Continuous Integration and Automated Testing Best Practices

03.01.2018

Ed Kuzemchak
Software Design Solutions, Inc.
4091 Saltsburg Rd., Suite 5
Murrysville, PA 15668

software design solutions
an applied visions company

Executive Summary

Continuous Integration (CI) and Automated Testing are key components to a successful Agile software process. Over the last 14 years, SDS has built many embedded systems – including automated test systems, industrial controls, medical devices, and more – for our customers, and we have assisted customers in creating and improving their Agile software environments.

This white paper describes some of the best practices for CI and automated testing that SDS has identified over many years and many projects. Some of these are non-controversial Agile practices that will get little argument from any reader. Others may seem like arbitrary opinion, but they are backed up by a combination of successful projects, as well as painful object lessons.

Table of Contents

Introduction

The core tenet of Agile development is to be constantly building and releasing steadily improving versions of your product, in short increments (two- to three-week sprints), with each release adding and demonstrating some set of new capabilities. This is a departure from the old approach, where teams of programmers would spend months writing code in isolation, then spend days or weeks trying to bring it all together and get it to build into something that could be demonstrated and tested.

Development cannot be very agile if that last integrate-and-build step takes so much effort. And so the discipline of *continuous integration* (CI) arose: a process whereby the executable application is continually rebuilt and retested as coding progresses.

CI requires that developers work from a common source code repository, and integrate – or *check in* – their code into that repository frequently. Every code check-in is verified for correct syntax by an automated build: it is compiled, integrated to the rest of the system, and basic tests are run. If the check-in causes any problems, or “breaks the build” as it’s known, it is flagged immediately and the developer must fix it. In this manner errors in coding are detected earlier in the development process so they can be remediated quickly and efficiently.

CI reduces the work required for each phase of code integration in software development. Early detection and remediation of errors ensures that each version of your software is production-ready.

A closer look at CI

CI involves the following steps:

- Developers check out code from the source code repository to perform coding work.
- Changes are committed back to the repository.
- The CI server monitors the repository.
- The CI server checks out changes as they occur.
- The CI server builds the system and runs some fundamental tests (or *unit tests*).

The outcome of the testing determines the next step. The development team is notified of a successful build, or is alerted if the build or test fails. Failures can be immediately addressed. This process of continuous integration proceeds in this manner throughout the build.

The benefits of CI include:

- Faster identification of errors and bugs
- Easier identification of where errors are located in the code since smaller chunks of code are checked more frequently
- Reduction in time spent debugging
- Increased time available for adding features to your software

So where does automated testing factor in? Best practices dictate the testing performed by the CI system should be automated. Automated testing brings many advantages to CI:

- ***Speed*** – Manual testing is not fast enough. Remember the purpose of CI is to keep the process "continuous."
- ***Agility*** – Automated testing is easily adjusted as needs change. Tools can be quickly swapped out. Manual testing requires rewriting test documentation – a laborious process.
- ***Resource optimization*** – Expensive human resources should be devoted to higher value-added tasks such as fixing coding issues and building new features, rather than constant testing at each code check-in.
- ***Scalability*** – CI involves constant, small changes to code. Automated testing is ideal for small-scale changes since it can be done quickly. Manually testing for constant, small updates is time-consuming and a waste of resources.

How Continuous Integration makes you Agile

Agile development organizes the software development process into user stories, which are broken down into smaller chunks of work called sprints. The sprints – typically one to two weeks in length – are prioritized, with development focusing on the most important chunks first. Application software is built incrementally to ensure it meets the design requirements.

Developers cannot wait until the end of each sprint to integrate their work. It takes too much time to locate and address errors in the large volume of coding that has been generated. Continuous Integration provides a solution.

CI was developed *for* Agile development. It requires software developers to integrate their work frequently, resulting in multiple integrations per day across the development team. Integrations are tested when they are checked in. Errors should be addressed as soon as they arise. This leads to fewer problems by the end of development and saves time trying to go back through large chunks of code, scouring for errors. It keeps your development Agile – responsive to change, and ready to deploy rapidly.

The remainder of this white paper is devoted to a discussion on best practices as they pertain to CI and Automated Testing. Adhering to these principles increases the success of your software development, and keeps you within the Agile framework.

Continuous Integration on every change

Every high-performance software engineer is in constant search of ways to do less work – as in less repetitive, boring, error-prone work. That's what engineers do. Some will even spend two hours building an automated system to perform a one-hour, one-time task. In general, aside from that extreme case, automating any manual process yields a payoff in efficiency and developer productivity.

When we ask clients about their automated build system, many of them say, "Yes, we have one. It runs every night." A common response to our follow-up question of when the last time the build succeeded is, "A few days/weeks/months ago." There are two things wrong with these responses.

First, "every night" is the wrong frequency. Automated builds need to occur on *every* change that goes into source control, *immediately*. There should be a quick "build and smoke test" that can be completed in *five minutes or less* – before the developer has changed context and moved on to the next task. This doesn't mean full testing – the smoke test just proves that the system builds (e.g., that a new file hasn't been neglected to be added to source control), and that the simplest of tests will run. It ensures the most important functions work and the build is stable enough to proceed. After the smoke test, the more elaborate testing can begin.

