PARASOFT®

Today's SDLC Requires a New Perspective On "Test"

Issue 2

Featuring research from Gartner®

In response to today's demand for "Continuous Everything," the software delivery conveyer belt keeps moving faster and faster. However, considering that testing has been the primary constraint of the software delivery process, it's unreasonable to expect that simply speeding up a troubled process will yield better results. (I Love Lucy fans: Just think of Lucy and Ethel at the candy factory, struggling to keep pace as the conveyer belt starts putting out chocolates faster and faster.)

In most organizations, quality software is clearly the intention, yet the culture of the organization drives behavior that increases the risk of exposing faulty software to the market. Most software quality efforts stem from a bottom-up approach to testing, which is focused on adding incremental tests to validate new functionality. This approach is no longer sufficient for today's accelerated release cycles—where faulty software has a direct impact to the brand and bottom line. Releasing with both speed and confidence requires a definitive understanding of each application's distinct business risks and the probability of exposure.

In order to achieve this, we need a method to federate quality information from multiple infrastructure sources (source code management, build management, defect management, testing, etc.). A Development Testing Platform is this central "system of decision" which aligns development activities with business expectations, providing insight and control over the process of creating quality software. The goal is not only to reduce business risk but also dramatically reduce the number of defects that are introduced into the code base in the first place.

The following resources are designed to help software development leaders achieve the optimal balance between speed and quality in order to accelerate the SDLC and release with confidence. To learn more about how Parasoft can help your organization deliver defect-free software efficiently, visit www.parasoft.com.

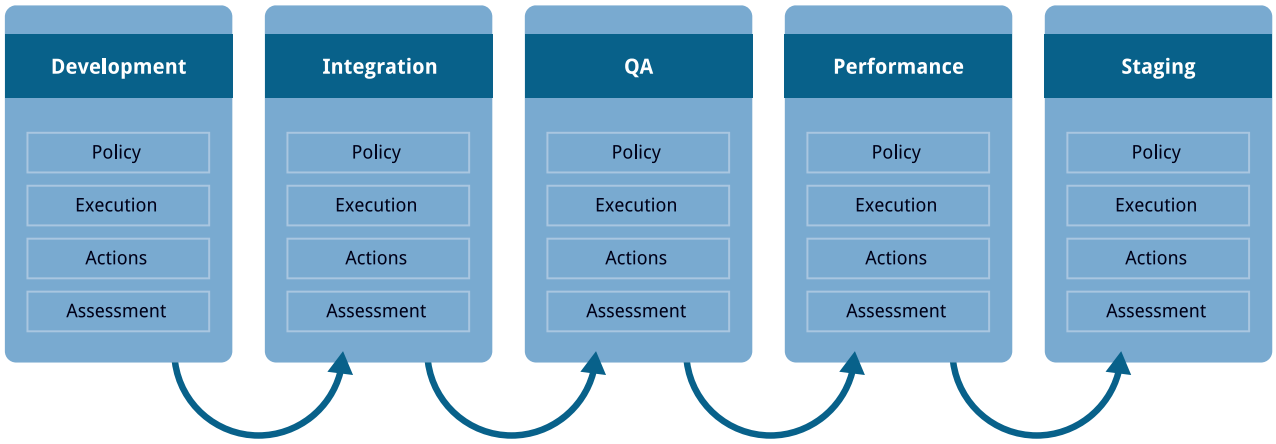
Source: Parasoft

Development Testing Platforms: Essential For Accelerating The SDLC

"Accelerate the SDLC" has become a popular mantra for software development across virtually all industries now that software is increasingly serving as the interface to the business. This is driving, among other things, the push towards DevOps and Continuous Delivery. Yet, considering that software testing has long been a thorn in the side of the development process, it's unreasonable to expect that attempts to speed up an already-troublesome process with more automation will achieve the desired results. If you fail to address the inefficiencies of the testing process, the elephant in the room, you're likely to just end up releasing bugs into the wild at an unprecedented pace.

To truly accelerate software delivery without exposing the business to increased risk, a solid, continuous quality process is essential. Ultimately, such a process provides quantitative assessment of risk and produces actionable tasks that will help mitigate these risks before progressing to the next stage of the SDLC.

