The Art in Computer Programming

Andrew Hunt
David Thomas
The Pragmatic Programmers, LLC

September, 2001

The following is an adaptation of material originally presented in Åarhus, Denmark, at the Java and Object Oriented Conference, on September 12, 2001.

What exactly is software development, and why is it so hard? This is a question that continues to engage our thoughts. Is software development an engineering discipline? Is it art? Is it more like a craft?

We think that it is all of these things, and none of them. Software is a uniquely human endeavor, because despite all of the technological trimmings, we're manipulating little more than the thoughts in our heads. That's pretty ephemeral stuff. Fred Brooks put it rather eloquently some 30 odd years ago[Bro95]:

> *"The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. (As we shall see later, this very tractability has its own problems.)"*

In a way, we programmers are quite lucky. We get the opportunity to create entire worlds out of nothing but thin air. Our very own worlds, complete with our own laws of physics. We may get those laws wrong of course, but it's still fun.

This wonderful ability comes at a price, however. We continually face the most frightening sight known to a creative person: the blank page.

1 Writer's Block

Writers face the blank page, painters face the empty canvas, and programmers face the empty editor buffer. Perhaps it's not literally empty—an IDE may want us to specify a few things first. Here we haven't even started the project yet, and already we're forced to answer many questions: what will this thing be named, what directory will it be in, what type of module is it, how should it be compiled, and so on.

The completely empty editor buffer is even worse. Here we have an infinite number of choices of text with which to fill it.

So it seems we share some of the same problems with artists and writers:

1. How to start
2. When to stop
3. Satisfying the person who commissioned the work

Writers have a name for difficulties in starting a piece: they call it *Writer's Block.*

Sometimes writer's block is borne of fear: Fear of going in the wrong direction, of getting too far down the wrong path. Sometimes it's just a little voice in your head saying "don't start yet". Perhaps your subconscious is trying to tell you that you're missing something important that you need before you can start.

How do other creative artists break this sort of logjam? Painters sketch; writers write a stream of consciousness. (Writers may also do lots of drugs and get drunk,

but we're not necessarily advocating that particular approach.) What then, is the programming equivalent of sketching?

SOFTWARE SKETCHES

Sometimes you need to practice ideas, just to see if something works. You'll sketch it out roughly. If you're not happy with it, you'll do it again. And again. After all, it takes almost no time to do, and you can crumple it up and throw it away at the end.

For instance, there's a pencil sketch by Leonardo da Vinci that he used a study for the Trivulzio equestrian monument. The single fragment of paper contains several quick sketches of different views of the monument: a profile of the horse and rider by themselves, several views of the base with the figures, and so on. Even though the finished piece was to be cast in bronze, da Vinci's sketches were simply done in pencil, on a nearly-scrap piece of paper. These scribblings were so unimportant that they didn't even deserve a separate piece of paper! But they served their purpose nonetheless.[1]

Pencil sketches make fine prototypes for a sculpture or an oil painting. Post-It notes are fine prototypes for GUI layouts. Scripting languages can be used to try out algorithms before they're recoded in something more demanding and lower level. This is what we've traditionally called prototyping: a quick, disposable exercise that concentrates on a particular aspect of the project.

In software development, we can prototype to get the details in a number of different areas:

1. a new algorithm, or combination of algorithms
2. a portion of an object model
3. interactions and data flow between components
4. any high-risk detail that needs exploration

A slightly different approach to sketching can be seen in da Vinci's *Study for the Composition of the Last Supper*. In this sketch, you can see the beginnings of the placement of figures for that famous painting. The attention is not placed on any detail—the figures are crude and unfinished. Instead, da Vinci paid attention to focus, balance and flow. How do you arrange the figures, position the hands and arms in order to get the balance and flow of the entire piece to work out?

Sometimes you need to prototype various components of the whole to make sure that they work well together. Again, concentrate of the important aspects and discard unimportant details. Make it easy for yourself. Concentrate on learning, not doing.

As we say in *The Pragmatic Programmer*[HT00], you must firmly have in your head what you are doing before you do it. It's not at all important to get it right the first time. It's vitally important to get it right the last time.

