Arlo Being Bloody Stupid

Conclusively demonstrating why you should listen to others…

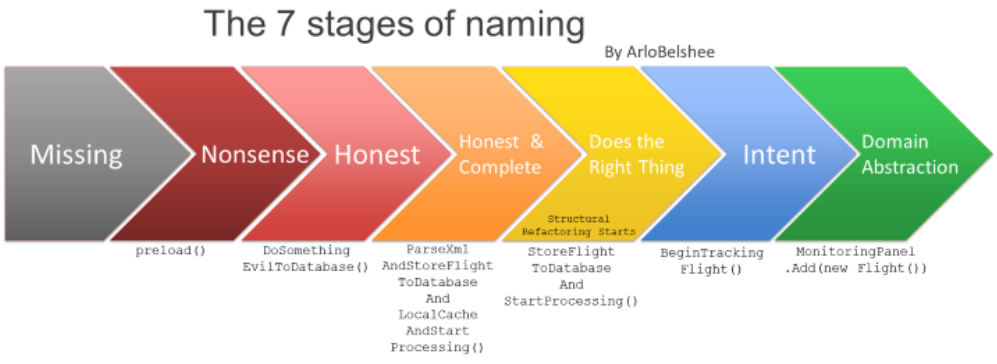
Good naming is a process, not a single step

August 21, 2015 by Arlo

Many people try to come up with a great name all at once. This is hard and rarely works well. The problem is that naming is design: it is picking the correct place for each thing and creating the right abstraction. Doing that perfectly the first time is unlikely. Let's talk about evolutionary naming.

> This post is the first in an 8-part blog series. See the bottom of this article for links to other parts and publishing dates for those parts that I haven't yet written.

The easiest approach I've yet found for finding good names is to progress along a series of regular steps. The steps a name goes through are:



(click for larger image)

1. Missing
2. Nonsense
3. Honest
4. Honest and Complete
5. Does the Right Thing

6. Intent
7. Domain Abstraction

At each step, I look at one part of the code, understand one kind of thing that is happening, have an insight, and write it down. I repeat this until I have moved one step down this list. I keep going until I have a good enough name for my purpose.

Working effectively with indebted code

Those of you who are reading ahead will wonder why I am focusing on names. I mean, names are annoying to come up with and perhaps this will make it easier, but is that really a problem?

The answer to that question lies at the heart of understanding, preventing, and paying off technical debt.

The source of all technical debt

> Indebted code is any code that is hard to scan. Technical debt is anything that increases the difficulty of reading code.

My focus on code reading may seem odd to many of you. After all, we're programmers. We're good at reading complex code, and our job is to update code. Shouldn't the definition of technical debt be something about the cost and risk of changing code?

Mine is. It turns out that the largest single thing developers spend time doing is reading code. More than design, more than writing code, more than scanning, even more than meetings (well, probably). According to an analysis I saw of Eclipse data, programmers spend around 60-70% of their entire programming time reading code.

So if we want to make programmers more efficient, we need to improve their ability to read code.

Furthermore, the probability of introducing a bug into new code is super-linear with respect to cyclomatic complexity. Make a method longer or more nested and people write more bugs when they edit it. Interestingly, it also correlates with syntactical legibility. Make the indentation inconsistent and programmers write a *lot* more bugs. More even than if you make the code more complex.

That's because bugs come from incomplete understanding. Incomplete understanding arises when the system is harder to understand than we can store in our heads at once. And the big thing that determines how much we can store is how legible it is. How quickly we can identify what it does and label things so that we don't have to remember and manipulate details.

So if our definition of technical debt is code that is difficult, expensive, or risky to change, then the root cause of that is code that is hard to scan.

And how do we make code easy to scan? Use good names to encapsulate details.

Working effectively with any code

Any code contains information. Some of this information it screams out at you. You can see this with a quick scan; you don't even have to read. Some of the information requires a careful reading to discover. Some is in between; it requires a read, but most any reader will discover the information once they look.

If we want to make code more scannable, we need to increase the percentage of relevant information that it screams at you. Which also means hiding the irrelevant information.

The process of reducing debt is simple:

1. Look at something.
2. Have an insight.
3. Write it down.
4. Check it in.

Look at something: pick one thing to examine. Don't try to understand everything. That will take way too long and will overflow the capacity of your brain. Yes, even your brain.

Have an insight: it doesn't matter what. Don't go for the perfect insight. Find something useful that is not yet scannably obvious.

Write it down: in a name. There are only a few places where you can write down anything you want in a programming language. Names are the best.

You can use a comment. But comments aren't actually part of the code. They duplicate the code, which causes all the usual duplication problems.

Other than comments, the only places to record arbitrary things are in names or in assertions.

If your insight is structural then it belongs in a name. If it is a runtime insight then use an assertion.

Assertions need to be easy to find. So don't litter them around your core code. Express your insight as an example and write it down in a test. And name the test about the insight (not about what code it happens to execute).

So even runtime insights are stored in names (test names). The specific example and measure are stored in the test body / assertion.

Insights belong in names.

Check it in: yes, right now. You have made the code better. Not perfect. But remember the mantra of legacy code: Good is too expensive; all I want is better (quickly). So now that you have made the world a little better, lock in your gains! Check it in.

> Good is too expensive; all I want is better (quickly).

The insight loop is all there is

This loop, BTW, is all of modern software development.