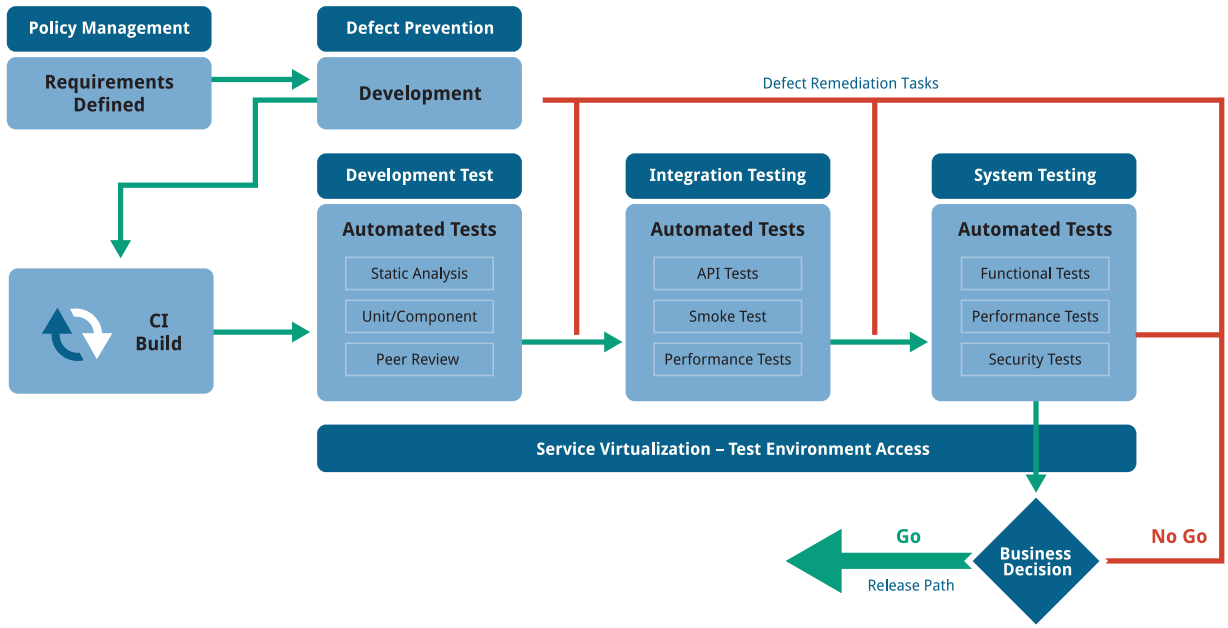
Figure 1 | Mitigate risks before they progress to the next stage of the SDLC



Source: Parasoft

Achieving the required breadth and depth of continuous, automated testing obviously requires a method to federate quality information from multiple infrastructure sources (source code management, build management, defect management, testing, etc.). A Development Testing Platform is this central "system of decision" which translates policies into prioritized tasks as well as delivers insight and control over the process of creating quality software.

Figure 2 Continuous Testing accelerates the SDLC and reduces business risk



Source: Parasoft

The following sections outline key factors to consider when selecting a Development Testing Platform for your organization.

Openness and Ease of Integration

Leveraging a Development Testing Platform requires openness: the platform should furnish well-defined APIs that allow information and data from associated infrastructure systems to be consumed and published with ease. The ease of integrating data from disparate systems will be the key to truly establishing a system of decision.

One of the primary considerations for adopting a commercial Development Testing Platform versus building one yourself should be access to an ecosystem of add-ons or value-added plugins to the platform. A marketplace for plugins significantly reduces the time and effort required to either customize data filtering or integrate with niche tools.

Driven by Policy

A policy is a business expectation translated for the development and testing staff. A Development Testing Platform is the central repository for putting those policies into practice. It correlates policy with analysis techniques and testing practices that assess the level of policy adherence. The platform also generates notifications and tasks by exception, guiding the team to achieve the stated policy objectives.

In broad terms, think of a policy as a non-functional requirement. For example, you could have the "Company X - Secure Coding Policy." This policy would define the minimum criteria for how code should be constructed to prevent and/or eliminate potential application vulnerabilities. The policy must be enforced automatically; in this case, it could be enforced via static code analysis (to prevent vulnerabilities) and application penetration testing (to root out any vulnerabilities that slipped through your prevention efforts and reached the built application).

