Code Ownership

JimCoplien suggests ...

A Developer cannot keep up with a constantly changing base of implementation code. **Therefore:** Have each code module in the system owned by a single Developer.

- http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?CodeOwnership

Fans of WikiWikiWeb may notice that this space takes a contradictory position, as does Episodes and its PairProgrammingFacilities ...

- http://c2.com/ppr/episodes.html

One stumbling block to discussions about CodeOwnership is the variety of interpretations that reasonable people can apply to the idea of ownership over code. I've sat in a meeting where two engineers were pushing hard for "code ownership", but on examination one was found to mean "I want a single set of hands coordinating all changes" while the other meant "I want complete artistic control."

Depending on the size and maturity of an organization, and the processes and standards that teams follow within the organization, "code ownership" can cover:

- *What* the code is supposed to do (requirements)
- *How* is the code supposed to do it (design, coding standards)
- *Who* makes changes (coordination)
- *When* do changes get made (coordination, project planning, configuration management)

(Many of the startups I've been with have experienced growing pains as the scope of CodeOwnership has been reduced from all-of-the-above to how-and-who to who.)

Cope's solution from the OrgPatterns pattern cited above reads:

- "Each code module in the system is owned by a single Developer. Except in exceptional and explicit circumstances, code may be modified only by its owner."

This is a clear statement about *who*, but the other implications of ownership (particularly the *how* and *when*) are less clear (to me, at least) without further context. -- DaveSmith

---

Right, thanks Dave! In CodeOwnership, as stated, ownership is about editing the code; owners must know the details of all changes under their purview. The only way to do that is to have the owners actually make the changes. Of course, because they're editing the code, by extension they understand the design ramifications.

Other project members can certainly propose changes to support changes to their own code, but the owner needs to make the changes. A slightly weaker version of the pattern allows the owner to "sign off" on all changes instead of making them, but that ends up being a RubberStampRole.

The expectation is that, in most cases, the owner will make suggested changes. It is also the expectation that the architect will make all changes proposed by the programmers. The role of architect is another kind of ownership, at least in part.

CodeOwnership, I think, is best viewed as a mechanism rather than a policy. It is a useful mechanism. It flexibly supports many policies.

-- JimCoplien

I submit that code ownership is not a generally good thing. I have been working with a group of eight engineers on a fairly complex system. We have no such concept. In fact, we go the other way - no one works on the same part of the system for more than two ThreeWeekIterations, and generally not for more than one. The result is a team that can quickly and easily shift resources where they are needed, code that is communicated and refined, and a high TruckNumber.

Of course, I could be wrong. Or maybe this paragraph was really typed in by JimCoplien to make the point that, even here, there is an ownership culture implemented by convention.

-- KentBeck

I would suggest that this pattern works well in the right context. There are no absolute patterns, and I'm careful about making absolute statements ("X is a good thing" or "X is a bad thing"). Let's explore the context a bit. Help me out here, the rest of you.

I haven't studied this in enough depth to make the context precise. It is probably a function of things like this:

Market structure: If a single product serves multiple customers in a diverse market, you clearly need ownership (really, advocacy) on a per-customer basis. This is related to ArchitectureFollowsMarket. To maintain architectural integrity, you need ownership in several dimensions. One dimension is code; a code owner can represent the interests of architectural integrity as they resolve diverse market needs. You tend to get architectural cruft if you have feature ownership without code ownership (and it's even worse if you have neither).

Geographic Distribution: Some projects suffer geographically or politically distributed development as a casualty of unreasonable market and organizational forces. This leads to strong social issues of allegiance, which leads to issues of ownership at a very fundamental level (all the way back to Rousseau's LawDerivesFromProperty).

Scale: Ownership isn't an issue on an 8-person project where communication is almost free (there is a knee in the curve at about 7, and another knee in the curve at about 20, from our empirical organization studies). Some projects aren't easily accomplished with that many people. You need ownership on large projects - large in terms of the people involved. I work with projects from 3 to 3000 people.

Robustness: While all programmers on a small team can understand all the code to some degree, some programmers can understand other parts of the code better than others. If you have an urgent field problem that needs to be escalated, you want to be able to be able to bring in the right folks. That's usually the code owner and the feature owner. Sure, you can use Envy to figure out who put in the last mod, but without code ownership, you end up with a patchwork quilt that's difficult to unweave.

