5 Common Problems with Rails Projects

By Eric Davis | www.LittleStreamSoftware.com

TABLE OF CONTENTS

TABLE OF CONTENTS

INTRODUCTION

INTRODUCTION

Over the years I've reviewed dozens of Ruby and Ruby on Rails applications. After a while doing anything like this, you start to see the same problems creep up over and over again. In this document I'll explain five of the most common and risk-causing problems in Rails projects so you can watch out for them in your own projects.

1. BAD TESTS

1. BAD TESTS

The most common problem I've seen is bad tests. These might be poorly written tests, brittle tests, or even an outright lack of tests. Just about everybody apologizes for their tests like they have a guilty conscience.

The sad thing is, Rails comes with some amazing support for testing built right in. In a client project I did in Cake PHP (a Rails-like PHP framework), it took me dozens of hours to setup the infrastructure to support testing that is standard with Rails. Rails gives developers the ability and power to test the application, so it makes sense to use this power.

Regarding missing tests specifically, adding them to an existing project easily takes twice or more work. Sometimes due to how the code was written, it's not worth even trying to go and back-fill old tests. On the other hand, adding tests while code is written takes much less time and may even make development with tests faster than the development alone.

2. COMPLEXITY INSIDE CODE

2. COMPLEXITY INSIDE CODE

Complexity kills projects, but not in a direct way. The symptoms it causes can be the death-knell for a project:

> slowing development
> hard or impossible to reproduce bugs
> wild estimates off by 2x, 3x and maybe even 10x

Code complexity is pretty easy to comprehend. It's a concept that explains how difficult a piece of code is to understand. Addition is simple, a 900 line method with 42 conditionals and 15 return statements is hard (even if you are a developer).

Since complexity is a concept, it's also very easy to measure. Expert developers can even judge complexity just by glancing at some code. This means that reducing complexity is an activity that can be planned and performed very easily.

3. OVER-ENGINEERING (PROJECT-WIDE COMPLEXITY)

3. OVER-ENGINEERING (PROJECT-WIDE …

Taking a wider view than code complexity, project complexity is another common problem.

By calling it another name, however, we can get a much better emotional response: over-engineering.

Over-engineering basically means designing and building a Rails application that provides a solution in a too-complex fashion.

> If it's raining outside, it's much easier to put on a hat than to invent a weather control machine.

Every developer wants to avoid over-engineering, but many don't succeed. Over-engineering can be difficult to spot, especially by the developers who are actively working on a project. It's much, much easier for an outside developer or

3. OVER-ENGINEERING (PROJECT-WIDE …

consultant to spot it. That said, active developers can learn to spot over-engineering with training and practice.

The big downsides to over-engineering are straightforward:

> Development cost more
> Development took longer
> Additional technology risks

The main problem is that *more* was built and sometimes to support that *more*, riskier pieces of technology have be used.

4. DOCUMENTATION

4. DOCUMENTATION

Documentation, or the lack of it, is another common problem with Rails projects.

There are usually two extremes:

1. No documentation at all
2. Excessive and/or outdated documentation

The first extreme, no documentation, is usually a sign of a team that is moving too fast or aren't looking back enough (e.g re-viewing).

The second extreme is usually a sign of a team that is forced, coerced, or threatened to create documentation - or they are fearful of what could happen in the future.

Whatever the case, problems with documentation affect the team both in the present and in the future.

4. DOCUMENTATION

In the present they might not understand parts of the application and will make incorrect assumptions about how parts of it work. This leads to bugs, integration problems and communication issues.

Later on, documentation has an even greater cost. Without documentation, assumptions tend to be larger and I've even seen this leading to arguments (e.g. one developer argues it works one way versus another).

Documentation also affects any new team member. They either need to be training 1-on-1 (lack of documentation) which has a huge productivity hit to the project, or they learn the wrong things and waste time wading through the documentation (excessive/outdated documentation).

5. BEING DIFFERENT

5. BEING DIFFERENT

The fifth problem that occurs in Rails projects is trying to be too different. This encompasses other problems, including ones that aren't listed here.

What being different means is that the Rails application or team is trying to do things differently than everyone else for some reason.

In moderation and for specific, focused areas, this is actually a benefit. When dealing with your application's competitive advantage in the marketplace, being different is warranted.

Unfocused difference, especially in non-visible or non-core areas, is actually harmful.

One huge benefit of Rails is that the framework makes many decisions for you so you don't have to make them. This convention also makes it easier for other developers to understand how the application works.

5. BEING DIFFERENT

But being different changes this. Instead of the regular "it works this way", it's now "it works this other way because...". That "because" can kill the project.

One of the worst ways I've seen this was when a team changed so much in Rails that they practically re-wrote Rails itself in their application. At that point they would have been better off not using Rails or admitting that they were addicted to **being different**.

> Be different only where it really matters ”

SOLUTIONS

You've read about the problems and might have even recognized a few in your own Rails project. I'm going to take time now to propose some solutions.

Every problem has many different possible solutions, your case included. Think of these as recommendations or ideas to get you started.

Bad Tests

The best solution to bad tests is to:

1. perform a test audit
2. decide on the top three problems, and then
3. work on them

This might entail adding more tests, deleting old tests, rewriting confusing tests, or some combination of these.

Focus on the most valuable and highest-level tests first.

Complexity inside code

Fixing complexity inside of the code is a simple process. There are tools and services such as my Healthy Rails service that can be used to review the code and flag complex areas. Combine that with strong code reviews and you can fix highest complexity before it becomes a problem.

Over-engineering

To prevent over-engineering all you need to do is to ask the question:

> Is there a simpler way we can do this for now?

Once over-engineering has taken hold, it can be more difficult to remove. Depending on the depth of the problem, it might require replacing components, rewriting entire sections, or even starting something from scratch.

Sometimes you can't remove over-engineering at all. When that happens, just try to prevent any more over-engineering and keep that section isolated from the rest of your application.

Documentation

Depending on the problem with your documentation you'll either need to write more or write less. Treat documentation like code that has to be maintained, and perform regular documen-

tation reviews to delete old documentation and update outdated docs.

Being different

Being different is a hard problem to solve because you don't want to stifle creatively at the same time. Questioning decisions and trying to make the distinction between "perfect" and "good enough" has worked well.

YOU CAN SOLVE ANY PROBLEM

YOU CAN SOLVE ANY PROBLEM

I'd like to leave you with one final idea.

> No matter the problem, you have the power to solve it

Software is very malleable and with the right resources, attitude, knowledge, and people you can solve any problem with it. If you are getting stuck, you might be missing one of those components.

Don't be afraid to ask for help either. Getting help for one week might unstick you enough to save you months of work.

For a more comprehensive review of your Ruby on Rails application, Healthy Rails can guide you and give you the assurance you need to find and fix problems before they grow out-of-control.