A policy must have (at least) three components:

1. It must be human-accessible, readable, and understandable. A business expectation should be associated with each policy. A sample policy could read, "Company X is a 125 year old financial institution that bases its success on earning the trust of our clients. Part of that trust includes information security and privacy. A single security breach could erode the trust that we have built with our clients. Additionally, a breached security vulnerability has a physical cost of $250 per record as well as severe negative impacts to stock price and brand equity. This is why our secure coding guidelines have been formally defined and supported by our CEO…"

2. It must be enforceable automatically via an automated, exception-based notification system. Managing the policy itself should not impede productivity. A process that forces developers and testers to manually report on policy adherence is not sustainable.

3. It must be measurable and visible to management. Through a simple, intuitive reporting interface, managers must be able to rapidly assess policy compliance—and, more importantly, determine what actions to take to address non-compliance.

Most organizations have a development or SDLC policy that is passive and reactive. This policy might be referenced when a new hire is brought onboard or when some drastic incident compels management to consult, update, and train on the policy. The reactive nature of how management expectations are expressed and measured poses a significant business risk. The lack of a coordinated governance mechanism also severely hampers IT productivity (since you can't improve what you can't measure).

Policy analysis through a Development Testing Platform is the solution to this pervasive issue. With a central interface where a manager or group lead defines and implements "how," "when," and "why" quality practices are implemented and enforced, management can adapt the process to evolving market conditions, changing regulatory environments, or customer demands. The result: management goals and expectations are translated into executable and monitor-able actions.

The primary business objectives of policy analysis are:

- Expose trends associated with the injection of dangerous patterns in the code
- Target areas where risks can be isolated within a stage
- Identify higher risk activities where defect prevention practices need to be augmented or applied

With effective policy analysis, "policy" is no longer relegated to being a reactive measure that documents what is assumed to occur; it is promoted to being the primary driver for risk mitigation.

As IT deliverables increasingly serve as the "face" of the business, the inherent risks associated with application failure expose the organization to severe financial repercussions. Furthermore, business stakeholders are demanding increased visibility into corporate governance mechanisms. This means that merely documenting policies and processes is no longer sufficient; we must also demonstrate that policies are actually executed in practice.

This centralization of management expectations not only establishes the reference point needed to analyze risk, but also provides the control required to continuously improve the process of delivering software.

Flexible Execution

To ensure speed and accuracy when executing specific analyses or tests, it's imperative to have a platform that offers flexibility—from execution options natively available within the platform, to an API that's specifically designed for executing test artifacts over distributed resources. First, the API must be callable by popular build management, continuous integration, and DevOps tools. Second, the API must provide appropriate operations that orchestrate the execution of test artifacts at the desired stage of the SDLC. This flexibility is key for ensuring speed as well as achieving actionable outcomes that can ultimately mitigate business risks prior to release.

Considering the myriad execution scenarios that could transpire, having the flexibility to run the right tests at the right time becomes the critical path. The flexibility to execute specific sets of tests also requires access to a complete test environment. This is where simulated test environments (via Service Virtualization) become an indispensable component of your development and test infrastructure.

Process Intelligence

Process control throughout the SDLC requires the ability to observe and synthesize data across systems, analysis techniques, and testing practices. Siloed or one-off reports generated by single systems will usually provide a small fraction of the process story. Ultimately, the aggregation and intelligent interpretation of the data generated from various sub-systems should deliver suggestions for optimizing the process.

As the central system of decision for SDLC quality, the Development Testing Platform must readily manage multiple data inputs from various infrastructure sources. The collection of raw observations across systems is the first step in transforming compartmentalized data points into process intelligence. The second step is the ability to process raw observations through correlation, advanced analysis, and the application of patterns. Finally, observations are filtered based on the organization's policies to more accurately pinpoint findings that represent the highest risks associated with the specific stage of the SDLC.

The effective application of advanced data analysis will enable the organization to systematically prevent defects. Process intelligence also assists the organization to detect inefficiencies or waste in the SDLC that can hamper acceleration and advanced automation.