Bleeding-edge Technology: Some of our products build on leading-edge technology that is still well-understood only in a few peoples' heads. The worst disasters occur when you turn loose sorcerers' apprentices on code they don't understand. In a typical project, not everyone can know everything - except in some mature domains where there have been few business paradigm shifts in the past decade or two.

Reuse: Some companies do one-at-a-time, turnkey systems, built and maintained by SWAT teams. Other companies have dozens of related

products that build on a small collection of platforms. The platform frameworks can't bend to the whim of every programmer; they must be owned by someone who can negotiate between the different users. One reason behind this need is that the number of users involved is larger than it would be if a single project used the platform, and it's difficult to bring everyone together to negotiate the change. This is a special case of Robustness.

I know of some empirical studies using visualization techniques to find patterns in huge data sets, that looked at the issue of locality of change in some mature projects. I think they found that the long-term bug density correlates directly with how many different people had their hands in the corresponding piece of code. That's pretty hard data, and it's real. I'll see if they've published anything on this (other than my CodeOwnership pattern, that is). It would be interesting to see if they monitored any experimental parameters that explain the difference between Kent's opinion and the experience encoded in CodeOwnership.

If I were to guess, I'd say that the most powerful context element here is project size. But I'm cautious about making that explicit, because I've small teams differentiate themselves positively on the basis of code ownership.

By the way, we've found that too high a TruckNumber is just as disastrous - and far more common - than too low of a TruckNumber. It goes along with JoeDavison's UniversalProgrammer (or PlugCompatibleProgrammer or PlugCompatibleMoron or whatever name you choose for the pattern).

-- JimCoplien

CodeOwnership of this form (one owner) is problematic for small projects (and maybe large ones) that are under heavy schedule pressure, where the absence of a team member, even for a relatively short period, can hold up an integration. In these situations it helps to have a "buddy" (as in PairProgramming) who knows the code and can be trusted to make changes and inform the owner. We call this CodeStewardship, which reframes the role in an interesting way.

I've lived with CodeStewardship off and on for several years, and have rarely seen an owner devolve into a RubberStampRole. (Though I have seen RubberStamping when a team was trying to maintain the fiction of peer review when there were too few experts to go around.)

--DaveSmith

Size is one factor, but, as I mentioned above, expertise is another. As a counter to the small-teams trend you mention, I saw pretty strong code ownership on the Borland QPW effort. The reason? They had an inference engine expert doing the inference engine; a database expert doing the database; and HMI expert doing the human interface; etc. I believe it would have been a much less pleasing system had the project used PlugCompatibleInterchangeableEngineers. Each programmer knew the patterns of his or her domain, and owned the results.

It matters a lot less if you're doing another business system one more time with feeling.

-- JimCoplien

For the little difference it may make, what I advocate is that the team establish an "ownership model". That means that they come to an agreement as to who can change what - the key part being who gets to delete things. If they choose that only one person gets to alter a class, I suspect things will be a little less efficient, but it can be made to work. Most of the time, particularly in small and medium-sized projects, they agree that one person has final say-so over a range of classes, but others have editing rights, and they should either sign and date, or send the owner an email, so she/he can look over the changes. Occasionally, they decide to make full joint ownership, so that anyone can change anything (and that specifically means rewrite or delete). These projects also work.

The one that doesn't work is when there is not explicit agreement, and no explicit ownership model. The common failure mode here is that anyone can add code, but no one can delete code. That's when you get the SharedRefrigeratorSyndrome ("whose ugh leftovers are these?"). The other one that could be expected to fail, is if there is bad blood between developers and they have established full joint ownership. I have heard tell of such things happening on projects in which joint ownership was not officially established, but they just didn't have any ownership mechanism in place. In other words, they didn't have a clear ownership model. -- AlistairCockburn

---

IMO, CollectiveCodeOwnership is not equivalent to PlugCompatibleInterchangeableEngineers. In the real world, there will always be certain people on a project with more expertise in certain areas than the rest of the team. But with PairProgramming, the core knowledge of each area of expertise is continually disseminated throughout the team as time goes on, and the UnitTest suite makes sure that whatever is undertaken doesn't break anything. I think most of what is being discussed here has more to do with effective communication of design decisions than who actually "owns" the code. Referring back to the list of counter examples above, I think Scale and Geographic Distribution are the only two that some form of ownership actually has any bearing on success (I have to admit I really didn't get the Market Structure example, so I can't argue either way on that). However, I would also contend that within each sub-team or local group that CollectiveCodeOwnership be practiced. What this implies is that the project be split along non-overlapping segments, which I realize is not always possible, and for anything that reaches over to the other realm, a meeting of the minds does need to take place. What the CodeOwnership accomplishes, then, is setting constraints on a project. And these constraints only need to be set because effective communication becomes more and more difficult as teams grow or get spread around.

