Leaner Programmer Anarachy

by

Fred George fredgeorge@acm.org

Antonio Terreno antonio.terreno@forward.co.uk

Abstract

At Forward Internet Group in London, we are practicing a post-Agile style that follows few of the standard Agile practices. Yet the style conforms to the Agile Manifesto [Beck 2001] and the XP Values [Beck 2005] ascribed by Kent Beck. We call this style *Programmer* Anarchy. This style is not unique with us; indeed, we know other firms (including financial trading firms) who follow this style, as well as the Facebook development team [North 2011].

Much as the movement from waterfall processes to Agile has shifted roles and titles, we have seen additional shifts in roles and titles when moving from Agile to Programmer Anarachy. Similarly in this shift, we have seen benefits in even faster delivery to market, improved team morale, and market-correct functionality.

This paper characterizes the new environment of Programmer Anarchy, examines the significant Agile practices that have been rendered moot with Programmer Anarchy, and closes with a discussion of the aspects at Forward that have enabled Programmer Anarachy.

Evolving Trust Relationships

In the beginning…

Traditional (waterfall) processes are characterized by requirements specifications by customers being passed to developers to implement. In this *contract* between the parties, the roles and responsibilities are clear. The waterfall processes allow for separation in time and location between the parties; indeed, the oft-cited benefit of complete specification is to be able to outsource the implementation to lower cost labor. Failure rates are high, and are little helped by process tools. Rather, the tools in the last twenty years have only helped to properly assess blame for the failures, rather than preventing failures. Such is the purpose of contracts in general.

Trust between the parties is not assumed. Indeed, we would argue that trust has diminished over time, partly due to outsourcing arrangements, but at least equally due to the increased complexity of the applications under development and the increased ecomonic pressure to delivery quicker in a rapidly evolving marketplace.

Agile effect on trust

Agile seeks to establish a more equal partnership between the customer and the developer. Conversations between the parties are encouraged (whether Scrum's sprints, daily stand-up meetings, or XP's on-site customer practice). As consultants we have seen trust re-established between stakeholders and development during the first Agile projects in an establishment.

But the renewed trust has not come easy. "Trust me," has been said too many times with too few successes. Skepticism reigns. We have found that the skepticism is usually overcome by great desperation, desperation usually brought on by the combination of a critical business need and a failing traditional plan. However, once established, the organization delights in the new trust relationship, and the trust grows steadily from delivery to delivery. The transition is accompanied by several ancillary traits:

- The need for detailed specifications diminishes to the point of vanishing.
- Jobs become roles, with most individuals able to play several roles.
- The importance of job titles wanes.
- Teams in place solely to coordinate (*project offices*, for example) become redundant.

These factors among others create a social cost to the organization to transition from traditional to Agile methods. It is not a smooth, continuous process from traditional to Agile, but rather more of a revolution, a discontinuity; jobs disappear, titles become less important, power shifts. Mark Durrand of uSwitch first labeled this gap or cultural chasm to be overcome when an organization shifts from tradtional methods and Agile methods [Durrand 2011].

On to Anarchy

A shift to Programmer Anarchy creates yet another cultural chasm to the organization. Once again, we have seen roles redefined and certain jobs disappear. And as for trust, even more is required between the stakeholder and the developer.

Characterizing Programmer Anarchy

Programmer Anarchy exhibits three distinct differences from Agile methods, but also shares on key characteristic.

Reduction in roles

The term Programmer Anarchy derives from the reduction in roles. Gone are the roles of Business Analyst, Quality Assurance (or Tester), and Project Manager. The only remaining roles are Stakeholder (or Customer, which I will use interchangeably) and Developer. In our environment at Forward, there are not even Managers of Programmers.

So how can such a group function? Through the classic concept of *anarchy*, defined as the simple absence of publicly recognized government or enforced political authority We want to let the programmers as a social group organizing and re-organizing itself as they see fit.

The missing roles become unnecessary for different reasons.

Business Analysts serve to understand the needs of the Customer, and to translate those needs for consumption by the Programmers. In Agile, the unit of consumption is a *story* [Beck 2005]. We would contend that from a perspective of value, no business value is created by transferring knowledge into the head of a Business Analyst, who then must convey that knowledge into the head of a Programmer. There is potential for information lost in the double-exchange. At a minimum, time is lost in this double-exchange. And all this is compounded if a Developer has a question for the Stakeholder that has not been anticipated by the Business Analyst. In our experiences as consultants, I cannot recall an engagement

where, as a Developer, we have not become one of the top Business Analysts for an enterprise simply because we have implemented a complex portion of the business for the Stakeholder. So at Forward, we have never had the middleman of a Business Analyst between the Stakeholder and the Developer.

