Using Agile and DevOps to Achieve Quality by Design

INSTEAD OF TESTING FOR QUALITY ON A FINISHED PRODUCT, HERE ARE THREE APPROACHES TO BUILD IN QUALITY AT EVERY STEP OF THE SOFTWARE LIFECYCLE.

by Mike Sowers | *msowers@techwell.com*

Anyone who has attempted to architect, design, develop, test, deliver, and deploy business value with software knows that there are no silver bullets. Humans are fallible, and as hard as we try, we make unintentional mistakes due to the fragility of our methods, tools, techniques, and skill sets. Mistakes are amplified by the stresses of our working environment.

Whether the approach taken is traditional or agile, these fragilities and stresses always exist. This is especially true with projects that rely on continuous integration or a DevOps framework. However, our opportunity to better mitigate these risks significantly improves as we adopt more modern software engineering practices. Key principles such as rapid design and refactoring, delivering small increments of customer value more frequently, engaging customers throughout the process, failing fast and learning, collaboration across development and operations, simplicity, and automation are accelerating our ability to significantly ratchet up our engineering excellence.

Shift from a traditional mental model toward a continuous, iterative series of automated verification steps.

The software development industry now has the opportunity to realize the goal we've always had to build in quality rather than attempt to test quality in after the fact.

Over the years, I have learned a number of techniques and approaches to achieve the goal of designing for quality at every step of the software engineering value stream. This works for both agile and DevOps project environments.

Invest in a Plan

Upfront planning approaches and whole-team engagement and accountability help clarify the value increments to be delivered.

Collective ownership engages the entire team in quality delivery. The customer, product owner, analyst, architect, developer, tester, and operations roles form a cohesive team, laser-focused on delivering customer value.

Design planning helps teams collectively envision and design features and functionality from the beginning of the project.

Prototyping offers the team techniques to assist with the definition of functional and nonfunctional requirements. This includes wireframes, mockups, personas, and in-sprint experiments.

Peer reviews enlist the expertise of others and help ensure that the multiple perspectives are represented throughout a project.

This includes pairing between the developer and tester, business analysts and product owner, and operations team members.

Grooming is vital to keep the team aligned on the most important value to be delivered. This can be accomplished through backlog grooming, prioritization, and story refinement and definition.

Test-first development assists the team in defining the required customer value prior to implementing code.

Techniques such as specification by example help specify the requirements and "test first" approaches help to ensure code is right the first time.

Implement in Small Increments

Implementation approaches quickly verify small increments of change and new functionality.

Guidelines and checklists for architecture, design, coding, testing, and operations, as well as other team norms, promote preventive practices and help avoid mistakes and rework.

Static and dynamic analysis tools help the team understand code structure, complexity, coverage, and security vulnerabilities, and detect anomalies that need to be corrected quickly.

Unit testing and refactoring reduce the risk that code changes cause additional problems, promote early detection of defects at the structural level, and allow the code to be refined with refactoring.

Mocking helps the team have higher confidence in progressive integrations, abstract out dependencies, and verify interactions between dependent classes early. This is accomplished by faking and stubbing at the interface level.

Continuous integration for each code commit and performing integration testing at multiple levels effectively exposes interface defects at the unit and each succeeding level as code is further integrated into the application or system.

Automation of the infrastructure that moves the code and that assists with the testing of the code eliminates errors so that code deployment and delivery yields consistent and repeatable results. Applying the concept of "infrastructure as code" and then defining and automating the value stream (the pipeline) codifies the creation of environments and reduces the variability between development, testing, staging, and preproduction.

Defining "done" clearly specifies the criteria necessary to move an increment of value forward in the pipeline, eliminating disagreements and setting expectations up front for what work completion means.

Think Continuous

After planning and incremental implementation, continuous processing techniques can lead to rapid product delivery.

Fast feedback informs the team immediately when something is not performing or is drifting off course by using in-process dashboards or postprocess production and user experience monitoring.

Iterative risk assessments based on fast feedback enable the team to adjust their verification focus and strategies with immediacy.

Functional and nonfunctional testing increase defect yields earlier in the development lifecycle, as does performing story tests, exploratory tests, and user acceptance tests. Using heuristics helps the team design better tests.

Regression testing for each level validates changes and ensures that a recent change did not impact another area of the application or system.

Combining microservices with container deployment, thus decomposing the application into smaller services, results in improved modularity, makes the application easier to test, reduces resource consumption, and speeds deployment. Together, these practices reduce fragility and aid continuous delivery and deployment.

Continuous delivery and deployment provide quick feedback, employ pipeline automation, and keep the code production line running smoothly and efficiently with a defined set of quality assurance steps and gates that are automated.

Operational tests that validate security, user provisioning, backup, and failover are shifted left and incorporated into earlier testing stages. Operational requirements should be specified and agreed upon by the collective DevOps team early in the project.

Feature and application delivery using feature toggles helps reduce risks when a change is deployed, as the change verification rollout and rollback can be better controlled. Change automation adds an additional level of production control and mitigates defect exposure and risks.

Monitoring of both preproduction and production environments provides the team with a deeper understanding about the quality and usage of the customer value being developed and delivered. In addition to standard environment monitoring, the team also can use application monitoring and user experience monitoring and analysis to understand usage patterns more quickly.

All of This Results in Higher Quality

I've always thought of software development, delivery, and deployment as a series of imperfect translations. Defects are often introduced during the translation process from one handoff to the next.

The above approaches enable the team to — from a traditional mental model toward a continuous, iterative series of automated verification steps.

This investment drives software assurance left to earlier in the lifecycle, which helps us achieve the quality-by-design goal and reduces translation errors. [BSM]