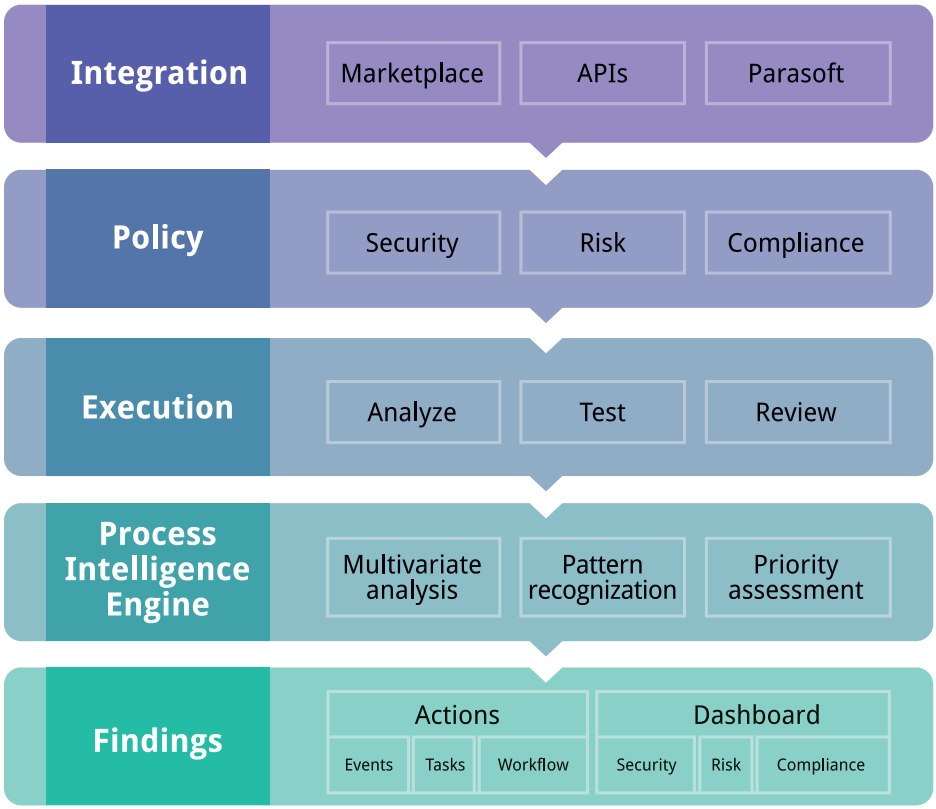
Prioritized Findings

The primary challenges associated with adopting development testing tools are managing expected outcomes and presenting information in a way that's valuable (and actionable) to managers, developers, and testers.

Throughout the SDLC, there are numerous opportunities to collect raw observations; however, there is usually very limited time to investigate, research, and remediate potential defects. A Development Testing Platform must automatically deliver prioritized findings that directly correlate to the reduction of risk. Additionally, it must be flexible enough to deliver the findings as actionable remediation tasks at the optimal stage of the project. Systems that triage results through a human reviewer cannot scale sufficiently for SDLC acceleration efforts to be effective.

Advanced analysis and testing is critical for success— yet, without a centralized process to systematically fix the defects discovered, quality practices will typically disintegrate and then resurface when the organization faces a painful or highly-publicized failure. A Development Testing Platform must make defect remediation achievable by prioritizing the actionable findings and automatically distributing tasks to the correct resource. The tasks should be accessible not only within the Development Testing Platform, but also via an open API that provides access to tasks within workflows of other, complementary process tools.

Figure 3 | Parasoft Development Testing Platform



Source: Parasoft

In addition to driving a central process for defect remediation, the Development Testing Platform should also offer actionable information to managers. Rows of data do not deliver readily-accessible analysis about risk. Data must be converted to manager-friendly dashboards that help the team make optimal trade-off decisions.

Re-Engineering the Software Quality Process

Consider this: if software quality has traditionally been a "time-boxed" exercise, then we can't possibly expect that accelerating the SDLC will yield better results from a testing perspective. If organizations want to accelerate software releases, they must reassess the current testing practices in order to keep quality as status quo. However, in order to improve software quality in conjunction with SDLC acceleration, organizations will have to truly consider re-engineering the software quality process.

