards, and they form a high-functioning, well-coordinated whole:

* Do I have all of these skills? Do I work well in a team, or could I improve in some areas?
* Am I committed to my team, willing to work for the good of everyone?
* Am I still learning about software development? Do I learn from others, and am I perfecting my team skills?

# Software Development Is…Child’s Play

For me, this observation seems particularly appropriate; I’m really just a child at heart. Aren’t we all?

It’s fascinating to see how children grow and learn, how their world view changes and is shaped by each new experience. We can glean a lot from the way a child learns and reacts to the world.

Consider how this applies to our software development:

Learning

A child is aware that they are learning, that they don’t know everything. This requires a simple characteristic: humility. Some of the programmers I have found hardest to work with think that they know it all. If there’s something new they need to know, they read a book and then presume that they’re an expert. A total humility bypass.

A child is constantly assimilating new knowledge. We must recognise that if we want to improve, we must learn. And we must be realistic about what we do, and do not, know.

Enjoy learning, savour finding out new things. Practise and improve your craft.

###### Key

Good programmers work with humility. They admit that they don’t know it all.

Simplicity

Do you write the simplest code possible? Do you reduce everything to the least complex form to make it easier to understand and easier to code?

I love the way kids try to get to the bottom of things, to understand things from their own limited perspective. They’re always asking why. Take, for example, a conversation I had with my daughter when she was six: Daddy, why is Millie my sister? Because you’re in the same family as her, Alice. Why? Well, because you have the same mummy and daddy. Why? Because, well, you see, there are the birds and the bees… Oh go and get a book! … (thinking) … Why?…

We should be constantly asking why—questioning what we are doing and the reasons for it. Seeking a better understanding of the problem and the best solution. And we should strive for simplicity in our handiwork. That is not the most simplistic “dumb” code possible, but the appropriately non-complex code.

Having fun

If all else fails, there’s nothing wrong with this. All good developers enjoy a little playtime. My office currently houses a unicycle and a makeshift cricket pitch.

With that in mind, we can ask ourselves:

* Do I strive to write the simplest code possible? Or do I type what comes to mind, and not think about commonality, refactoring, or code design?
* Am I still learning? What can I learn about? What do I need to learn about?
* Am I a humble programmer?

# Software Development Is…a Chore

A lot of our software development work is not pleasant. It’s not glamourous. It’s not plain sailing. It’s just donkeywork that has to be done to get a project completed.

To be an effective programmer, you mustn’t be afraid of the chores. Recognise that programming is hard work. Yes, it’s great to do a cool design on the newest product version, but sometimes you need to do the tedious bug fixing and grubbing around the old awful messy code to get a product shipping and make some money.

From time to time we must become software janitors. This requires us to:

Clean up

We must spot problems and address them; work out where breakages are and what the appropriate fixes are. These fixes must be made in a timely and nondisruptive manner. A janitor does not leave the unpleasant tasks to someone else, but takes responsibility for them.

Work in the background

Janitors do not work in the limelight. They probably receive little recognition for their heroic efforts. This is very much a supporting, not a lead role.

Maintenance

A software janitor will remove dead code, fix broken code, refactor and rebuild inappropriate workmanship, and tidy and clean the code to ensure that it doesn’t fall into disrepair.

Ask yourself:

* Am I happy to do code chores? Or do I only want the glamourous work?
* Do I take responsibility for messy code and clean it up?