Nor has Forward ever employed anyone in Quality Assurance. First, having someone dedicated to that role creates yet another person that needs knowledge transfer in order to be effective. Second, the architectures of our systems have generally rendered QA moot, with the pieces of the system being rediculously simple, and the overall system being sophistocated enough to require architectural skills to comprehend (no sweet spot for traditionally trained QA role). Third, we have shifted to active monitoring of our systems, an activity very akin to pre-deployment acceptance tests. With the myriad of external services on which our systems use, monitoring provides the needed robustness with the inevitable failures (whether ours or an external service), and is a much more valuable asset to create than acceptance tests.

The demise of Project Managers arises out of answering the following questions of ourself: Among the roles of Stakeholder, Manager, and Developer, who is best qualified to:

- Decide how much effort is required on a certain project? We answer: Developers.
- Determine the best team to assign to a project? We answer: Developers.
- Decide when a Developer can afford to role off a project? We answer: Developers.

- Decide the priority among projects in the face of scarce resource or time? We answer: Stakeholder.
- Guide a Developer in career growth? We answer: Other Developers.

As you can see, we don't answer "Manager" for any of these questions.

Missing Agile practices

As we have discussed Programmer Anarchy with colleagues and groups, the most astounding reactions occur when we list the Agile best practices we don't follow:

- Stand-ups
- Story narratives
- Retrospectives
- Estimates (of stories)
- Iterations
- Mandatory pairing
- Acceptance tests
- Unit tests
- Refactoring
- Patterns
- Continuous integration

As you can see, the list is long and full of accepted, "required" practices. Let's discuss the reasons that these have fallen away.

Our developers trust one another, and have mutual respect for the opinions of each other. Developers working on the same project sit in close proximity to each other (the same table, generally as close as possible). *Stand-ups* to check on the status are unnecessary; everyone knows what is going on. *Story narratives* are unnecessary as developers will talk about what is needed continuously, without the burden of writing it down in detail, or worse, having someone approve it. Trust and mutual respect eliminate the need for *retrospectives*; they feel comfortable questioning process and adopting/discarding ideas freely. Trust and mutual respect also renders *mandatory pairing* unnecessary; a Developer is comfortable with his colleagues to ask for help when needed, or to assist his colleagues if they are stuck.

With their focus on the business (wrought by eliminating the Business Analyst barrier between them and the Stakeholder), they use business metrics as the measures of success. The focus is on results, not on creating fear to innovate. *Estimates* beg to be compared to actuals; that leads to more pointless discussion about why there was a difference between estimates and actuals (something that a Manager role can really get sink their teeth into). So don't make estimates; rather focus on doing what is most important, and if it is too hard, do something else. With that attitude, *iterations* lose all value.

Forward has embraced services architectures, and strive to build simple services deployed in frameworks. Developers focus on keeping each micro-service small, often having it do only one thing. If the need to enhance the service arises, the service is simply rewritten. It is a rare service that doesn't need changing on occasion, so no service is actually very old. If services are small and have short

lifespans, there are a set of best practices (designed to create maintainable, long-lived applications) that are rendered moot: *acceptance tests, unit tests, refactoring,* and *patterns*.

Finally, we practice *continuous deployment* of the small applications in lieu of *continuous integration* of a bigger application. Statistics bare this out: From active monitoring of GitHub over a recent seven day period, 38 of our developers deployed 492 times against 86 projects. That is a pace of one deployment every 5 minutes of a business day. Rather than focus on perfect deployments (although it is harder to make mistakes on something small), we focus on rapid rollback and staged deployments to catch the inevitable slips. We note that this is consistent with the Lean tradeoff of *fast failure* over *defect prevention*.

This continuous deployment is worth one more footnote on roles. Developers do their own deployments, mostly into virtual machines in the cloud. Much of the traditional IT role of controlling the deployment environment has disappeared. It makes us faster.

New, broader trust relationship

We initially discussed the shift in trust required in moving from traditional processes to Agile processes. The move from Agile to Programmer Anarchy requires a similar shift in trust.

In our environment at Forward, Stakeholders set goals and objectives to be accomplished and leave the implementation of those goals and objectives to the Developers. The Stakeholders may be constantly consulted for domain knowledge, but they are no longer the source for stories, nor are they setting the priority of

stories day to day. The Stakeholders *trust* that the Developers will drive to meet the goals and objectives in the most clever, efficient way.

We avoid the trap we are now calling *Story Tyranny*. Consider the following environment:

- ☒ Development is driven through stories,
- ☒ Stories are small,
- ☒ Customer/Stakeholder sets the priority of stories, and
- ☒ Estimates, metrics, and deliverables are centered around stories.

It sounds perfectly Agile, and it is. We had teams in this environment, yet we found our Developers were becoming unmotivated, focusing solely on delivering the stream of stories they were presented as fast as possible. Their understanding of the business environment they were building was unexploited. Time to reflect on the direction and suggest innovative changes was missing. In essence, they had become mindless drones to the story process.

With Programming Anarchy running successfully in other parts of the organization, it took very little to shake the developers out of their lethargy. "What do you think we should be doing? Why aren't you doing that already?" Now they are doing the things they think will make the project successful, and organizing themselves around the task they way they think is best. Anarchy in action. Stakeholder trust.