Parasoft designed its Development Testing Platform (DTP) to address this need. Parasoft DTP eliminates the business risk of faulty software by consistently applying software quality practices throughout the SDLC. It enables software quality efforts to shift left—delivering a platform for automated defect prevention and the uniform measurement of risk across project teams. With seamless integration into any software development environment, enterprises can observe and collect data from any SDLC infrastructure system, including open source and third-party testing tools. Parasoft DTP allows enterprises to aggregate disparate data and apply statistical analysis techniques—transforming traditional reporting into a central system of decision.

The Parasoft Development Testing Platform:

- Provides SDLC process visibility and control across teams
- Guides developers and testers to remediate the most critical software defects
- Establishes a central control point for software defect prevention
- Prioritizes findings in order to prevent business risks
- Delivers a central system of decision for managers

Client Example: Reducing the Risks of Embedded Software

The world's leading beverage company selected Parasoft Development Testing Platform as a strategic part of its mission to ensure the continued reliability and safety of its next-generation beverage dispensers.

The embedded software driving these dispensers must run flawlessly—not only to ensure that the dispensed drinks meet safety and taste expectations, but also to collect and report data that is critical for optimizing the company's provisioning, servicing, and marketing operations. Software defects in this high-profile system would have substantial impacts on revenues, brand reputation, and legal liabilities. As a result, the organization decided to adopt Parasoft static analysis as the standard for ensuring that all code developed for this project—across numerous distributed teams, development platforms, and programming languages—satisfies corporate expectations around reliability, security, and performance.

Solution Results:

- Development and testing efforts are consistently aligned with business expectations.
- Many defects are prevented, and high-severity defects are mitigated before progressing to the next phase of the SDLC.
- Managers gain real-time risk assessments that help them make informed decisions.

"Customer safety and satisfaction are essential for protecting our brand. This proactive, policy-based approach enables us to rapidly roll out innovations that help us maintain our competitive edge."

–Senior Manager, Software Innovation

Source: Parasoft

From the Gartner Files:

Accelerate Development With Automated Testing

IT leaders need to look at software testing as more than just a way to mitigate the risks posed by bugs in an application. Implemented correctly, automated testing is an essential component to a highly productive software development organization.

Key Challenges

- An unstable and untested codebase reduces the effectiveness of developers.
- Most organizations lack a comprehensive automated test suite.
- Many attempts to build an automated test suite fail due to the slow and fragile nature of end-to-end user interface (UI)-driven tests.
- Unit tests are a necessity, but they are only part of a comprehensive test suite.

Recommendations

- Keep your codebase as defect-free as possible.
- Continuously test your codebase.
- Structure your test suite using the automated test pyramid.
- Write automated tests before the code is written.
- Take a pragmatic and incremental approach to build test suites for legacy code.

Introduction

What You Need to Know

Many organizations focus their testing efforts after the implementation phase of a project. While this ensures that the software meets some basic level of quality, delaying the testing until after the coding phase significantly increases the effort needed to fix issues that are found. Implementing continuous testing practices will enable earlier detection of defects, improve overall quality, and maximize delivery velocity. A comprehensive set of automated tests will improve development productivity by allowing for continuous testing, which will allow bugs to be found and fixed while the code is still fresh in the developer's mind. Additionally, a comprehensive test suite will ensure that the codebase stays stable at all times. This isolates the potential causes of a defect to the current changes, making defects much easier to debug.

Analysis

Keep Your Codebase Stable

One of the most valuable assets of a development organization is stable code. The traditional split of the development process into a coding phase and a testing phase results in a wasteful use of this asset.

A traditional software development effort includes a long and unpredictable testing phase after the coding phase is complete, where bugs are identified, triaged and fixed. The time frame for this testing is often compressed by schedule slips earlier in the development process. The combination of unpredictable effort and compressed time frames often results in the code being deployed with significant known and unknown defects. This results in a reduction of the product's quality and can result in development resources being pulled off of their next project to fix issues in the production code. The result is that the quality assurance (QA) organization is often pushed to rush their work and then is blamed for the software being both late and defect-ridden.

Despite the time and effort required to stabilize a codebase before deployment, the separation of the project into coding and test phases means that stability is quickly lost at the beginning of the next project. During the coding phase, multiple changes are made without a full analysis of the impact of each change on the quality and stability of the codebase. The result is that the code stability declines and it is difficult to determine the root cause of defects that are found.