PAINT OVER IT

Sometimes the artist will sketch out a more finished looking piece, such as Rembrandt's sketch for *Abraham's Sacrifice Of Isaac* in 1635. It's a crude sketch that has all of the important elements of the final painting, all in roughly the right areas. It proved the composition, the balance of light and shadow, and so on. The sketch is accurate, but not precise. There are no fine details.

Media willing, you can start with such a sketch, where changes are quick and easy to make, and then paint right over top of it with the more permanent, less-forgiving media to form the final product.

To simulate that "paint over a sketch" technique in software, we use a Tracer Bullet development. If you haven't read *The Pragmatic Programmer* yet, here's a quick explanation of why we call it a Tracer Bullet.

There are two ways to fire a big artillery gun. The first way is to carefully measure the distance to the target, compensate for wind speed and direction, the weight of the ordinance, and so on, crunch all the numbers and give the orders to fire:

> "Range 1000!"
> *whirr. click.*
> "Elevation 7.42!"
> *whirr. click.*
> "Azimuth 3.44"
> *whirr. click.*

[1] Sadly, the project's sponsor canceled the monument due to lack of funds. Some things never change.

"FIRE!"
BOOM. *Oh bad luck, there. Missed.*
"Range 2015!"
whirr. click.
"Elevation 9.15!"
etc...

By the time you've set up, checked and rechecked the numbers, and issued the orders to the grunts manning the machine, the target has long since moved.

In software, this kind of approach can seen in any method that emphasizes planning and documenting over producing working software. Requirements are generally finalized before design begins. Design and architecture, detailed in exquisite UML diagrams, is firmly established before any code is written (presumably that would make coders analogous to the "grunts" who actually fire the weapon, oblivious to the target).

Don't misunderstand: if you're firing a really huge missile at a known, stable target (like a city), this works out just great and is the preferable way to go. If you're shooting at something more maneuverable than a city, though, you need something that provides a bit more real-time feedback.

Tracer bullets.

With tracer bullets, you simply fill the magazine with phosphorus-tipped bullets spaced every so often. Now you've got streaks of light showing you the path to the target right next to the live ammunition.

For our software equivalent, we need a skeletally thin system that does next to nothing, but does it from end to end, encompassing areas such as the database, any middleware, the application logic or business rules, and so on. Because it is so thin, we can easily shift position as we we try to track the target. By watching the tracer fire, we don't have to calculate the effect of the wind, or precisely know the location of the target or the weight of the ammunition. We watch the dynamics of the entire system in motion, and adjust our aim to hit the target *under actual conditions*.

As with the paintings, the important thing isn't the details, but the relationships, the responsibilities, the balance, and the flow. With a proven base—however thin it may be—you can proceed in greater confidence towards the final product.

GROUP WRITER'S BLOCK

Up till now, we've talked about writer's block as it applies to you as an individual. What do you do when the entire team has a collective case of writer's block? Teams that are just starting out can quickly become paralyzed in the initial confusion over roles, design goals, and requirements.

One effective way to get the ball rolling is to start the project off with a group-wide, tactile design session. Gather all of the developers in a room[2] and provide sets of Lego blocks, plenty of Post-It notes, whiteboards and markers. Using these, proceed to talk about the system you'll be building and how you think you might want to build it.

Keep the atmosphere loose and flexible; this gets the team comfortable with the idea of change. Because this is low-inertia design, anyone can contribute. It's well within any participant's skills to walk up to the whiteboard and move a PostIt-note, or to grab a few Lego blocks and rearrange them. That's not necessarily true of a CASE tool or drawing software: those tools do not lend themselves readily to rapid-feedback, group interaction.

Jim Highsmith offers us a most excellent piece of advice: The best way to get a project done faster is to start sooner. Blast through that writer's block, and just start.

JUST START

Whether you're using prototypes or tracer bullets, individually or with a group, you're working—not panicking. You're getting to know the subject, the medium, and the relationship between the two. You're warmed up, and have started filling that blank canvas.