Common Traits Between Agile and Anarchy

In one way, we would be hard pressed to argue that Anarchy is not another form of Agile. Reviewing each of the elements of the Agile Manifesto [Beck 2001], we concur with them all. Reviewing each of the XP values (feedback, communication, simplicity, courage, and respect) [Beck 2005], we equally subscribe to these. Indeed if anything, we could claim we have more feedback, rely on greater communication, create even simplier solutions, and so on. That is hardly a difference.

At the end of the day, however, the reduction of roles, the myriad of missing Agile standard practices, and the new level of trust required between Stakeholder and Developer lead us to conclude that Programmer Anarchy is indeed post-Agile.

Environmental Enablers

We acknowledge that at Forward, we have some environmental elements that contribute directly to the success we are enjoying with Programmer Anarchy.

Business environment

Forward, founded six years ago, focused on Internet advertising in its early years. In this cowboy environment, bets were placed on search terms in vicious bidding wars. Financial success meant finding the 90% losing search terms quickly, and cutting those losses, while focusing the 10% successes on compound growth. Thus our founder, Neil Hutchinson, was rewarded for being risk-affine (in contrast to risk averse). Our culture is rooted in taking chances, accepting lots of failures, finding failures quickly, and capitalizing on our successes. Failures are not over-analyzed; that wastes a lot of time. Similarly, successes are not over-analyzed either. We just accept them and exploit them.

Forward has enjoyed years of triple-digit growth under this philosophy. Financial success in combination with our risk-affine nature encourages (and even demands) constant experimentation that we undertake every day.

Developer culture

About four years ago, Forward brought in ThoughtWorks to implement a new commercial Web site. ThoughtWorks over-delivered in a short, 10-week project. Neil Hutchinson and his team recognized two things: Agile processes are brutally efficient and remarkably predictable, and fewer skilled programmers armed with supporting tools and processes are more valuable than large numbers of out-sourced developers. Over the next year, Forward began recruiting skilled (and relatively expensive) developers, turning them loose on various projects. Financial results grew quickly.

Neil seeded a programmer into a key revenue-producing group that had just lost its manager. Left to his own devices, this programmer built what he needed to get his job done and support the rest of the team. They made more money. He did additional things on his own initiative. More money again. Not one to miss the obvious, Neil pushed more skilled programmers into the mix with even greater benefits accruing. Thus began Programmer Anarchy.

The programmers exhibited several key charactertics. In general, they had broad industry experience from consulting, derived personal satisfaction from deliverying software that others really use, and a thirst for more technical knowledge. They respected each other, and invited others to join them only if they felt mutual respect would be preserved (our recruiting process aligns with this in the obvious ways).

In summary, Forward is developer-focused, our developers have clarity of business success, and respect one another.

Experimentation drives innovation

Our developers identified a core belief themselves: *experimentation drives innovation*. Often experiments fail; the experiments of our developers are no different. Their first attempt to do something with map-reduce failed. Ditto for their first use of Clojure as an alternative to our default Ruby preference. Nevertheless, the developers learned. These learnings were soon applied to new problems for which these technologies did work. Our production Hadoop cluster runs thousands of jobs for us daily, taking hours instead of days to do the work we require. Clojure is now at the heart of several of our systems, exploited for what it does best.

Courage, aka fearless

We believe that fear kills the spirit of the individual and the team, is a powerful de-motivator, and frankly makes work feel like work. Forward continually strives to eliminate fear through its practices. If we want to be truly successful, we cannot let fear tame us. Paul Fisher, one of our new senior leaders with a VC background, wrapped up his first presentation to the company by quoting, “The greatest barrier to success is the fear of failure.”

From the top of the company on down, we work to keep fears at bay.

Required environmental factors

The environmental factors we have discussed above are certainly sufficient to enable Programming Anarchy. We can’t say with any certainty, however, that you need all of them in order for Programming Anarchy to succeed. Indeed, we hope that

you would need far fewer of them for success. For us, this is an area of interest and further study.

Conclusion

Programmer Anarchy represents an evolving set of practices that offers an alternative to Agile practices. The successful adoption of Programmer Anarchy in Forward and a few other companies bodes well. Still, it seems premature to announce the demise of Agile without more adopters as well as more experience with the execution of Programmer Anarchy.

References

Beck, Kent & Other (2001) *Agile Manifesto*. Retrieved 18 March 2011 from http://agilemanifesto.org/

Beck, Kent (2005) *Extreme Programming Explained, 2nd Edition*. Retrieved 18 March 2011 from http://www.physicsdaily.com/physics/Extreme_programming

Durrand, Mark (2011) *Personal conversations in February 2011.*

North, Dan (2011) *From months to minutes – upping the stakes*. QCon London 2011. Retrieved 18 March 2011 from http://qconlondon.com/london-2011/presentation/From+months+to+minutes+-+upping+the+stakes