

Of course, in the real world, testing is still necessary (even the most perfect-looking chair can unexpectedly collapse under the wrong circumstances), but the Ruby mindset is all about removing the unnecessary. It's about finding joy in coding and eliminating drudgery wherever possible.

On one end, developers who relished the painstaking reinvention of the wheel with every project. On the other, those who strive to build a world where you can comfortably lean back, sip your coffee, and let the code do the heavy lifting.

The grim reality is that even the smoothest-running systems occasionally fall apart.

Keep everything documented, every line of code, every decision, every time someone opened a mysterious new tab in their browser and pretended it was for research.

In the end, software development is a mad, wonderful, infuriating rollercoaster of ambition and chaos.



That peculiar ecosystem where business logic collides with the chaotic nature of software development.

Now, imagine starting from scratch in this swirling vortex of change, where every product requirement is an unpredictable moving target. In a global economy, these targets have the attention span of a goldfish. No sooner have you coded one feature than it evolves into a monstrous hydra of bugs, each one more mind-bending than the last. And why? Because everyone is in love with features. Features! “More features,” they cry, waving their shiny new ideas around like swords, oblivious to the horror of a thousand unresolved bugs crawling behind them.

But, here’s the thing: robustness before bling-bling. It sounds simple, doesn’t it? Like saying “eat your vegetables before dessert.” In every software startup company, where everyone is feasting on dessert first and then wonder why they’ve got a stomach ache.

Now, some people test software like they’re artfully constructing a cathedral, each module a delicately placed stone. Others, however, test software like they’re bashing a pinata, hoping something good comes out. The wise ones, the true sages, design tests using page objects and templates, independent, modular, serene. These tests are the unsung heroes of the startup world, reacting to change like a Zen monk amidst chaos. Meanwhile, the poor software is being pulled in all directions: currencies, languages, layouts, and laws (oh, the laws!). Email templates! Pop-ups! Hovering windows of doom! Networks that

The inventor of Ruby, Yukihiro “Matz” Matsumoto, has a rather curious philosophy. He openly admits that his goal when creating Ruby was to be lazy. Yes, you heard that right, lazy. And not just ordinary laziness, but the kind of laziness that requires deep, deliberate thought. The idea is simple: work hard upfront so you never have to work hard again. Hide the complexity, make it disappear beneath layers of simplicity, and create a world where the programmer’s job is to merely enjoy the process of coding.

In the world of Ruby, the goal is to create code that is so clear, so flawless, that you never have to test anything. Well, that’s the dream, anyway. In reality, even Ruby developers can’t quite escape the need for testing, but the philosophy behind Ruby is still to make everything as smooth as possible. Why spend hours wrangling with convoluted syntax and arcane error messages when you can craft something elegant and intuitive? It’s as if Ruby developers have unlocked some cosmic secret: programming should be fun, not painful.

The believe in hiding the complexity of the system so that it seems like everything just works. It’s like finding out the wizard behind the curtain is actually a really chill guy who just wants to make life easier for you.

Ruby developers craft their code with the same sort of ambition that someone might have when trying to build the perfect chair. They don’t want it to just work; they want it to be so comfortable that you never think about the mechanics behind it. And why test a chair that’s so perfectly engineered that it couldn’t possibly wobble?

Re-invent the wheel. Re-invent the entire car, in fact. Why use someone else's code when you could painstakingly rewrite everything yourself, right down to the game logic and hardware interfaces?

The philosophy behind this madness was that if you, the developer, wrote every line from scratch, you'd know the codebase inside and out. No external dependencies, no mysterious third-party libraries, just pure, unadulterated programming joy, or possibly despair, depending on your point of view.

It's worth pausing to appreciate the sheer audacity of this process. Atari developers operated in a world where overnight compilation was considered normal, where code sharing was a suspicious activity, and where testing was a luxury you might get to indulge in once a day, if the typist didn't get a cramp.

And yet, somehow, they created legendary games that defined a generation. Perhaps there's something to be said for writing code by hand, one painstaking line at a time, with nothing but a stack of paper and a dream.