- Refactoring legacy code is running this loop and writing stuff down in names.
- Understanding legacy code is running this loop and writing stuff down as examples in tests.
- TDD is running this loop three times:
  - First a loop where we look at the customer interview and we write it down as one example in a test.
  - Second a loop where we look at the test and we write it down in names in the code.
  - Third a loop of refactoring the (new) legacy code.
- Design is a loop where the place you look is "how hard was it to write this test" and you write down insights by changing names (usually fixing the Does the Right Thing step).

Know where you are in this loop at all times and learn to do it quickly and you are a master-class software engineer.

The rub, of course, lies in knowing how to do it quickly. The masters are *really fast* at this loop. I can execute it

regularly in between 2 seconds and 45 seconds. I can do that speed no matter where I am looking for insights: customer interview, existing code, or existing test. And no matter where I need to write down my insight: a new name, change a name, change an abstraction, a new test, or a change in a test. I know other people who are even faster.

Which brings us back to names

Names are the place we record our insights. And we need a good way to know where to look next for the highest probability to find a useful insight. Interestingly, I find the quality of existing names to be a great way to find out where to look. And that leads us to the naming progression I described at the top of this article.

In summary, becoming a master at software development means iterating quickly. Using the core loop flexibly. Which means:

1. Look at something. **Use the quality of current names to decide where to look.**
2. Have an insight. **By reading things, usually names.**
3. Write it down. **In a name, using an automated refactoring. Or in an assertion in a test, using a fluent / easy to read assertion.**
4. Check it in. **Express your intent by naming your commit using a message.**

In one word: Names.

The rest of this blog series will describe the transitions a name goes through. Each will tell you where to look, what kinds of insights to glean, and how to write them down. I'll also give lots of tips on how to use tools to do this at speed. Finally, we'll close with some concluding comments and a short learning path—a sequence of the micro-skills involved that is designed to be easy to learn and pay off with advantages from the start.

The Naming is a Process blog series

1. Good naming is a process, not a single step (this entry)
2. Missing to Nonsense
3. Nonsense to Honest
4. Honest to Honest and Complete
5. Honest and Complete to Does the Right Thing
6. Does the Right Thing to Intent
7. Intent to Domain Abstraction
8. Summary and Learning Path (will publish Tuesday, 9/1/2015)

Tags: design, legacy code, naming, naming is a process, refactoring, tdd

Comments (4)

Comments (4)

Filip · *6 weeks ago* 0

Oh man, this is just what I was looking for. What you describe as "indebted code" is something that I've unintentionally add in a

lot of places throughout my project. Junior devs (such as myself) have a lot of trouble with this.

Not only do they not notice it, but they aren't even aware of how much naming is important. Academic literature and self-learning course never put any emphasis on this and it is something devs become aware of only by going through painful, old projects or by learning it from a mentor / senior developer.

Reply **1 reply** · *active 3 weeks ago* Report

abelshee 50p · *3 weeks ago* +1

I'm glad you are finding it useful. Awareness of technical debt and how to work with it is the difference between CS (the study of software) and software engineering (actually doing software creation). Very few courses provide any opportunities to learn software engineering, yet it is the only one which is useful in industry.

Reply Report

robid rabay · *9 weeks ago* 0

nice

Reply Report

Sarina Gesell · *32 weeks ago* 0

Thanks for sharing.

Reply Report

Post a new comment

Enter text right here!

Comment as a Guest, or login:

Name Email Website (optional)

Displayed next to your comments. *Not displayed publicly.* *If you have a website, link to it here.*

Subscribe to None ▼ Submit Comment

- **Categories**

  Categories Select Category ▼

- **Tags**

legacy code culture Simulator Agile refactoring measurement technical debt working tiny pair programming tdd architecture estimation learning no mocks naming naming is a process proficiency design example feedback

The last comments for
WET: When DRY Doesn't Apply

 @jaybazuzi

For reference, here's James' talk from AATC 2016:

The last comments for
Naming is a Process, part 5: Honest and Complete to Does the Right Thing

 EvilDBMethod

Hey, great article!
I really liked the practical approach to refactoring methods and using naming as...
» 3 weeks ago

The last comments for
Good naming is a process, not a single step

 abelshee 50p

I'm glad you are finding it useful. Awareness of technical debt and how to work with it is the ...
» 3 weeks ago

The last comments for
Llewellyn Falco - What makes a good test suite?

 kingroot apk new 14p

i thing granularity is the most importont among the 4 factors
» 4 weeks ago

The last comments for
Good naming is a process, not a single step

 Filip

Oh man, this is just what I was looking for. What you describe as “indebted code” is something...
» 6 weeks ago

The last comments for
Naming is a Process, Part 7: Intent to Domain Abstraction

 abelshee 50p

It will. But it is currently blocked on a software project. I mentioned the learning path? Well, I'm...
» 6 weeks ago

The last comments for
The Core 6 Refactorings

 abelshee 50p

Interesting. Very different context / direction. I'm not looking for best.

I'm looking...
» 6 weeks ago

The last comments for
Naming is a Process, Part 6: Does the Right Thing to Intent

 abelshee 50p

I hear you. I've heard others with the same request. I think it would make it better. And I'm...
» 6 weeks ago

Comments by IntenseDebate

Top 5 Posts

The No Mocks Book (33 comments)

Is Pair Programming for Me? (23 comments)

How I Learned to Stop Worrying and Love the Mock (20 comments)

Scaling Agile - the Easy Way (15 comments)

That's Not Agile (13 comments)

Comments by IntenseDebate