Four Principles of Agile Triage

Doug Rosenberg
ICONIX

There's an epidemic of bad software floating around these days. Chances are you've encountered buggy and/or unnecessarily hard to use software recently.

This epidemic isn't really surprising, especially if you've read the books I've written with Matt Stephens: Design Driven Testing or Extreme Programming Refactored or Agile Development with ICONIX Process. There's a bunch of agile development shops out there underspecifying their software and (as a result) testing it inadequately, so that you and I can have the privilege of debugging it for them and putting new user stories onto their backlog.

Imagine for a moment that you're working at a company where the agile development process has gone off the rails and there is a train wreck. There's a lot of smoke from the burndown charts burning up. Bodies are strewn all around, there are broken bones and people are bleeding everywhere. Some parts of your code are savable, and some have to be thrown away. You need to triage the situation and do damage control. Here are some guiding principles for your triage effort:

1. **You can't unit test what you forgot to remember.**
2. **You can't do scenario testing without modeling scenarios**
3. **You can't do requirement testing without modeling requirements**
4. **Excessive timeboxing results in shoddy workmanship**

Lets take these one at a time:

Unit testing is necessary but it's not sufficient.

It doesn't seem like it should be necessary to point this out, but by definition, you can only write unit tests for the conditions you've already thought about. And anybody who has ever seen a blue screen of death already knows, programmers don't always think of everything when they write code.

Sunny-day scenarios are the easiest to code. Everything works as anticipated and you can instrument your code with unit tests to make sure you haven't made any coding errors. But anticipating all of the rainy-day scenarios is a lot more difficult – even more so when you're living in a timebox. And unit testing doesn't really help you unless you've already remembered to think about it. So the combination of timeboxing and over-relying on unit tests is inevitably going to lead to problems.

You're going to need a couple of other kinds of testing in addition to your unit tests. Those will include **scenario tests** and **requirement tests**. Because it's easy to forget a requirement, and it's even easier to forget a rainy-day scenario.

The most visible sign of a scrummtastically bug-infested piece of software is the obvious lack of scenario testing. This results from project management robotically following what Matt Stephens so aptly termed **The Green Bar of Shangri-La**. In other words, all the unit tests pass, so hit the **Ship It** button and go on to the next sprint. But it's obvious to anybody who tries to actually use the software that it's actually full of bugs. These would be your customers, who presumably buy your products and services with their hard-earned money. So you'd better do some scenario testing. But wait, you have no model of your scenarios. *Che cosa fare?*

In addition to scenario testing, you need to test your software against a set of requirements. This might seem to be obvious, but it often gets forgotten in the rush to be agile. Writing requirements down slows you down, don't you see? And testing to make sure they're all satisfied slows you down even more. You could be shipping three new bug-filled releases in the time it takes to verify the requirements for one release. And your clients all love you more for delivering three new bug-filled releases. Sure they do.

Finally, we all work more carefully under time pressure. And even more carefully when there is perpetual time pressure. Yep, all of that time pressure helps you deliver higher quality product. Bet on it. Never take the time to do it right the first time, when you can take even more time to get it wrong a second and third time.

A Visit from The Hyphen Police

Another telltale sign that the relentless-time-pressure-pulse of the scrum cycle is leading you into a danger zone is when your software begins to develop an increasingly "user-hostile" tone. It's part of the scrummtastic mindset that the users exist to serve the developers, by feeding them bug reports to be added to the perpetual backlog of new user stories that will then be burned-down by the development team. That mindset of burning down the user stories has a funny way of manifesting itself into software that's excessively difficult and unpleasant to use, for thousands or millions of users, in order to save a junior programmer 10 minutes of coding time.

I've found that the difficulty-of-usage level of a piece of software is often an indicator of how buggy it's going to be overall. This is because the lack of attention to detail in the UI is often symptomatic of an overall lack of attention to detail in things like identifying rainy-day scenarios. The software just feels like it was slapped together in a hurry (and it probably was, because everybody's coding in timeboxed sprints).

For example, consider the following abuse case, encountered by me recently in a real piece of production software while attempting to publish a book (not surprisingly there were a host of other bugs found in this same piece of software, most of them significantly worse than this one):

Abuse Case: Enter tax information

The system displays the Tax Information Page, presenting a form for the user to fill out that includes the tax-ID number
The user enters their tax ID number XX-YYYYYYY and clicks Done
The system scans the form for illegal hyphens

The system detects an illegal hyphen and informs the user that they must enter a 9 digit number with no characters other than 0-9. The user removes the hyphen and clicks Done

If anybody had bothered to write this scenario down in this form before the code was written, somebody (I pray) would have asked the question "***Why don't we just remove the hyphens instead of tormenting the user?***".

In the scrummtastic world, there's never time to do this simple review, and the users have to take it and like it. Until you get a competitor who decides not to abuse their users...then your customers leave your business for the competition. And you're left with the magazine articles about what a great success agile has been at your company.

Is your software process broken? Think about how long it would take a competent programmer to remove hyphens from a tax ID number by typing

```
taxID = taxID.replace('-', '');
```

and think about how many users will be annoyed by having to re-enter their tax ID numbers because the programmer was too lazy to do it.

Or, to paraphrase JFK....

Ask not what your users can do to you, ask what you can do to your users.

Try this simple self-assessment test to determine if you might be a candidate for agile triage:

1. [YES / NO] **Can you describe your software process using only the words sprint, burndown chart, unit test, scrum, user story, and stand-up meeting?**
2. [YES / NO] **Do you have trouble with getting good unit test coverage across all of your stories?**
3. [YES / NO] **Do your burndown charts have a tendency to burn up?**
4. [YES / NO] **Do you frequently release software into the field that contains major defects?**
5. [YES / NO] **Do you have trouble estimating your sprints accurately**
6. [YES / NO] **Do you sleep poorly at night due to a nagging feeling of impending doom at work?**

If you've answered ***YES*** to two or more of the above questions, you're a good candidate. If you've answered ***YES*** to four or more questions, shoot me an email at agiletriage@iconixsw.com