The importance of keeping the code stable is increased for agile projects where the output of each iteration is potentially shippable (see Note 1). A two-to-four week iteration does not allow for time to spend weeks stabilizing the code at the end of the project. The code must be kept as stable as possible at all times.

Continuously Test Your Codebase

Optimize the use of stable code by testing continuously to keep the code stable at all times. By running a comprehensive test suite after each code change, defects are found quickly. The concept of a cost curve for fixing bugs was well documented by researchers, such as Barry Boehm and Victor Basilli in the late 1980s.[1] Some of the big advantages of finding bugs early include:

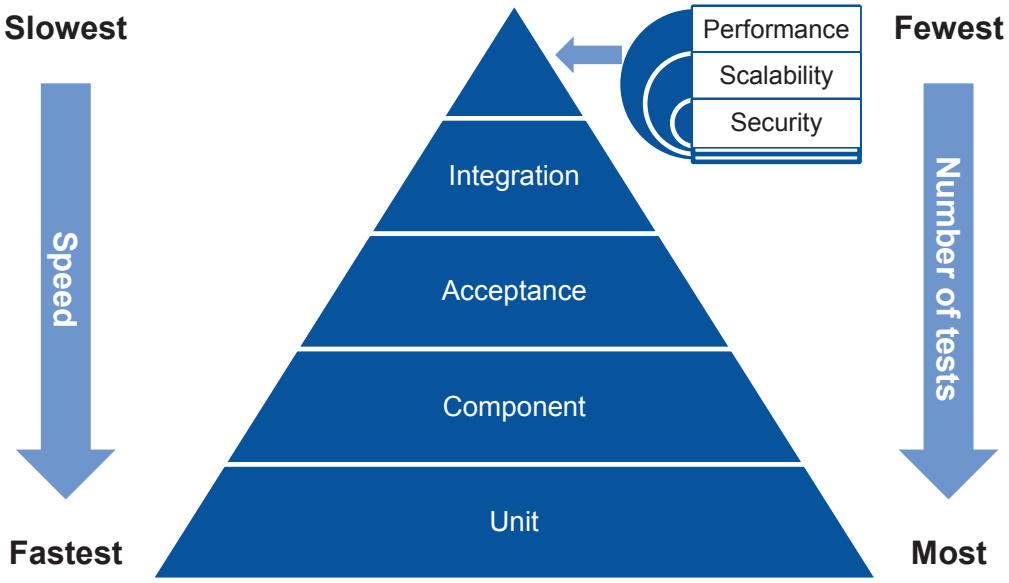
- The defect is found when the changes are isolated to a specific change set. Since it is known that the most recent change caused the defect, there is no need to analyze what change caused the defect.
- The defect is found while the change is fresh in the developer's mind. The defect can be fixed more effectively when the developer remembers the details of the change.
- The defect is found before the developer has started their next task. This prevents a time-consuming context switch to go back and work on a previous task.
- Déjà-View defects (defects that occur again after they are fixed the first time) are eliminated by adding new tests for escaped defects so that they are caught the next time.
- Refactoring is much easier in a stable codebase, since any defects introduced by the refactoring will be immediately visible.

To achieve the benefits of continuous testing listed above, it is common for a full test suite to be run many times a day. This makes it impractical to run manual tests frequently enough to keep the code continuously stable. The tests must be automated to allow for continuous testing. It is also necessary to have a continuous integration server to build and test the code after each change. The continuous integration (CI) server should also run static analysis to detect technical debt early in the process for more information on technical debt and static analysis.

Structure Tests Using the Automated Test Pyramid

Generally credited to Mick Cohn at Mountain Goat Software, the Automated Test Pyramid (see Note 2) is a way to organize your automated test suite. A well-constructed test suite will have tests for each level shown. The bottom layer is made up of a large number of very fast unit tests. Each layer above the unit test layer has a smaller number of slower tests. Organizing your tests around this model can avoid the common problem of having too many slow and fragile end-to-end integration tests (see Figure 1).

Figure 1 | Test Pyramid



Source: Gartner (December 2013)

Unit Tests

Unit testing has been widely adopted in the last 10 years, especially in agile projects. These tests are very effective at ensuring that a specific class or method is functioning correctly. In many cases, unit tests are written to test the cooperation of several classes or even major subsystems. This will make the unit tests slower, and therefore diminish the frequency that the tests are run. To be fully useful, unit tests need to be run continuously, ideally every time that a file is saved in the integrated development environment (IDE). Gartner has seen clients who have unit test suites that take several hours to run, making this impossible.

