Principles of Continuous Integration
Best Practices to Simultaneously Improve Speed, Quality and Responsiveness in Mobile Development
# Table of Contents

- Mobile Services ........................................................................................................... 3
- Benefits of Continuous Integration ............................................................................. 3
- Key Features of Continuous Integration ........................................................................ 4
  - Small Batch Sizes ........................................................................................................ 4
  - Comprehensive Version Control ............................................................................... 4
  - Simplified Branch Strategies .................................................................................... 5
  - Automation ................................................................................................................ 5
  - Continuous Feedback ................................................................................................. 5
  - Emphasis on Working Builds ..................................................................................... 5
  - Developer-Tester Collaboration .................................................................................. 6
  - Consistent Environments ........................................................................................... 6
- Continuous Integration and Mobile Development ......................................................... 6
- About SOASTA, Inc ....................................................................................................... 7
Mobile Services

In the Post-PC Era, companies and individuals rely more and more on mobile devices to access and share information hosted in both private and public cloud services. Mobile development is about more than client-side applications. A user’s ability to succeed depends on how well all aspects of the software service work together.

Service quality happens across multiple dimensions: functionality, operability and deliverability.

  - Functionality is the area developers and testers focus on most. Does the application let the user accomplish what they’re trying to do? Does it give them access to the right data in the right way at the right time?
  
  - Operability means that, aside from functional correctness, software systems provide high levels of availability, scalability, performance, consistency, and security. They should not lose data or allow it to be stolen. They should not go down at critical times (such as April 14th for a tax service) or in the face of localized outages (such as a hurricane). They should be able to scale to maintain performance in the face of dynamic load (again, such as on April 14th for the tax service).
  
  - Deliverability defines a service’s ability to deliver functional and operational improvements quickly, frequently and safely. The combination of mobile devices and cloud services drives the cost of change for users towards zero. Consumer services and app stores encourage people to expect ever-greater responsiveness. The consumerization of IT leads employees to have the same expectations of enterprise services. Their patience for dealing with bugs-in-the-wild, missing functionality, or poor operability has narrowed from months or weeks to days or hours.

Benefits of Continuous Integration

In order to be able to deliver functional and operational improvements quickly, frequently and safely, software providers need to minimize the risk of change. Software risk manifests itself as complexity and waste. Continuous integration can help reduce both and, as a result, can significantly compress the development lifecycle.

Traditionally, software development organizations have treated integration testing as a separate development phase. Integration testing combined weeks or months of work by multiple developers or teams of developers. It was time-consuming and painful. Integration bugs often arose from interactions between code that had been written early in the development process. In order to fix those bugs, developers had to think long and hard about why they’d written that code in the first place.

Development organizations viewed integration testing as difficult. As a result and out of fear, they delayed it. Ironically, delaying integration testing increases its complexity, and risk instead of reducing it.
Continuous integration turns integration testing on its head. Rather than integrating code at the end of a project or even a sprint, continuous integration makes code integration uninterrupted, occurring consistently throughout the project. It makes the unit of integration as small as possible, and makes it happen as quickly as possible after the writing of any given unit of code. Minimizing the number of changes being integrated at one time makes it easier to isolate and correct the source of any integration bugs. Continuous integration reduces complexity and its attendant risk.

Waste is another source of deliverability risk. Waste comes from spending time and resources on activities that don’t add value. These activities often include manual configuration that are also error-prone. Fixing problems caused by manual configuration is its own form of waste.

Continuous integration addresses waste through automation. If the build server runs a build on every check-in, developers don’t have to remember to do a build. If it runs all tests on every build, they don’t have to worry about checking whether important tests were run.

**Key Features of Continuous Integration**

Continuous integration involves more than just tipping up a Jenkins server. At its core it’s a set of practices, not just a set of tools. The entire development organization, including developers, testers and product owners, needs to engage with these practices.

Key Continuous Integration Practices include:

1. Small batch sizes
2. Comprehensive version control
3. Simple branch strategies
4. Automation
5. Continuous feedback
6. Emphasis on working builds
7. Developer-tester collaboration
8. Consistent environments

**Small Batch Sizes**

The larger the unit of work, the greater its complexity. Complexity makes it harder to understand, plan, design, implement, test, or release changes. Reducing batch size is by far the most important component of continuous integration. Developers should check working code into version control daily, if not more often. Teams should strive to reduce the size of user stories.

From a customer visibility perspective, some stories may take longer than a day to complete. From a development and release perspective, a story may consist of multiple tasks, each of which only takes a few hours to build and test. Consider the example of a web page implemented using the Model-View-Controller pattern. The model can be developed and released into production first without any end-user impact. This approach has the added advantage of flushing out potential production issues sooner.

**Comprehensive Version Control**

Without version control there is no reliable way to know what a given unit of work contains. At any point in the development-testing-release process, the first debugging question should always be, “What changed?” A version control system helps answer that question.

Version control also ensures reproducible builds. It removes the waste of manually assembling code. Reproducibility is a key component of safety: you can’t safely deliver frequent changes if you don’t know what you’re releasing.

Version control applies to more than just functional application code. Ideally, every aspect of the service environment – application code, configuration files, build server configurations, infrastructure configurations, etc. – is versionable. Disaster recovery is part of operability. If your environment suffered a catastrophic loss, could you rebuild it from scratch? Comprehensive version control helps answer this question “yes”.
Simplified Branch Strategies

Development teams often employ complicated version control branching strategies to give themselves greater control over the release process. Unfortunately, this approach increases both complexity and risk.