But we have one additional problem that the painters do not have. We face not one blank canvas per project, but hundreds. Thousands, maybe. One for every new module, every new class, every new source file. What can we do

[2] If you've got more developers on the team than will fit in an ordinary room, then you've got bigger problems than we can address here.

to tackle that multiplicity of blank of canvases? The Extreme Programming[Bec00] notion of Test First Design can help.

The first test you are supposed to write—before you even write the code—is a painfully simple, nearly trivial one. It seems to do almost nothing. Maybe it only instantiates the new class, or simply calls the one routine you haven't written yet. It sounds so simple, and so stupid, that you might be tempted not to do it.

The advantage to starting with such a trivial test is that it helps fill in the blank canvas without facing the distraction of trying to write production code. By just writing this very simple test, you have to get a certain level of infrastructure in place and answer the dozen or so typical startup questions: What do I call it? Where do I put it in the development tree? You have to add it to version control, and possibly to the build and/or release procedures. Suddenly, a very simple test doesn't look so simple any more. So ignore the exquisite logic of the routine you are about to write, and get the one-line test to compile and work first. Once that test passes, you can now proceed to fill in the canvas—it's not blank anymore. You're not writing anything from scratch, you're just adding a few routines....

2 When to Stop

We share another problem with painters: knowing when to stop. You don't want to stop prematurely; the project won't yet be finished.[3] But if you don't stop in time, and keep adding to it unnecessarily, the painting becomes lost in the paint and is ruined.

There's only one way avoid either trap: feedback. Before you even start a particular task, you have to have a way to determine that you're done. For example:

[3] In software as well as in modern art, the distinction between intentional and accidental omissions is often difficult to make.

A...	is done when...
Project	Customer accepts
Development	Passes functional tests
Module	Passes unit tests
Bug fix	Test that previously failed now passes
Meeting	objective for meeting achieved
Document	Deliver exactly what's needed
Talk	Done when audience throws rotten fruit
Paper	You *are* still reading this, right?

We had a client once who seemed to have some difficulty in the definition of "done" with regard to code. After toiling for weeks and weeks on a moderately complex piece of software, Matthew (not his real name) proudly announced the Code Was Done. He went on to explain that it didn't always produce the correct output. Oh, and every now and again, the code would crash for no apparent reason. But it's done. Unfortunately, wishful thinking alone doesn't help us get working software out to users.

It's easy to err on the other side of the fence too—have you ever seen a developer make a career of one little module? Have you ever done that? It can happen for any number of political reasons ("I'm still working on XYZ, so you can't reassign me yet"), or maybe we just fall in love with some particularly elegant bit of code. But instead of making the code better and better, we actually run a huge risk of ruining it completely. Every line of code *not written* is correct—or at least, guaranteed not to fail. Every line of code we write, well, there are no guarantees. Each extra line carries some risk of failure, carries an additional cost to maintain, document, and teach a newcomer. When you multiply it out, any bit of code that isn't absolutely necessary incurs a shockingly large cost. Maybe enough to kill the project.

How then, can we tell when it's time to stop?

PAINTING MURALS

Knowing when to stop is especially hard when you can't see the whole thing that you're working on. Mural painting, for instance, takes a special eye. In corporate software development, you may only ever see the one little piece of detail that you're working on. If you watch mural painters up close, it's quite difficult to discern that the splash of paint they're working on is someone's hand, or

eyeball. If you can't see the big picture, you won't be able to see how you fit in.

The opposite problem is even worse—suppose you're the lone developer on a project of this size. Most muralists are simply painting walls, but anyone who's ever painted their house can tell you that ceilings are a lot harder than walls, especially when the ceiling in question covers 5,000 square feet and you have to lie on your back 20 meters above the floor to paint it. So what did Michelangelo do when planning to paint the Sistine Chapel? The same thing you should do when faced with a big task.

Michelangelo divided his mural into panels: separate, free-standing areas, each of which tells a story. But he did so fairly carefully, such that the panels exhibit these characteristics:

- High cohesion
- Low coupling
- Conceptual integrity

These are things we can learn from.

