Putting the Kart before the Horse?

by Antony Marcano

Deep down, I'm a frustrated racing driver—one who occasionally indulged his passion at his local go-kart circuit (coincidentally the same track on which Lewis Hamilton, current Formula One world champion, first learned to race). Later, I raced in an adults' amateur karting league—pure exhilaration! Super-car acceleration, 70 mph+ top speeds, and no seatbelts!

> "Scrum doesn't tell you how to develop software while you are inside the sprint."

My passion for racing seems to have rubbed off on my ten-year-old son. After his many polite and persistent requests, I recently took him karting for the first time.

While experienced racing drivers make speed seem effortless, consistently fast lap times are physically demanding and require enormous skill—even in go-karts. At the open practice session my son and I attended, novice drivers seemed to approach their first lap as if it really were as easy as experienced drivers make it look. They floored the throttle and flew down the straight only to crash—painfully and expensively—into the barrier at the first hairpin turn.

Thanks to a little coaching from me, my son started out slowly. At low speed he practiced the line I showed him on each corner, gradually improving the quality of his cornering and then increasing his speed. He wasn't concerned with going full-throttle. He was developing the cornering skills needed to achieve consistently fast average speeds. I suggested an earlier braking point here and a later apex there and by the end of the afternoon he was achieving the fastest and most *consistent* lap times—even faster than the adults (except for me, of course).

Organizations starting their transition to agile methods with Scrum are much like the novice drivers trying to go full-throttle before learning how to take the corners. Before being able to truly bend and change their software with speed and ease, they fly into short iterations and frequent incremental product delivery—crashing painfully into the barrier of hard-to-change software.

As described on controlchaos.com, Scrum is "… used to manage and control complex software and product development using iterative, incremental practices." Using timeboxes of two to four weeks, called "sprints," Scrum arms you with techniques and memorable terminology for *managing* iterative and incremental development. At the end of each sprint, working software should be demonstrated for customer feedback or immediate release. Other than techniques for planning, managing, and sharing information, Scrum doesn't tell you how to develop software while you are inside the sprint. For greatest success, it relies on your team's having the software practices and associated skills for creating software that is truly "soft," malleable, and inexpensive to change.

Another racing analogy comes to mind to explain this. Scrum is more akin to what happens trackside—how the racing team plans the race, monitors lap times, communicates information, and reflects and learns from the race event. It doesn't tell you how to build a race car or how to drive it. It relies on the race engineers and the driver to have the skills to do that already.

Despite this, many organizations declare that they are "going agile" and see Scrum as the panacea to the problems standing between them and that goal. They start with the expectation that the light at the end of the agile-adoption tunnel is their mastery of Scrum, only to find that it is the spotlight that illuminates the brittle nature of a product created using legacy development practices.

In my experience, these organizations:

- Assume that, once implemented, the software will be expensive to change, so they create speculative designs intended to solve future problems—making the design far more complex than necessary
- Develop the software in a way that makes evolutionary design impractical, causing them to slice the product increments by "horizontal" architectural component, which makes it difficult to demonstrate the value to the customer, reducing trust and delaying feedback
- Believe that writing unit tests slows you down, leading to low unit test coverage and infrequent test runs and causing deteriorating throughput by deferring the detection of bugs, often until it is too late
- Underestimate the gradient of the learning curve in automating acceptance tests, abandoning it before the true value can be realized, which reduces clarity of requirements and consuming the sprint with increasing manual regression testing overheads
- Continue with a phased development cycle separating programmers and testers (and other disciplines), making communicating changes in requirements unnecessarily difficult and causing friction between disciplines

After an eventual peak in demonstrable features, these behaviors contribute to an ever-decreasing velocity and an increasing proportion of each new sprint spent debugging and testing bug fixes. The term "agile" is quickly replaced with "fragile" and feels more like *brittleware development* than software development.

The team reacts by spilling activities, like testing, into later sprints or abandoning testing altogether and pulling

design into earlier sprints. Superficially, these teams appear to be agile with index cards and project boards and daily stand ups called scrums. But the time between successful demonstrations of working software continually grows, well beyond a single sprint. Agile this is not.

Now, I do appreciate that there are many Scrum-first agile adopters out there who are experiencing success. However, this perception of success may well be relative to how the organization was doing before using Scrum and very much related to what they expected to happen once they'd mastered Scrum.

Adopting Scrum *before* developing the technical practices required to create easy-to-change software does have a benefit. At the end of each sprint that fails to show a product increment or refinement of value to the customer, Scrum painfully highlights how hard it really is to change the software produced by these legacy practices. Sometimes this is the only way to influence cultural change—for the organization to learn a few lessons the hard way.

That was how the lesson was learned for one delegate who spoke up during a presentation at a conference I attended last year. With the benefit of hindsight, he wished that his organization had started with the agile technical practices first. Strangely, it was only after hearing his story that I reflected on my own experience in that regard. It was only then that I *really* appreciated how much "going agile" with Scrum first was actually putting the cart before the horse.

Interestingly, as I write this column, I realize that I'd already been doing exactly what that delegate had wished for. Over the past year, one of my clients has, with my help, gradually reduced the time between delivering releasable software from three months to two weeks. The client did this by first focusing on its technical practices. Much like my son's approach to delivering faster lap times, the client started slow and gradually developed the skills required to deliver software more quickly, frequently, and consistently through improved quality.

It was only when the frequency of each releasable version of its product had been reduced to about six weeks—coincidentally closer to the typical recommended length of a sprint—that the team started to employ any noteworthy techniques from Scrum. I guess the team's skill on the track had reached a point where it saw the value of having better trackside support and information. I suspect that for my son, at the rate his lap times are improving, it'll not be long before he will want the same. **{end}**

Have the Last Word!

If you have a point to make or a side to take on issues and trends that affect the industry, we want to hear from you.

We are looking for insightful, thought-provoking commentary for possible use as The Last Word.

Please send an abstract to editors@bettersoftware.com.

Index to Advertisers

Display Advertising
advertisingsales@sqe.com

All Other Inquiries
info@bettersoftware.com

Better Software (USPS: 019-578, ISSN: 1553-1929) is published seven times per year January/February, March, April, May/June, July/August, September/October, November/December. Subscription rate is US $39.99 per year. A US $35 shipping charge is incurred for all non-US addresses. Payments to Software Quality Engineering must be made in US funds drawn from a US bank. For more information, contact info@bettersoftware.com or call 800.450.7854. Back issues may be purchased for $15 per issue (plus shipping). Volume discounts available.

Periodicals Postage paid in Orange Park, FL, and other mailing offices. POSTMASTER: Send address changes to Better Software, 330 Corporate Way, Suite 300, Orange Park, FL 32073, info@bettersoftware.com.