SCOUT CORPS

Introduction

This case study covers a software organization with 5-20 engineers and covers a period of 1-2 years. (Specifics are deliberately vague, and based on experiences with multiple clients, to protect privacy)

At the beginning of that period, there was an existential question from investors over whether they should even continue to invest in this line of business, or whether it was a lost cause. By the end of that period, the company had revamped its product, shattered its internal and external goals, and underwent a successful exit.

My assertion – which previous clients will back up – is not simply that I'm a "10x'er", it's that I will help your team produce valuable software at 10x the rate they had before.

Overview

Investigation Phase (2-4 weeks)

Take stock of the situation. Talk to the team, familiarize with existing processes, sit in on meetings, review designs and documentation, review code, set up a development environment

Pick a single topic or pain point to focus on, treat that as a microcosm from which to draw larger lessons

End of Investigation Phase

Actionable plan to address the specific focus topic chosen above

Report on general impressions and initial recommendations; some immediately actionable, others where the action is "gather more information"

Ongoing Phase

Take on specific pain point directly by writing code, designs, architecture, policies & procedures, or whatever else is necessary

Embed with team, use that experience to come up with lessons and broader recommendations

Refine report recommendations; start making process changes

Common Theme

I prefer getting my hands dirty and doing the work that anyone else on the team would do, but with an overarching mission of making the team, organization, and processes better.

In my view, these always go hand in hand. Doing the dirty work without the high-level view only moves the needle a little bit; while the high-level can become cookie-cutter platitudes unless accompanied and informed by the dirty work.

SCOUT CORPS

Investigation Findings

In abstract, we're looking at projects going beyond deadline or spiraling for months with nothing to show. This boils down to a culture of perfect-better-than-done, a stream-of-consciousness prioritization mindset, and a cycle of scope creep – when a delivery date is pushed back, the motivation to squeeze something unrelated in "before it's too late" increases, pushing the delivery date back farther…

Problems Identified

1. Project goals are poorly defined.
2. Project plans are nonexistent.
3. No clear responsibility for QA. This isn't a critical priority because few bugs are being noticed by clients. But that is, in turn, because few clients *use* the product. This will become important as sales improve.
4. One more, specific to the technologies used.
5. One more, specific to the team and personalities.

Recommendations

1. You need two levels of project documentation:
   a. A document intended to be shared and discussed with the customer-facing part of the business (sales, support, account management) and the tech team (design, product, user experience).
   b. A document intended to be shared and discussed between the design team and the engineering team.

   These might be referred to as a "design document" and a "technical document" or by other names. In Agile/Scrum, the second role is usually played by "grooming" (reaching a consensus on project difficulty requires shared assumptions about how it will be implemented).

   I'm a fan of Agile, but in my experience more teams go through the motions of doing Agile "by the book" than get to the meat of it – which is ironically contrary to the first sentence of "the book".

   ***Individuals and interactions** over processes and tools*
   ***Working software** over comprehensive documentation*
   ***Customer collaboration** over contract negotiation*
   ***Responding to change** over following a plan*

   It's less important that you go through the motions of some idealized "agile" or "waterfall" than it is that you make sure the right conversations happen to build the best software and iterate on it.

1. You need two levels of project documentation: (continued)

   ### So, let's talk goals:

   A design document includes

   - "User stories" – what a user wants to do with the feature
   - specifics of what it's supposed to accomplish
   - along with mockups or whiteboards of the user interface

   What's important is that the design document is **a shared understanding between the technology and business organization** of **what** they're going to build and **why** that can be understood by both. This is necessary to appropriately prioritize the **when**.

   The design document should not attempt to proscribe technical implementation details or delve any more into the "how" than necessary.

   The technical document describes the **how**.

   The goals for the engineers are to validate the approach, to catch systemic issues before it's built, and to prevent huge surprises at code review.

   It also serves to expose the limitations the software may have, as it's adapted in the future – the use cases it can and can't easily handle. The technical document also fulfills the important role of creating **a shared understanding between the engineers and the product organization of exactly what sorts of enhancements the software will be suitable as a platform upon which to build.**

   This isn't strictly necessary for trivially small or straightforward projects – the goal is to reach a shared understanding; boilerplate documentation for its own sake is doesn't contribute to understanding, and indeed may even detract from it.

2. Create a database of projects and bugs.
   a. Preferably, use a system like JIRA or Asana. But remember, the tools aren't the goal: a small team might use whiteboards and sticky notes more effectively.
   b. Prioritization is crucial. If everything is high priority nothing is high priority.
   c. Understanding whether an issue is *caused by* a pending code change is also important. Don't hold new functionality hostage to fixing years-old bugs.

   The goals here are to make sure nothing falls through the cracks, while also ensuring that the team can focus on their jobs without having to drop what they're doing several times per day.

3. Improve QA
   a. Hire a dedicated QA team. You want to be starting QA *before* your new clients are *using* the software, but you can *demo* products that have not been fully QA tested.
   b. You need a good set of test data readily available. This must be created by the product / design / user experience team in conjunction with the account management team, it must reflect realistic use cases, and it should be kept up to date as new functionality is added.

Other Notes

These are things that weren't issues and didn't prompt immediate recommendations for this case study but are important and common enough that they should be acknowledged.

- Evaluation of team talent and morale

  Lack of talent was not an issue for this case study. There's no nice way to put it, but the software profession is plagued by charlatans – there are absolutely people with 10+ years of experience that can't – and never could – perform their basic job duties. Generally, I recommend giving those people support and training if possible or transitioning them to less demanding roles if not.

  Just as often as a lack of talent, though, is finding talented people stuck in roles that aren't suited to them. Someone with a mind for design stuck answering the phones in customer support, or a talented engineer being micromanaged by a less talented "senior" engineer. Identifying these people and lifting them up can bring a lot of bang for a little buck.

