Theory-building and why employee churn is lethal to software companies

Baldur Bjarnason
9-11 minutes

(What follows is an extract from *Out of the Software Crisis*)

Programming as theory-building

> The building of the program is the same as the building of the theory of it by and in the team of programmers. During the program life a programmer team possessing its theory remains in active control of the program, and in particular retains control over all modifications. The death of a program happens when the programmer team possessing its theory is dissolved. A dead program may continue to be used for execution in a computer and to produce useful results. The actual state of death becomes visible when demands for modifications of the program cannot be intelligently answered. Revival of a program is the rebuilding of its theory by a new programmer team.

Programming as Theory Building (Peter Naur, 1985)

Software is a temporary garden whose fate is inextricably intertwined with its gardeners. Beyond that, software is a theory. It's a theory about a particular solution to a problem. Like the proverbial garden, it is composed of a microscopic ecosystem of artefacts, each of whom has to be treated like a living thing. The gardener develops a sense of how the parts connect and affect each other, what makes them thrive, what kills them off, and how you prompt them to grow. The software project and its programmers are an indivisible and organic entity that our industry treats like a toy model made of easily replaceable lego blocks. They believe a software project and its developers can be broken apart and reassembled without dying.

What keeps the software alive are the programmers who have an accurate mental model (*theory*) of how it is built and works. That mental model can only be learned by having worked on the project while it grew or by working alongside somebody who did, who can help you absorb the theory. Replace enough of the programmers, and their mental models become disconnected from the reality of the code, and the code dies. That dead code can only be replaced by new code that has been 'grown' by the current programmers.

A successful software project is grown from a small living thing to a larger living thing. Building the project large from the start and will never come to life. Replacing the gardeners that brought it to life will lead it to whither.

Most of our errors with software development are category errors. We treat these software projects as one kind of animal when they are a different species entirely. They are grown thought-stuff. We're treating them like lego blocks.

This is why the Stock that code provides to a software development system isn't the code itself but the value the code represents. That value depends on how well it is understood by the team that needs to fix, improve, and modify it. It depends on the stability of the underlying language and platform. It depends on how often it needs to be rewritten or modified to continue to function normally as its dependencies change. Code isn't valuable in and of itself. Code can be a liability. What you want is code that has value and whose value depreciates slowly.

Misunderstanding the nature of software development will increase project failures. Going against the grain of software development makes catastrophe more likely, much like how the wrong type of wood, steel, or concrete will lead a bridge to collapse.

Software is the insights of the development team made manifest. Software has no life on its own but exists as a kind of cyborg simultaneously in the programmers and the code. To reuse Donna Haraway's words, software is *simultaneously fiercely material and irreducibly imaginary* (p. xii, Gray 1995). Software can be real and unreal at the same time. The written code is how software interacts with end users and other software systems. The insights and knowledge that exist in the minds of the developers are how software lives, changes, and grows. Without the code, it has no way to interact with the real world. Without the knowledge in the minds of the developers, it has no way to adapt and survive.

Documentation only works up to a point because it can both get out of sync with what the code is doing and because documenting the internals of a complex piece of software is a rare skill. A skill that most developers don't possess. Most internal documentation only begins to make sense to a developer *after* they've developed an internal mental model of how it all hangs together. Most code documentation becomes useful *after* you have built the theory in your mind, not before. It operates as a mnemonic for what you already know, not as a tool for learning.

This has practical implications for software projects.

The first is that it's a partial explanation for *bitrot*, the phenomenon where working software seems to deteriorate even when the code is untouched. Sometimes bitrot is straightforward: the platform or dependencies has been updated. The code "rots" because its context has changed. But sometimes bitrot happens because the *programmers* have changed. Memory is fallible. The mental model that a programmer has integrated about a particularly convoluted piece of code can fade away. The coder returns to the code after a period, and it no longer makes sense. The programmer no longer has a cohesive, integrated theory about that part of the software which means it has to be replaced.

This can happen in the other direction as well. The programmer revisits a module after working on other parts of the software. That work has left them with a sharper, more accurate theory of the software. Now they realise the original module didn't fit in with the rest and, with their improved insight, see that if left untouched the old module would become a source of friction, bugs, and defects. The code didn't change, but the programmer did. *The code rotted because the programmer changed.*

Team structure and churn

Another consequence of the theory-building model has to do with team structure and churn. The most reliable method a programmer has for building an accurate 'theory' of a piece of software is to have been there when it was first written. That programmer will have the best understanding of how the various components interact, how best to change any given part, how to minimise bitrot, and which pieces of code are actually unused. This is the *first generation* programmer.

The *second-generation* programmer—the second best method for building an accurate theory of the code—is somebody who worked on the code with somebody who was there when it was first written. Whenever you encounter unfamiliar code or modules that don't make sense, you have a first-generation developer on hand to help you understand—help you develop a mental model of how it all fits together. Over time, the proportion of the code written during your tenure will grow until you have effectively become a *first generation* programmer.

Team stability is *vital* for software development. Each team needs to be composed of a majority of first-generation developers. Too many second-generation developers and the first generation gets overwhelmed, and work stalls. The work slows down either because too many core tasks are waiting on the first generation or because the second generation keeps having to rewrite components from scratch because they didn't understand the theory behind the original component. Too few second-generation developers and there is no renewal—each developer that leaves the team is a potential catastrophe.

Many teams in the industry constantly rewrite parts of their code. Not because they keep figuring out better ways of approaching the problem but because nobody on the team has an accurate mental model for how it works and how the code fits in with the rest. If you can't hold onto the original team, if you grow the team too quickly, you end up running to stay in place. Code keeps getting written without any improvements of substance to the software itself.

The various different software processes will do little to counter these issues on their own. This is a team management issue that can only be addressed by a manager who understands the fundamental nature of software development. There is no magic development process that will prevent these problems. It takes skill.

Churn is destructive

Constant churn in a software development team, both among the programmers and designers, is ***absolutely devastating***. It is the death knell for a software project. Makes deadlines meaningless. It turns software into a disposable, single-use product like a paper towel. *Anything* that increases team member churn threatens the very viability of the project and the software it's creating.