COHESION

What is cohesion? As used here, cohesion refers to the panel's focus and clarity of purpose. In the Sistine Chapel ceiling, each panel tells a single Old Testament story—completely, but without any extraneous elements.

In software, the Unix command line tool's philosophy of small, sharp tools ("do one thing and do it well") is one example. Each tool is narrowly focused on it's primary task. Low cohesion occurs when you have giant "manager" classes that try to do too many disparate things at once.

COUPLING

Coupling is related to orthogonality[HT00]: unrelated things should remain, well, unrelated. Following the object-oriented principle of encapsulation helps to prevent unintended coupling, but there are still other ways to fall into the coupling trap. Michelangelo's panels have low coupling; they are all self-contained; there are no instances of figures reaching from one panel into the next, for instance. Why is that important?

If you look closely at one of the panels that portrays angels gliding about the firmament of heaven, you'll notice that one of the angels is turning his back to, and gliding away from, the other angels. You'll also notice that said angel isn't wearing any pants. He's rather pointedly "mooning" the other angels.

There is surely a tale that explains the bare tail of the mooning angel, but for now let's assume that the Pope discovered the mooning angel and demanded that it be replaced. If the panels weren't independent, then the replacement of one panel would entail replacing some adjacent panels as well—and if you had to use different pigments because the originals weren't available, maybe you have to replace the *next* set of panels that were indirectly affected. Let the nightmare begin. But as it stands, the panels are independent, so the offending angel (who was apparently on Spring Break) could have been easily replaced with a less caustic image and the rest of the project would remain unaffected.

CONCEPTUAL INTEGRITY

But despite that independence, there is conceptual integrity—the style, the themes, the mood, tie it all together. In computer languages, Smalltalk has conceptual integrity, so does Ruby, so does C. C++ doesn't: it tries to be too many things at once, so you get an awkward marriage of concepts that don't really fit together well.

The trick then is to divide up your work while maintaining a holistic integrity; each Sistine Chapel panel is a separate piece of art, complete unto itself, but together they tell a coherent story.

For our projects, we have several techniques we need to use inside code, including modularity, decoupling, and orthogonality. At the project level, consider architecting the project as a collection of many small applications that work together. These interacting applications might simply use a network connection or even flat files, or a heavier-duty component technology such as Enterprise Java Beans (EJB).

TIME

Up until now, we've concentrated on splitting up a project in space, but there is another very import dimension that we need to touch on briefly—time. In the time dimension, you need to use iterations to split up a project.

Generally speaking, you don't want to go more than a few weeks without a genuine deliverable. Longer than that introduces too large of a feedback gap—you can't get the feedback quickly enough in to act on it. Iterations need to be short and regular in order to provide the most beneficial feedback.

The other important thing about iterations is that there is no such thing as 80% done. You can't get 80% pregnant—it's a boolean condition. We want to get to the position where we only ship what really works, and have the team agree on the meaning of words like "done". If a feature isn't done, save it for the next iteration. As the iterations are short, that's not too far off.

In time or space, feedback is critical. For individual pieces of code, it is vital to have competent unit tests that will provide that feedback. Beware of excuses such as "oh, that code's too complicated to test." If it's too complicated to test, then it logically follows that the code is too complicated to write! If the code seems to be too complicated to test, that's a warning sign that you have a poor design. Refactor the code in order to make it easy to test, and you'll not only improve the feedback loop (and the future extensibility and maintainability of the system), you'll improve the design of the system itself.

3 Satisfying the Sponsor

Now comes the hard part. So far, we've talked about problems that have simple, straightforward answers. Organize your system this way; always have good unit tests; look for and apply feedback to improve the code and the process; etc. But now we're headed into much more uncertain terrain—dealing with people. In particular, dealing with the sponsor: the person or persons who are paying to make this project happen. They have goals and expectations all their own, and probably do not understand the technology with which we create the work. They may not know exactly what they want, but they want the project to come out perfect in the end.

This must be the artist's worst nightmare. The person paying for the portrait is also sitting for it, and says simply "Make me Look Good". The fact that the sitter is royalty who commands a well-oiled guillotine doesn't help. Sounds pretty close to the position we find ourselves in as we write software, doesn't it?