- How easy is it to "do a build"?
  - Set up a developer environment
  - Build to a QA environment

  This wasn't an issue for this case study, which makes it a bit of an outlier.

  *Most companies at this stage have trouble building a development or QA environment. They know they **should** be doing continuous integration and indeed do some form of it, but it doesn't accomplish what it needs to – for example: a test environment with no data, a continuous integration pipeline that doesn't cover database migration scripts, or new engineers needing to learn dozens of command-line steps that aren't scripted or documented anywhere. If you frequently hear "I dunno, works fine on my machine," you probably have a DevOps problem.*

- Technical debt in the code base: moderate to low, with some standouts that should be addressed

  Breaking technical debt down into specific issues, with an objective evaluation of which ones will materially hinder future productivity in the future and by how much is important.

  Categorizing can help break the stalemate between engineering and other departments that often devolves into "whether or not" technical debt is a real thing that needs to be addressed – it is! But "technical debt" often means "code I don't like" – having back-of-the-envelope ideas of the cost of dealing with versus not dealing with each bit of it makes it a conversation all stakeholders can participate in.

  *Some examples: Reliance on a language or toolset that's no longer supported by its authors can quickly become a **blocker**, design patterns that lead to frequent copy & paste can become a **nuisance**, and code that's just sloppy might make the next iteration of that functionality more difficult but typically **aren't worth fixing** for their own sake. Rewriting code that doesn't conform to your style preferences is often **busywork** unless there's an independent reason to heavily modify it.*

  Using a complex alphabet soup of specialized technologies where a simple arrangement of generalized ones would do just fine can also be a form of technical debt!

Specific Pain Points

During the investigation phase, I drew up design and technical documents for two specific projects that the team was struggling with.

Ongoing Phase

Specific Pain Points

I addressed one of these directly, writing the code from my own technical documentation, and then used that successful project as an example of how to improve processes.

Continued Engineering Contributions on Large Initiatives

Two examples of how a project might get stuck, and how I've gotten them unstuck.

Change Tracking

The mission here was to "help a user understand what data changed in the system, who did it, why, and when". Because there wasn't a complete design document, this got reinterpreted as "I want to be able to go back in time and see what it looked like on a given date in the past".

These aren't, on their surface, very different – but to an engineering team that lacked good and specific communication with other teams – this difference caused a 6-day successful project to become a 6-month failed project.

The engineering team had been trying to integrate their database change tracking with their code repository – such that, in addition to seeing what data you had X days in the past, you saw what it looked like with whatever calculations, formatting, and styles were present at that time.

The engineers lacked the understanding that this (exponentially harder) project they'd given themselves wasn't one anyone wanted, and the business side lacked the tools to clarify that this wasn't what they were asking for. The design document *forced* the conversation that they weren't previously having.

Graph Cycles

This one is more technical – a team was given the task of ensuring that a data set didn't contain a circular reference (A belongs to B; B belongs to C; C belongs to A). They determined that this was impossible because of the Halting Problem, a famously (proven) impossible computer science problem.

The Halting Problem, however, covers computer programs. A formula or dependency chain may resemble a computer program, but it's in fact more akin to a directed graph. Applying a graph cycle detection algorithm solved the problem.

There was a larger lesson to be drawn from this, though …

Culture Changes

Dealing with egos is frequently the largest challenge.

One way this often manifests is an engineer pushing a breaking code change with no regard for how it will affect code owned by other employees, then insisting that they need to fix "their" bugs. Alternately, an engineer could declare that code that they don't like will be destructive and block the change until it's exactly as they would have written it, thus making themselves a bottleneck.

From the outside, those two conflicts would look identical. Getting into the specifics and getting to the bottom of it is crucial to productivity and the ability to train and keep the right employees, not to mention maintaining a healthy and inclusive workplace!

Hiring

I successfully helped hire and onboard several engineers and a new CTO. It's worth noting that a drastic expansion of the technology team was considered, but never necessary, as the output of the existing team grew exponentially.

Cyber-Security Reviews, Training, etc.

I'll frequently manage the process of going through external reviews – either RFPs from clients or third-party audits – and doing trainings for non-technical employees on their responsibilities. I don't *do* the SOC2 audit, but I help get you through it.

What I Ask of Clients

Strategy and Prioritization

A common saying that I believe in: "what you decide *not* to do is as important as what you decide to do"

Clear direction on what you want the product to be and what problem you want to solve sets the tone for the entire team. It's a fine line to walk – key accounts can have valuable insight, and some pivoting and market-fit finding is natural, but a project pipeline that's exclusively sales-focused will turn you into a consulting company.

To reach the next level, you need a product, not just a collection of features.

Write it down

If a meeting doesn't have any notes, decisions, or written action items, did the meeting happen?

Written records are crucial to scaling up, delegating, tracking goals and progress toward them, keeping yourself honest, and not repeating intellectual work – and they're only becoming more important as remote work gets more popular.

Especially when it comes to technical details, it's easy to speak past each other and not realize it. While strategic ambiguity can be an asset in sales and diplomacy, it never is in product development – if you're delegating a decision or haven't yet made one, it's better to be clear that that work is yet to be done – and ultimately, the code must be instructions specific enough for a computer to follow.

Don't view regular interactions with your own team as a negotiation

For example, when someone non-technical reports a bug, and an engineer asks for more information to reproduce it, give them the benefit of assuming that they *need* that information to diagnose and solve the problem. Don't assume that they're just kicking it over the fence.

For another, reference the previous passages about documents forcing a conversation and shared understanding. With or without those documents, when engineers explain why a project is languishing, that *could be* an opportunity to reach that understanding – hear them out and you might find that they're trying to solve the wrong problem, rather than "just making excuses."