Next time your team mates complain about automated builds taking too long, remind them of the heroic souls at Atari. They'd laugh in the face of overnight compilation, and then quietly pass you a handwritten note telling you to leave them alone so they can get back to inventing everything from scratch.

If the Atari developers were akin to ancient monks, dutifully transcribing sacred code by hand, then Ruby developers are more like modern philosophers, pondering how to do less while achieving more.

work sometimes and fail at the worst moments. Every platform and device under the sun will fail at some time.

What you can do is try to avoid depending on other people's flaky solutions. Because the minute you do, they'll vanish. Sold. Shut down. Replaced by something new, shiny, expensive, and incompatible framework. Just like fashion trends, but with more sleepless nights and fewer runway models.

"Programming is rather thankless," a wise person once said. Possibly while staring at the smouldering remains of their once-glorious codebase. "One year later, it's obsolete. Replaced by something better. And probably doesn't even run anymore." Such is the circle of life in software.

Tests for everything. Unit tests, behavior-driven tests, tests for things you didn't even know existed. Manual testing? Pfft. That's for masochists. The real goal is to automate yourself out of a job, leaving the machines to do all the work. Let the code evolve, but make sure the tests evolve too. That way, even if your entire tech stack morphs into something unrecognizable, the tests will still work.

Then there's the temptation of the Wild West, where building a shiny, superficial product is seen as the fast track to investor dollars. Throw in some flashy user numbers, and boom, investment secured. But oh, how quickly that brittle, bug-riddled product starts to crack under the weight of actual users. The solution? Hire more developers! But testers? Who needs testers? You can get away with a few, surely. They'll be fine, right?

Then comes the real masterstroke: hire inexperienced young talent and give them extravagant titles like "Manager of

Something Important” or “Lead Whatever.” Promise them the moon, and push them hard. The result? A shiny product, held together by duct tape and hope, while the testers panic and release code blindly into the wild, praying it works. Spoiler: it won’t. But that’s the startup way!

And then, dear reader, there are the dynamics, the glorious interpersonal drama that unfolds in every corner of the office. Developers pitted against testers, testers against product managers, managers against reality. Internal employees stare suspiciously at the externals, seniors sneer at juniors, introverts hide from extroverts, and the human resources department somehow manages to make everyone equally miserable. It’s like a soap opera, but with more keyboards and fewer commercial breaks.

And just when you think it couldn’t get worse, there’s the software itself. The onboarding process becomes in a new company is a labyrinth, full of monsters called “technical debt” and “organizational complexity.” The product matures like a carton of milk left in the sun, and every new feature becomes a harrowing ordeal. The developers flee, the good ones first, leaving behind the burned-out husks of what was once a passionate team. Let the employees like choose their work environment, like working in the home they already live in, and maybe, just maybe, they’ll stick around long enough to build something sustainable. Don’t judge how much time someone spends in your office. True productivity often happens in the least likely of places, like when you’re supposed to be doing something else entirely.

Ah yes, performance! Let’s talk about Atari. At Atari, that bastion of early video game development, where software engineering was less of a discipline and more of an exercise in masochism.

Coding wasn’t done on a keyboard, oh no, that would be too easy. Instead, you, brave coder, would spend your day handwriting assembly code on paper. Yes, you heard that right. A long, glorious stack of handwritten code. Each day’s worth of programming resembled something more akin to a medieval scroll than a modern development process. When you were done scribbling your brilliant opus of code, it wasn’t whisked away to some fancy compiler on a sleek MacBook. No, it was handed over to a typist, a person whose sole job was to take your notes and type them into a machine overnight.

In the morning, you would receive a beautiful artifact: a chip. This chip contained your code, freshly compiled and ready to be tested. You’d plug it into a game console, cross your fingers, and hope it didn’t catch fire (it probably didn’t, but you never know). Then, having learned from the results of your overnight compilation ritual, you’d start the whole process over again, handwriting more code, passing it off to the typist, and praying to the silicon gods for mercy.

As if that weren’t quirky enough, there was no such thing as “code reuse” at Atari. You see, senior developers, those grizzled veterans of assembly code, kept their stacks of handwritten brilliance locked away in personal cupboards, like some kind of digital treasure hoard. They didn’t share their code with new hires, oh no. Every new developer was expected to start from scratch.