The solution is to have a full set of fast unit tests that run extremely quickly. This is done by having each test only test one method on one class and extensively using test doubles (see Note 3) to replace collaborative classes in a test.

Component Tests

These tests are designed to ensure that modules or components function correctly. The main purpose of the test is to make sure that the various classes in the service work together correctly. Test doubles and service virtualization are used to replace other services and modules. These tests should be fast and portable enough so that they can be run by developers before each code change is committed to the source code repository. Where possible, these tests should link back to the user stories or use cases that they support.

Service/API Tests

For large and complex systems, a good SOA can help to reduce the dependencies between systems and the teams working on them. To take full advantage of this architecture, a comprehensive set of API tests are essential. If these tests are written before the code, they will enable the design by contract development pattern. They will also allow teams to work independently as long as the API contract is not changed.

These tests should use service virtualization to isolate the service under the test. API tests should be very robust, even when the software is changing. The tests are an expression of the contract that the service implements, any change to the service that breaks an existing test is also likely to break production software that the service supports.

Acceptance Tests

These tests are tied to a specific user story and verify that the overall software functions correctly. These tests may be somewhat slower, but they must be portable enough to run in the developer's environment. These tests should be directly linked back to a description of the user story or test case being tested. These tests can use service virtualization to eliminate dependences on external, slow or difficult to deploy services. This test layer is a natural fit for behavior-driven development (BDD; described below).

Integration Tests

Integration tests are designed to test the interaction of modules. These are end-to-end tests that are designed to test the software in an environment that closely matches production. Because these tests use a minimum of test doubles, they tend to be slow and environment-specific. Best practice is to run integration tests continuously or nightly on dedicated test hardware or using cloud-based testing resources.

UI Tests

In most cases, acceptance and integration tests are written at the UI level. Because the tools for UI test automation are mature and widely available, it is common for an organization to do too much testing through the UI layer. These UI tests are fragile and break with minor changes to the underlying system. UI test cases should be limited to a very small set of scenarios. A best practice is to have one successful and one failure test, the various failure cases are then handled in the lower test layers.

Specialized Tests

These tests are designed to test the performance, scalability and security of the software when running in an environment as close to the production system as possible. These tests require production-like machines with large datasets and take longer to run. Because of this, they are run less often. For noncontinuous deployment models, they need to be run before each deployment.

Whenever possible, use cloud testing resources and automated performance test tools, and increase the frequency that these tests are run. While technically not an automated test, static code analysis can be used to improve quality and spot common security vulnerabilities early in the process.

In a continuous deployment model, these tests are often done using a "canary" test where a small number of servers are upgraded to the new software and closely monitored. If the changes have any impacts to the performance, stability or latency of the initial servers, the canary servers are pulled out of usage until the software can be updated and is working correctly. When the canary servers have run successfully for a predefined period of time, the rest of the servers are updated.

Table 1. Test Types

Test Type	Speed*	Execution Environment	Execution Frequency
Unit	Very Fast	IDE	Every save
Component	Fast	Developer workstation	All before commit
Acceptance	Moderate	Developer workstation	After commit/nightly**
Integration	Slow	Continuous integration server	After commit/nightly
Specialized	Very Slow	Production environment clone	As needed/before deployment

* This column indicates relative duration of a single well-constructed test. There are certainly many slow unit tests that exist today, but the best practice is to make them extremely fast.

** The acceptance test for the story being developed should be run by the developer prior to committing the change to the source repository.

Source: Gartner (December 2013)

Write Tests Before the Code

Executing manual test scripts is very time-consuming and expensive, so it makes sense in a manual testing environment to defer testing until all changes are complete. Automating tests changes this cost-benefit analysis. Since it is very quick and inexpensive to run existing automated tests, they should be written when they can provide the most benefit. There is a growing consensus in the agile community that the most beneficial time to write tests is before the code is written. A 2010 survey by Scott W. Ambler shows that test-driven development (TDD) is used by over 50% of agile teams and some form of BDD is used by over 40% of agile teams. When tests are written first, they can significantly increase developer effectiveness.

