Arlo Being Bloody Stupid

Conclusively demonstrating why you should listen to others…

Naming is a Process, Part 7: Intent to Domain Abstraction

October 5, 2015 by Arlo

In part 1 we talked about naming as a process. We talked about how legacy code is really defined by its poor legibility, and that reading is the core of coding. And we talked about how working effectively with legacy code is simply the process of having an insight, writing it down, and checking it in.

Later parts have gotten us to a name that conveys Intent.

Now let's look deeply at the last transition in names, from Intent to Domain Abstraction.

Finally we are ready for the most important step in naming. This step is why all of evolutionary design really comes down to "name things well. Continuously." It is time to correct the domain abstractions we are using and adjust the names.

We currently have a set of names used together. Each one expresses Intent individually. Responsibilities are factored into the right places (each Does the Right Thing). But taken together, the names are a mishmash of ideas. Each expresses itself and its context well, but they don't create a shared context.

> A domain abstraction is just a shared context for some set of code.

At this point, the insights we seek are Whole Values. We want to find generative domain patterns. Concepts that, once they are in place, make the missing code as obvious as the code that is already written.

We won't necessarily write the missing code yet—we could be wrong about the domain abstraction we see—but we will leave the code in a state where it is obvious where and how to add this code if we need it.

The fundamental process we will do at this step is to find Primitive Obsession, have an insight about a missing Whole Value, and write it down. We will do this mostly mechanically as we want to be able to have insights in code where we don't yet see the unifying concept.

Finding Primitive Obsession mechanically

Where to look: sets of methods or classes that share some similarity.

Here are some of the common patterns I look for. Each identifies some primitive that I am obsessing over.

In sets of classes:

- Tight interdependencies with few connections outside the class.
- Several methods in one class that use public getters from another.
- Classes with similar names:
  - Thing / ThingIsValid,
  - Repeated prefixes or suffixes: -Transform, -Traversal, Tree-, Tail-.
- Classes with CSy or grab-bag names:
  - -Manager,
  - -Transform.
- Names that end with -er, indicating an action rather than a result / object.
- Feature envy: obsessing over the methods and properties of another object. Calling it all the time and generally being creepy.

In sets of methods:

- Multiple parameters that are passed together to multiple methods.
- Sets of methods that each use the same subset of an object's fields.
- Methods with similar names:
  - Begin /end / see progress / poll for data,
  - Create / read / update / destroy,
  - Source / sink / transform,
  - Names which contain near-synonyms.
- Chunks of methods which obsess over some variable or small set of variables for several lines (many operations on the same data).
- Methods which are often called together (especially common in error handling or creation logic).
- Common phrases in the name without a matching noun in the system.

Fields & parameters:

- Names that narrow what the thing is, because the variable's type name cannot (firstName, lastName, SocialSecurityNumber—all on parameters of type String).
- Names that narrow or interpret allowed values (rangeInMeters, hardwareModeFlags—both on fields of type int).

Each of these indicates that something is missing in your set of domain abstractions. Other code is coddling that missing abstraction.

Each of these also indicates code that is using an overly-general concept where something more explicit and single-purpose would do.

What to do about it

Every time we see one of the above, we know something is missing. Our goal is to figure out what is missing and create it. This comes in two steps. First we name the primitive, then we name the thing we should use instead.

Our insight: a primitive and the domain concept which would replace it.

We don't need to rename the primitive in the code. We're about to replace it. Instead we write it down on a notecard or whiteboard. State precisely which primitive we see and what obsessive behaviors we see the code

doing over it.

Now we want to write our insight down into code. We need to do the following:

1. Break the specific concept out of the general concept.
2. Examine each piece of coddling code.
3. Break the part that is related to the new concept out of its context.
4. Move the related part to the new concept.

Executing the change

> **What to write down:** one new Whole Value that matters to our domain + modify existing code to use it.

The first 3 steps use refactorings that we already used in earlier phases.

1. Break a concept up by moving from missing to nonsense, then climb the naming levels as desired.
    1. Method: use extract method.
    2. Class: use extract class.
    3. Parameter or field: use Introduce Parameter Object (the problem is the type).
2. Use find usages.
3. Breaking things apart again. See above.