The need for complicated branching strategies generally reflects the need to manage long-running development processes. If developers complete and commit user stories in less than a day, for example, they don’t need long-running feature branches.

Continuous delivery is a practice that extends continuous integration all the way to production. It delivers changes to customers using the same cadence that developers complete them. When done correctly, continuous delivery eliminates the need for branches (other than master) that live for more than a few hours. However, continuous delivery requires tremendous organizational maturity. Not every organization can completely do away with branches. Just as they should strive to minimize batch sizes, every team should strive to minimize branch complexity, especially as the two go hand-in-hand.

Automation

Unnecessary manual activities generate waste, both directly by defocusing engineers from value-adding activities, and indirectly by causing errors that require further defocusing effort. Continuous integration relies on automation. No one should have to manually trigger builds, or test runs, or decide which tests should run under what circumstances. No one should have to manually generate and distribute test results reports.

Traditionally, testers spend a significant percentage of their time executing and monitoring test runs. In a continuous integration environment, they spend the bulk of their time writing tests and helping developers understand the results. Release engineers have traditionally spent their time executing releases. In a continuous integration environment, their time is spent building and managing the release machine.

Continuous Feedback

The purpose of continuous integration is to help development teams find and fix problems more quickly and easily. This can’t be achieved without visibility into the status of the build process. In the case of a failed build, everyone on the team, including developers, testers and product owners, should receive immediate notification. Bringing everyone into the process creates a sense of shared responsibility. It helps reinforce the continuous integration mantra throughout the organization:

1. Build every change
2. Test every build
3. Fix every failure

Emphasis on Working Builds

Ideally, developers check in working code and builds succeed. Even if developers never make mistakes, failures can come from interactions between working code (thus the emphasis on integration). The worst kind of bug is one that rots. If teams allow build failures to pile up over the course of days or week, the whole purpose of continuous integration is lost.

Continuous integration teams need to prioritize working code over adding new features. A build failure notification should serve as a signal to stop checking in code and devote the necessary resources to fix the build. While doing this may appear to slow things down, addressing problems when they are the smallest and easiest to understand actually speeds development up.
Developer-Tester Collaboration

Continuous integration build failures may be relatively quick and easy to fix. As much as possible, however, they should be avoided. Development teams should have the goal of checking in complete, correct code. The further forward in the lifecycle you find a bug, the less costly it is to fix. Finding bugs prior to check-in is less costly still than finding them during a continuous integration build.

The best way to catch bugs prior to check-in is to have developers and testers work together, rather than to treat testing as a post-coding activity. Short-lived feature branches are ideal for this purpose. Developers and testers can share, review and run each other’s code. By running the test suite locally, they can minimize the likelihood of discovering bugs later in the lifecycle.

The value of running tests depends on the quality and completeness of the tests. In a continuous integration environment, tests are code just like any other code. Story planning needs to account for test development time. Development teams need to pay as much attention to specifying needed tests as they do to specifying needed functions. Test specification needs to account for both non-functional and functional validation.

Consistent Environments

Few things are more frustrating than “it worked on my machine”. Environmental inconsistency is the primary cause of this situation. Manual system configuration exacerbates consistency problems and it’s common for developers to be running different versions of an SDK. They also often test against system software that differs from the software running in the integration environment. That environment, in turn, doesn’t match production. The result is pure waste.

Fortunately, this problem can be solved using configuration automation and by treating infrastructure as code. A single set of configuration scripts can be used to provision development, testing and production environments. Consistent deployment of configuration changes across large numbers of environments and machines becomes as simple as checking in a configuration script change.

Continuous Integration and Mobile Development

The principles described here apply both to device-side and to server-side software. Providing a mobile application involves coupling the client experience with one or more back-end systems. Continuous integration needs to account for both sides of the client-server equation.

Server-side software should undergo continuous non-functional (performance, security, availability, etc.) and functional testing. Configuration automation should be used from desktop to production to ensure consistent server-side software development, testing and operational environments.

Development teams should test client and server software both together and separately. Tight release coupling between client and server increases the effective batch size, adding risk and decreasing speed. As much as possible, development organizations should decouple them.

At first glance, continuous integration may seem like it increases development complexity. However, the real complexity comes from the fact that users consider the entirety of their experience using a mobile application. Continuous integration makes it possible to simultaneously solve for all three dimensions, allowing mobile application providers to improve speed, quality and responsiveness at the same time.
About SOASTA, Inc.

SOASTA is the leader in cloud testing. Its web and mobile test automation and monitoring solutions, SOASTA is the leader in cloud testing. Its web and mobile test automation solutions, SOASTA CloudTest and SOASTA TouchTest, and SOASTA mPulse, enable developers, QA professionals and IT operations teams to test and monitor users with unprecedented speed, scale, and precision, and visibility. The innovative product set streamlines test creation, automates provisioning and execution, and analyzes real user behavior in real time distills analytics to deliver actionable intelligence faster. With SOASTA, companies can have confidence that their applications will perform as designed, delivering quality user experiences every time even in peak traffic. SOASTA’s customers are many of today’s most successful brands including Avaya, American Girl, Bonobos, Backcountry.com, Chegg, Experian, Gilt Groupe, Hallmark, Intuit, Microsoft, and Netflix. SOASTA is privately held and headquartered in Mountain View, Calif. For more information about SOASTA, please visit http://www.soasta.com.