Let's look at it from the sitter's point of view. You commission an artist to paint you. What do you get? Perhaps a traditional, if somewhat flat looking portrait such as da Vinci's *Portrait of Ginevra Benci* in 1474. Or maybe the realistic, haunting face of Vermeer's *Girl With a Pearl Earring*. How about the primitive (and topless) look of Matisse's *Seated Figure*, the wild and fractured *Portrait of Picasso* by Juan Gris, or the stick-figured jumble of Paul Klee's *Captive*?

All of these are portraits, all interpretations of a commonplace thing—a human face. All of which correctly implement the requirements, but all of which will *not* satisfy the client.

Beyond the Obvious

Each of these paintings captures the essence of a person, not just the form. More than simple photographs, each painting looks below the surface to capture something that the camera can't. As programmers, we must do the same thing, only we tend to call it *abstraction*.

The phrase "requirements gathering" implies that requirements are simply lying about, ready to be scooped up and worked on. That's akin to a simple photograph, in that it only examines the obvious, surface level elements. In order to emulate the painter, we need to go beyond what's asked for. We need to ask the wicked questions to help the client discover what's really needed.

Systems Thinking[Sen90] suggests asking a minimum of five "whys" beyond the first one. The classic example involves a factory floor where the consultant notices a small puddle of oil on the floor. He asks the shop manager about it, who grumbles and barks an order to the cleaning crew to get over here and clean up the oil. But the consultant persists: why is the oil there? The manager says it's the cleaning crew's fault. But where did the oil come from?

A little investigating and more than five "why" questions later, it turns out that an overly cost-conscious purchasing agent got a deal on cases of O-ring seals for the overhead pipes. Problem was, the rings were the wrong size—that's why they were such a deal. What seemed like a cost-savings was in fact costing quite a bit of money in various ways.

We once were approached to develop a complex, enterprise-level data processing system that mail room staff would use to coordinate, sort, and track incoming payment checks prior to their distribution to the correct department. The company's current manual procedure was error-prone and unreliable; checks were being lost or mis-routed to the destination department.

What's the real requirement here? A fancy system to sort and catalog mail for the sole purpose of delivering it to the right address? Hmm. Seems like there's already a system in place that handles that sort of thing. So instead of a nice, fat, year-long contract, we told the company to use a different postal address for each department. Let the Post Office do the sorting, hopefully without opening the pieces and losing the checks.

Requirements are rarely simple, and shouldn't be taken at face value. Remember, a portrait is more than just a picture.

Conventional Wisdom

Even stories about requirements may need deeper examination.

There's a marvelous story of technology and consultants gone wild, developing the Fisher Space Pen. The story goes that the U.S. Government spent millions of dollars of taxpayer's money developing a space pen—a pen that the astronauts could take to the moon that would operate in the harsh conditions of weightlessness, extreme heat and cold. Technology rushes to the rescue, and develops a miracle pen that can write upside down in a boiling toilet.

The Russians, by comparison, decided to use a pencil.

A marvelous tale of an inappropriate solution, except for one small problem. It's not true. Both the Russian and the U.S. astronauts used pencils at first, but there was a danger of the leads breaking and shorting out electric components, and the wood of the pencil itself was combustible as well. In a pure oxygen atmosphere, that's a really bad thing. The Fisher corporation realized this and, at its own cost, designed the Fisher Space Pen, which it then sold to NASA at reasonable cost. After the disastrous Apollo One fire, NASA made the Fisher pens mandatory.

Fisher listened to the real requirement, even before the client knew it. In time, NASA came to realize that they were right. It was an appropriate use of high-technology to solve a very real problem.

Technology For It's Own Sake

Of course, there's always the inappropriate solution: engineering for it's own sake. As luck would have it, we happen to have an anecdote for this case as well.

There was this company that had developed a sophisticated video camera that could pan and tilt, looking for a subject in its field of view. A wonderful, high-tech solution in search of a problem. In time, the company sold this technology to a government agency to help take pictures for driving licenses. You'd go into the licensing agency and have a seat in front of the machine, which would whir and click, grind and gyrate until it had locked onto your face. The flash would fire, and in a few minutes your completed driver's license would be ready.