UI tests are somewhat of a special case. The popular automated UI test environments (Selenium, QTP, etc.) make it difficult, if not impossible, to write tests without having working software first. In this case, the best practice is to write the test plan first and create the automated test as soon as possible. Look for ways to reduce the number of tests written with such tools by testing as much as possible below the UI layer. UI tests also tend to be fragile, which can be reduced by good test development practices.

Test-Driven Development

The trend to write tests before code started in the unit testing practices popularized by Kent Beck and Extreme Programming. In TDD, the unit test for a class is written before the actual code. As such, it defines the contract that the code needs to fulfill. Code written to contracts specified in this way will be more focused. It will clearly meet the desired need, or the test will not succeed. A test can also keep the code simple, as there is no need to add complexity that does not appear in the contract. TDD is well established and used by many effective development organizations.

Behavior-Driven Development and Acceptance-Test-Driven Development

With the success of TDD, there is a growing understanding of the limitations of only writing the unit tests before the code. Unit tests are very low level and granular, they provide a good way to model and test classes and methods, but do not test end-to-end functionality.

BDD was developed by Dan North[2] as a technique to write higher layer tests early in the development practice at about the same time the concept of acceptance-test-driven development (ATDD) was developed. Both of these practices accomplish the same thing: getting everyone involved in a project together to write the tests before the coding begins.

Collaboratively specifying the acceptance test allows the product owner, the developer and the test resource to define very precisely what behaviors the software should have. The benefits are similar to TDD, but applied at a much less granular layer. BDD frameworks, such as Cucumber and JBehave, provide a way to write tests that can be understood by everyone in the process, including the end-user representative.

BDD is an evolving practice, and tools are emerging that enable its use at the acceptance, integration and functional test layers.

Take a Pragmatic and Incremental Approach to Build Test Suites for Legacy Code

While new projects can develop automated tests along with or before the code, most existing products do not have sufficient automated testing. It is often too expensive and time consuming to go back and write comprehensive tests for all of an organization's existing software. The best approach is to write tests for untested code just before changing the code. Triggers for writing automated tests include:

- At the start of a project, write a smoke level regression test to prevent handing a fundamentally broken build to QA. Any blocking defect that is not caught by this test should have a test added to the test immediately.
- Write API tests before any significant change to a service.
- If there is a bug in the software, write a test that fails because of the bug, then fix the code.
- Before refactoring a class or service, write tests for it to make sure that the refactoring does not introduce bugs.
- If a class or service is being extended to support new behavior, build tests that reflect existing and new behavior as part of development.
- Ensure that all new code is properly tested.
- Before starting a major project on a legacy system, consider automating the existing test scripts as an internal or outsourced project.

The result of this will be to build tests for the most active and fragile parts of the code first.

Note 1
Potentially Shippable

The agile principle that each story needs to be ready for deployment into production before the next story is started.

Note 2
Automated Test Pyramid

The automated test pyramid has shown up in several forms. It is generally credited to Mick Cohn at Mountain Goat Software. It also appears in the book Agile Testing by Lisa Crispin and Janet Gregory. All of the versions differ on the number of layers and their names, but the basic concept is consistent with the Gartner version in this research.

Note 3
Test Double

An artificial stand-in for an object. Doubles are used in automated tests to replace objects that are slow or difficult to run in a test environment. Fakes, Stubs and Mock Objects are forms of test doubles.

Evidence

[1] Boehm, Barry W. and Philip N. Papaccio. "Understanding and Controlling Software Costs," IEEE Transactions on Software Engineering, v. 14, no. 10, October 1988, pp. 1462-1477

[2] Introducing BDD

Source: Gartner Research, G00230092, Nathan Wilson, 15 May 2012

About Parasoft

Parasoft researches and develops software solutions that help organizations deliver defect-free software efficiently. By integrating Service Virtualization, Development Testing, and API testing, we reduce the time, effort, and cost of delivering secure, reliable, and compliant software. Parasoft's enterprise and embedded development solutions are the industry's most comprehensive — including static analysis, unit testing, requirements traceability, coverage analysis, functional & load testing, dev/test environment management, and more. The majority of Fortune 500 companies rely on Parasoft in order to produce top-quality software consistently and efficiently as they pursue agile, lean, DevOps, compliance, and safety-critical development initiatives.