The robustness example almost provides its own counter argument. If an emergency comes up you **do** want the best and most experienced people for that problem. And that's just it, because the PlanningGame will help find those people, and PairProgramming will virtually guarantee you will have a pool to pick from instead of an elite one or two. For Bleeding Edge technology you never have "sorcerers' apprentices just turned loose" because you would pair them properly and also have a UnitTests to protect the innocent. Lastly, with the dissemination of knowledge mentioned above, RefactorMercilessly, and doing ExtremeProgrammingDesignReviews when necessary, reuse is utilized throughout the project. Without CollectiveCodeOwnership many of these practices, especially PairProgramming, UnitTests, and RefactorMercilessly, would be very difficult if not impossible.

In other words, whether you have CodeOwnership or CollectiveCodeOwnership, has little to do with the ability to solve the particular scenarios mentioned above. The solutions have more to do with communication across the team(s) and depending on your ownership model, the mechanisms are different for each. I happen to believe the collective approach provides the better solution. YMMV. -- TomKubit

---

CodeOwnership is an easy pattern to apply successfully in immature organizations: It's simple, easy to understand, and it effectively delegates both authority and responsibility at the same time.

But its major limitation is that it organizes responsibility by code instead of by business functions: So, if you practice OnceAndOnlyOnce or some form of code reuse, you'll probably find that many business functions you wish to implement may touch shared code or cross ownership boundaries. So, unless your team members have an unusual sense of social unity, you'll find that CodeOwnership makes your developers compromise the design to minimize the number of code owners who must cooperate to implement a given change.

-- JeffGrigg

Oh yeah. Been there seen that. Eventually the changes just don't happen or become too hard.

See also EgolessProgramming, ArtistsRights, XpTeamVsIndividualCodeOwnership PairProgramming

---

We allow almost anyone to make relatively minor changes to any code. But the responsible developer for the code has to be consulted or make major changes. Pair programming mixes up knowing the code with knowing the domain. It would be silly for someone who worked on part of the bsd tcp/ip stack and who isn't an expert in tcp/ip to reorganize the bsd stack. That anyone thinks this is a good idea is quite astonishing. -- Anybody

Taken to its logical conclusion then, code maintenance would be a bad idea. There are lots of reasons why someone not previously familiar with code will need to modify it or make additions. The key is the way in which he approaches the task (think UnitTests). -- WayneMack

On the other hand, *every* change is a minor change to someone with GreenBarAddiction.

CodeOwnership can refer to 2 very different ideas.

- Only the owner can change the file. (like a bank customer owning a safety-deposit box.)

vs.

- The owner is responsible for periodically checking to see if any changes have been made to the file, and if those changes are especially misguided, reverting them. (like a landlord owning an apartment).

-- DavidCary

CodeOwnership becomes a serious problem if a PrimaDonna gets to own something. No one else is allowed to touch the code (or perhaps even look at it), and the owner (when not busy throwing temper tantrums, or holding the project hostage by refusing to work on something that no one else is allowed to touch) holds up development for weeks or months while others wait for them to make necessary changes. When they finally do get fired, then the rest of the team has to spend months digging through their code, trying to understand it and fixing it, or else rewriting it from scratch. (That is, if they haven't already SetTheBozoBit and written code that entirely bypasses whatever the PrimaDonna was responsible for.)

Corollary 1: Prima Donnas love code ownership. It gives them a power trip, a bargaining chip, excuse for throwing tantrums, and a feeling of superiority over those who have to ask permission to change code or ask for changes. They love fingerpointing and playing the BlameGame.

Corollary 2: People who work well in teams usually don't care about code ownership. Without it, they have the freedom to help and have help, to learn from and mentor each other, and the security that other eyes may find (and fix) errors they overlooked, and that if anything needs to be changed while they're on vacation, someone else can handle it. They feel that if we succeed or fail as a team, then it doesn't matter who's responsible for which file. Once the fingerpointing starts, it's too late to succeed, so may as well skip the BlameGame and get to what went wrong and how we should do better next time.

CategoryCoding

Last edit November 7, 2014, See github about remodeling.