The second issue relates to broken builds, which are addressed below.

Broken builds are a black mark

Allowing checked-in changes to break a software build and leaving that build broken for any length of time is a signal to your team that substandard results are acceptable and maybe even the norm. This has a significant negative impact on your quality culture and morale.

Broken builds happen; the key is to address them quickly. When the build breaks, someone (preferably the offending party) should stop what they are doing and fix it. That is why the five minute smoke test above is so important. The offending party should not be too busy to fix the build they broke only five minutes ago.

SDS has run an online chat for over 12 years, first in Campfire and now in Slack. In both systems, we have added a bot to integrate our CI system to the chat. The bot posts to the chat channel when a build breaks, identifying the change (and sometimes the author of the change). When this happens, someone stops what they are doing and fixes the build. Yes, if builds were breaking all day long, we would not get anything else accomplished. But if the builds were breaking all day long, it's clear we aren't getting anything useful accomplished anyway, right?

Developers must take responsibility for test writing

Test writing is not a Quality Assurance (QA) or Test Team responsibility. These teams may own the testing process, and they may maintain the test equipment and the test cases. But allowing developers to abdicate the responsibility for test writing harms the product and the team overall. It fosters an adversarial QA approach that is the antithesis of Agile software development and high-performance teams.

Developers need to work directly with the QA and Test Team during requirements analysis, design, and development. The QA/Test Team should be the owners of the test frameworks and the test automation system. They should not be writing tests in a vacuum during or after the development is complete. This is the equivalent of trying to paint over a large dent in an auto fender. You can make it shiny, but it will still be a dent.

White box before black box

There is white box and black box testing, (and shades of gray, but we'll leave those out for now). White box testing is done knowing the inner workings of the system, leveraging that knowledge and perhaps exposing some of those inner workings to the test case. This type of testing is performed with access to the full source code, and source code scans are included as part of the testing process. Full access internally (locally) to the application is provided, including log-in credentials and full authentication. White box testing should be done before an application is released into production, so vulnerabilities can be identified and corrected prior to deployment.

Black box testing is done purely from an interface or user perspective. It mimics a hacker, attacking the application from the outside. It tests the functionality of the application. It is performed based on the assumption that the attacker has no knowledge of the inner workings of the application. This testing looks for known vulnerabilities, weak access controls, and weak defense mechanisms in the application. To be thorough, it is important to test for all variables and scenarios that may come into play.

We prefer white box over black box in all aspects of testing, with the exception of usability testing. The additional knowledge and visibility gained by having "hooks" into the system makes the white box testing process more rigorous and often more robust. There is no comparison in the efficiency with which a white box, scriptable test suite can be written (See Scripting Language below) versus a black box GUI test (see Paradox of GUI Testing below).

Build in a scripting language

A common refrain when the subject of automated testing comes up is, "It's hard to automate things that require user inputs." While that's true, that's no excuse. With a little thought, and some good engineering, it's possible to build your application in such a way that most of its features can be tested without involving the user.

One of our favorite techniques is to build a general-purpose, off-the-shelf scripting language into your application, even if you have no intention of making it available to the end user. We don't care which scripting language you build into your application – Python, Lua, or JavaScript – but if you're looking for a recommendation, we suggest Python.

"Building in" means it has access to the inner state and inner workings of your application. The scripting language should have the ability to perform actions normally performed by the graphical user interface (GUI), such as activating menu actions and the equivalent of drag/drop and selecting actions. Notice we say "the equivalent of." We are not talking about GUI testing (see below). A little bit of code refactoring can connect the menu and the script function to the same capability, so the function can be called from the menu item and from the scripting language.

Scripting can do things manual testers will tire of, such as loading or saving a file thousands of times to check for memory leaks in that portion of the code. Or having a regression test that times the loading of a large input file, then watches for slowdowns in the file load process and catches any change that causes it. We have done this, and it has paid off.

Building a scripting language into your application early prevents two major mistakes later:

- Building a small command access into the application. This is when you write a command to perform a single action, and then another command to perform another action. Before long, your list of commands grows into a scripting language that you built one piece at a time. This is a waste of time and resources.

- Buying an off-the-shelf automated testing tool that has a scripting language built into it. These tools come with substandard scripting languages that will not deliver the functionality that you need.

The paradox of GUI testing

There comes a point where you feel that you *need* to test things that respond to user inputs - for example, testing the performance of the graphical user interface (GUI) itself. GUI testing checks the system's screens and controls, including menus, icons, toolbars, and dialog boxes. GUI testing can be performed in three ways:

- ***Manually*** – A human reads a written set of instructions and performs the test.
- ***Record and replay*** – Automated tools are used to record test steps. These steps are executed by the automated tools during the replay process.
- ***Model-based testing*** – Models are used to generate test cases for the software application to discover if the actual outputs match the expected outputs.

