softwareDecisions
doing the right software right!

What Does "Done" Mean?

By Robbie Mac Iver

It's Not So Simple

On Thanksgiving Day the question on everyone's mind is "when will the turkey be done?" Cook books provide useful estimates of cooking time but they are not a foolproof indication that the turkey is "done". Ovens vary in capability to maintain a constant temperature and opening the oven to baste the turkey frequently will increase the cooking time. Fortunately for all of us turkey eaters, "done" can easily be determined by using a meat thermometer to gauge the desired temperature. Today some turkey's even come with a pop-up device that opens when the predetermined temperature is reached.

Things are not so simple for a software developer wanting to declare his code "done". Estimates that the code should be done in some number of hours are helpful, but hardly an indication that the code is really done. As a result projects often discover additional work is required for features that were previously thought to be done. Schedules and budgets begin to slip, the quality of the solution begins to degrade, and confidence is lost in the development team's ability to deliver. Unfortunately there is no "coding" thermometer to stick into the code to measure its doneness.

How then do we tell if our code is really done? For me, there are three conditions I have to prove as a developer before I can declare my code is "done":

- **My code works and does what it is suppose to do.**
- **My code does not break the build.**
- **My code does not break anyone else's code.**

Let's look at these separately.

Code That Works

First off as a developer I need to clearly understand what my code is supposed to do. Suppose we have the following user story:

As a pet owner I want to reserve the Poodle Palace Suite so that FiFi has the ultimate doggie day care experience.

The development team may break this down into a number of tasks such as:

- Code the reservation user interface.
- Write the reservation classes to verify availability and record the reservation.
- Write the database classes to retrieve and store reservation data.

These tasks could easily be done by different developers, so suppose I took on the database class task. The goal of my task is to accurately retrieve and store reservation data in the database. Some simple tests comparing expected storage and retrieval results to actual results will adequately prove that my database classes work. Am I done? Not quite.

Code That Builds

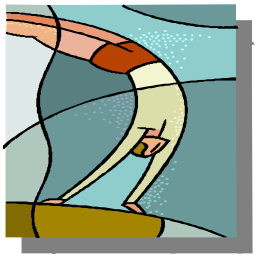
My code has to work in the real application environment, not just on my desktop. That means it has to compile and link with all the other code in the system. So the next step is to add my code to the common source code repository (you have one right?) and rebuild the entire application. Clearly if my code causes the build process to fail I have more work to do, but once the build runs successfully am I done? Not quite.

Code That Works with Others

The user's ultimate goal was not to retrieve and store data in a database. In fact the user doesn't need to know there is a database. The user wanted to make a reservation for FiFi. Therefore my code has to work in concert with the rest of the code involved in making a reservation. That means we need to test not only the database routines, but also the user interface and the business logic that verifies availability and confirms the reservation. Only when all of that works can I say my database code is "done".

Agile Practices

Agility brings many practices to the table that help keep the development team on the right track toward meeting these three steps of "doneness".



First, requirements are most often defined from a business usage perspective. User story formats such as:

"As a <user role> I want to <do some task>
so that <I get some value>"

provide the development team with a clearer picture of what their code is supposed to do. Certainly more detailed requirements will need to be defined, but this is done in the context of the user's business need.

Second, agile development teams often define acceptance conditions and criteria for the business requirement before writing the code.

This is the role of business sponsor or product owner and for our story the criteria might be something like:

- Reservation is recorded
- The Poodle Palace Suite is not double booked
- Confirmation is sent to the pet owner

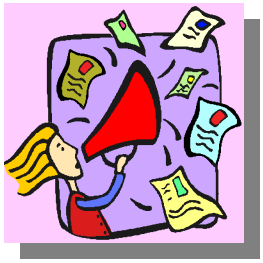
Using these, individual developers can determine when the first step of being done has been met; my code works and does what it is supposed to do.

Third, agile teams tend to integrate new code into the code base very frequently, often every night. Some teams even employ continuous integration techniques. Because of this a developer can confirm every day that the code he wrote yesterday did not break the build, and meets the second step of being done; my code does not break the build.

Fourth, because agile teams integrate new code into the code base frequently, acceptance or system testing can be done early and often. Rather waiting until near the end of "principal development" (as traditional development methodologies would call it) to start user acceptance testing, the integrated code base can be tested throughout the entire development cycle. When all the acceptance conditions are met, the development team knows the third step of being done is complete; my code does not break anyone else's code.

Frequent Feedback

The elapsed time between these three steps is important. The longer I wait to confirm that my code builds with the complete code base, the longer it will take me to fix it if it does not. I will have moved on to other code, so I will have to backtrack to the database class code and re-immerse myself in that to affect the repair. Further, I could have written additional code that compounds the error so there may be even more code to be fixed. Both time and money are wasted.



Similarly the longer I wait to verify that my code functionally works with the code base, and that the rest of the code base still performs as expected, the longer errors will take to correct. In addition to correcting my own code, other developers will have written code based on mine that may be need of repair as well. They too will have moved on to other things and will have to backtrack to make repairs. More time and money are wasted.

Further, until a test of the full user functionality is complete and all the acceptance conditions have been met the development team will not understand if the user's original need has been satisfied. Without that feedback the team could be continuing on the wrong path generating code that is either not needed or will have to be changed later. Changes

and new requirements can also quickly be identified and incorporated into the future development priorities keeping the development team on the track with the true business needs.

Time's a Wasting

Agile teams move at a rapid pace often delivering product increments in short time-boxed intervals, or iterations, typically between 2 and 4 weeks long. These teams can not afford to spend much time correcting coding errors discovered while meeting the three steps of "doneness". Change is therefore introduced incrementally in small doses always validating the integrity of the solution built to date.

Automation can assist. Using testing frameworks such a J-Unit for Java or N-Unit for C#, a developer can create an automated test that demonstrates his code works and meets the first step of being done. Tools such as Ant, can be used to automate the integrated build enabling a developer to quickly realize the second step of being done; my code does not break the build. Finally, additional testing frameworks such as FitNesse can automate acceptance testing to prove the final step of being done; my code does not break anyone else's code.

While in most cases automated testing alone is not sufficient to test all aspects of an application, it can help to validate the three steps of doneness very quickly. Imagine the possibilities, code is checked in every day. Every night the automated build is run, each developer's unit tests are run, and finally the acceptance tests are run. Every day the team knows the results of the changes introduced the previous day and the product owner gains confidence in what is being built and delivered. Time and money can now be focused on what to develop next, rather than fixing the build that failed from code written weeks or months ago.

Robbie Mac Iver is a PMP certified IT project manager and Certified Scrum Master (CSM) adept at bringing business and technology teams together to create and implement innovative solutions to complex business opportunities. He has more than 20 years diversified experience in all phases of the Software Development Life Cycle motivating and focusing teams to achieve both technical and business objectives that result in high value business solutions. He may be reached at rmaciver@swdecisions.com.

softwareDecisions is a Texas Limited Liability Partnership providing information technology consulting services to the greater Houston area, and is dedicated to making software development and integration efforts succeed by addressing the right requirements, focusing on the right priorities, and building the right relationships.