One day, 58 year-old Fred complained that the pretty 20 year-old blonde girl on his license just didn't look much like him.

The company and the government agency kinda scratched their heads; they weren't sure what the problem was. Problems like Fred's were popping up over, but other then getting a bunch in a row, there didn't seem to be any pattern to it. Finally, the police started to complain—and got quite upset—when they started seeing driver's licenses that featured beaming, cartoon smiley faces instead of a photo.

They discovered that the technology had gone awry: in some cases, the camera wouldn't get a lock, and would simply continue to grind and whir, looking all over the room for the subject. After a few minutes of watching the camera carefully inspect the ceiling and windows, folks like Fred would get bored and wander off. The next driver

comes in, and with a flourish of clicks and whirs, the camera would snap their picture—and associate it with the previous driver's license.

Now the office staff figured out pretty quickly what the problem was, but they had no feedback path to the developers. They knew that once the machine got out of sync, they'd get bad licenses all day. So one clever user figured out that one could draw a happy face with marker on a piece of white paper, stick that over the chair, and the machine would happily snap the picture.

The real requirements were ignored in the rush to be clever, with predictably poor results.

How We Do It

So how do you find out what's in the client's head? At The Pragmatic Programmer's offices, we use "special equipment" (picture a 1950's mad scientist's laboratory replete with buzzing vacuum tubes, arcing Jacob's Ladders, and cranial implants). If that doesn't work, or if we're out in the field where health and safety restrictions prevent us from using our "special equipment", we resort to the old-fashioned method of asking questions, both of the client and of ourselves.

What is the users's level of sophistication? What is the context in which the software is used? Real-time on the factory floor? In a life-critical system? For a home grocery list? What is the lifetime of the application? Unused after next week, or do you need to worry about the year-2038 bug? What are the risks? Not just the development or technical risks, but what are the *sponsor's* risks in taking on this project?

The best way to get these questions answered, of course, is to always involve the users as you go along. Seek frequent feedback to make sure you hear stories about anyone making smiley faces as soon as it happens.[4] Maintain short iterations with frequent deliveries, and work with the real users directly as much as possible. User representatives (such as a supervisor, manager or director) generally aren't as representative as we'd all like to think.

In our perpetual rush to jump in and start coding to the first neat idea we come across, we run the risk of getting locked in to a half-baked idea too early. Instead, try to cultivate emergence: Allow the solution to find itself where you can. Part of a developer's job is to provide a fertile ground in which ideas can grow. This means having code that is agile: code that supports rapid reassembly so you can try things out. Code that is easy to refactor, or that uses flexible configuration and/or metadata to facilitate rapid—but reliable—change, bolstered by a reliable safety net of complete revision control and competent unit tests.

Does all of this really work?

Yes, it does. We've done it successfully, we know other people who've done it successfully. It's lot of work, and it's a lot of hard work, and despite our best intentions, it might still not be a success due to factors beyond our control. So why do we bother with it all?

Because, as Brooks said, we programmers create. We can create awe-inspiring works with little more than the exertion of the imagination. Why do we do it? We do it for the pleasure of watching them show it off to others, of watching them use in novel ways we'd never imagined. For the thrill of watching millions on millions of dollars in transactions flow through your application, confident in the results. For the joy of building and being part of a team, and for the satisfaction of knowing that you started with a blank canvas and produced a work of art. And if you've gone to all that trouble, we think you should "sign your work". You should be proud of it.

It is, after all, a work of art.

[4] Again, the longer the gap before you get feedback the higher the likelihood of getting feedback in the form of a subpoena.

References

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 2000.

[Bro95] Frederick P. Brooks, Jr. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, anniversary edition, 1995.

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison-Wesley, Reading, MA, 2000.

[Sen90] Peter Senge. *The Fifth Discipline: The Art and Practice of the Learning Organization*. Currency/Doubleday, New York, NY, 1990.