This is probably the most controversial statement in this white paper: We have participated in half a dozen pure GUI testing projects and *none of them had a positive ROI (return on investment)*. Sure, we found bugs while writing the GUI tests. In fact, we found all the bugs while writing the tests. Then came the ROI drain of maintaining a GUI test suite. Months and years go by fixing the GUI tests, adapting to changes, and making them robust under all possible circumstances. Bye-bye ROI.

Macro-record-playback tests have a shelf life of about one software revision. Then they are useless and impossible to maintain. More sophisticated, object-matching frameworks yield tests that can be maintained, but in our experience, the cost of maintaining those tests would have been better spent writing new tests (and finding new bugs), or doing other process improvements.

So what is the answer? Scripting. In one of our systems (a drag-and-drop application builder with many GUI actions), SDS automated tens of thousands of "GUI tests" with no actual GUI interaction. Every GUI action was performed one level down by scripting that drove the same action as the GUI.

"One level down" means that instead of automating the process of navigating to "File," then "Save," and clicking the mouse, you should locate the function in the program that calls this action directly, and test that. You don't need to test that drag/drop from a

palette to an X,Y position that works in MS Windows. You need to test that the software does the right thing when component C1 is dropped at position X,Y, connected to component C2, and then the “Build” action is performed. Scripting is a far better choice for these scenarios.

You’re probably thinking, *We have to test the GUI. We need to know if it is responsive and visually appealing.* The SDS answer to this is: “Yes, you do need to test the GUI for these elements.” But in our experience, this type of GUI testing is also usability testing, which still requires human interaction. An automated GUI test won’t notice an annoying GUI flicker or a 300-ms latency, which is enough to annoy the user unless a GUI test is specifically written for it. But this is another waste of resources.

Perform your usability testing at the GUI manually. Do everything else that you can with scripting.

Start small and automate everything

These two statements seem contradictory, and they are. But at the same time, they are not. The end goal of a CI system is to automate the entire build / test / deliver / deploy process. One of our largest projects had a system that automated everything. It could:

- Take a source change
- Build
- Perform a smoke test
- Perform a full validation test
- Build the installer
- Test the installer
- Generate the release notes
- FTP the installer to the customer
- Send an email to the customer informing them of the release

That is a world-class system. It felt great to sit and watch it do our work while we added value in other ways.

But even that project started small, with a simple automated CI build. A few unit tests and an automated installer build were then added. Each step added to the system's capability, removing human error and freeing developers and testers for more engaging, productive work.

No need to roll your own CI system

It used to be challenging and time-consuming to build a CI system for software development. This is no longer the case. Here are the top three tools that deliver a quality CI system:

Jenkins

- An extensible automated server that can be used as a basic CI server out-of-the-box.
- Java-based program with packages for Windows, Mac OS X, and other Unix-like operating systems.
- Simple configuration through a web interface.
- Includes error checks and built-in help.
- Offers hundreds of plugins so you can integrate with just about every tool in the delivery chain.
- The Pipeline plugin allows developers to execute the entire build-test-deploy pipeline into a Jenkins file. The pipeline is treated like another piece of code that is checked in. Easily distribute work across multiple machines; building, testing, and deploying across multiple platforms is faster.

TeamCity

- Java-based CI server from JetBrains.
- Build, check, and run automated tests prior to committing source code changes.
- Progress reporting identifies issues during development.
- Ability to break down a single build procedure into pieces that can be run in parallel across many computers, on the cloud.
- Supports CI for Java, .NET, and mobile platform development.

- Over 100 plugins for adding features or create custom features through the TeamCity API.
- Free version provides 100 build configurations, access to all features, and three build agents.
- Option to purchase additional build agents with more build configurations.

Bamboo

- CI server from Atlassian.
- Ability to create build plans in stages.
- Establish triggers to start builds when code commits are made.
- Run multiple automated tests in parallel to thoroughly test each change.
- Real-time notifications of failed builds and tests.
- Integrates with Jira software, Bitbucket, Fisheye, and other tools.
- More than 150 add-ons available when you need to add features and functionality.

Conclusion

Continuous Integration and Automated Testing do not have to be all-or-nothing propositions. Selectively choosing those parts that are easily done first and getting some payback there can go a long way to convincing yourself, and your management, that CI and Automated Testing are well worth the investment.

Knowledge comes with experience. The SDS team has many years under its belt when it comes to Agile software development. Our best practices around CI and automated testing – even those you may find surprising – are based on real projects with real clients. The results speak for themselves.

Ed Kuzemchak
Chief Technology Officer and Director,
Embedded and IoT Engineering

Ed is the founder of Software Design Solutions. He has been creating embedded software solutions for nearly 30 years and has been the president of Software Design Solutions for over 13 years. The company provides embedded system and Internet of Things (IoT) software development, desktop application development, and software process improvement consulting. Software Design Solutions is also able to provide temporary software engineers for projects greater than the scope of its clients' abilities and to provide experienced programmers to companies who need to focus on other lines of business.