Step 4 uses:

- Move Method / Class
- Convert to Instance Method
- Convert to Static
- Convert to Extension Method (if in C#)
- Rename

The most common form of rename is to intentionally combine code chunks.

When a primitive has been running around the system for a while, often multiple pieced of code wanted to do the same operation to that primitive. The operation was small (often a 1-line loop with an accumulator variable or a simple conditional). So they just duplicated code inline.

In step 3 we extracted these duplicate chunks into tiny methods. We gave each a good name based on its context (an intention-level name).

In 4 we:

1. Move them all to the new class.
2. Use non-overlapping names so that we ensure zero behavior change.
3. Sort together those with identical implementation.
4. Examine pairs with identical implementation to see if they also share intent (not all duplicates have the same intent, especially when we are adding a missing domain abstraction).
5. Any time we find a pair with same implementation and intent, we rename whichever has the less-revealing name to match the name for the other.
6. Then delete one, leaving the callers sharing code.
7. Repeat until each pair that shares implementation doesn’t share intent.

It is also common to find pairs that share intent but not implementation. These could be bugs (someone updated one use of the primitive but missed another). They could also be features (the intent is nuanced).

When you find two chunks with similar intent but different implementation:

1. Rename them to be very similar.
    - Use a common prefix followed by something that distinguishes the difference.
    - The difference will be implementation, not intent / domain-relevant. That is a smell but better than we had before.
    - We can later address that smell by going through the naming process again. Finish this pass first.
2. Write a pair of tests, one for each chunk. Describe both the common part (with a shared assertion method) and the difference (with two clear, separate asserts).
3. Optional: combine the two methods into one method with a discriminator variable.

The last step can make things either more or less clear. It can hide the code smell and reduce the probability of fixing, but it can also create a new concept (the discriminator variable itself) which can grow into a valid domain abstraction.

For details on how, see my gist Combine 2 methods into one with a discriminator variable (I'm combining two slightly different implementations of `Car.Drive()`).

- The history shows a complete set of refactorings (17 steps) to implement the re-design without ever violating safety.
- I have tests only to show what happens to call sites. I never relied on tests to verify any step.
- I would be equally comfortable (and safe) performing this on entirely untested code with thousands of call sites.

And that's it

We identified our domain abstraction and wrote it down. At this point our naming is done.

In the real world it probably took me a week or so to get through this whole process (at a total time cost of 30 minutes spread across that week). I find missing names one at a time, create them, and get them up to the honest level. After doing that 4-6 times in one area area I upgrade each of those names up to does the right thing (through complete on the way). After I do that 2-3 times the intent becomes clear and I upgrade the names.

And every so often I see primitives lying around or duplicate sequences of statements that operate on the same variables. When all the names around are intent-revealing, the missing Whole Value becomes obvious. It often requires 2-5 flurries of creating intent-based names before a whole area is intent-revealing.

I introduce a new domain abstraction. Actually, usually I introduce 3-5 of them in a single quick flurry. The code rapidly changes across the project to use the new abstraction in about 60% of the applicable cases. Then slow change happens over another week or two as the abstraction proves to be valuable. It attracts new operations and cleans up the other applicable cases.

Next up: a summary of the whole thing, with a learning path to learn this whole approach to legacy code in short steps that each provide value and can be mastered in an hour or so.

The Naming is a Process blog series

1. Good naming is a process, not a single step
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 (this entry)

8. Summary and Learning Path (will publish Tuesday, 9/1/2015)

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

Tweet

Comments (2)

Comments (2)

Login

Sort by: Date **Rating** Last Activity

Alex · *8 weeks ago* 0

Hi Arlo!

Nice post, thank you. Any chance that the last part will be published?

Greetings,
Alex

Reply **1 reply** · *active 6 weeks ago* Report

abelshee 50p · *6 weeks ago* +1

It will. But it is currently blocked on a software project. I mentioned the learning path? Well, I'm creating some simple tools for learning this as part of your daily work.

Reply Report

Post a new comment

Enter text right here!

Comment as a Guest, or login:

Name

Displayed next to your comments.

Email

Not displayed publicly.

Website (optional)

If you have a website, link to it here.

Subscribe to None

Submit Comment

- **Categories**

Categories Select Category

- **Tags**

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

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