



White Paper

Uniting Your Automated and Manual Test Efforts

Balancing automated and manual testing is one of the best ways to optimize your efforts and obtain high quality releases quickly. This Whitepaper describes best practices to effectively combine automated and manual testing practices to improve your software releases.

Introduction

Software development teams are always looking for an edge to produce features more quickly while retaining a high level of software quality. Most software quality teams realize that it takes both automated and manual test efforts to keep pace with quickening release cycles but are not sure how to get the most out of their testing efforts.

This whitepaper discusses the need for a balanced automated and manual test approach and describes how to optimize your efforts to obtain the highest quality releases in the shortest amount of time. The white paper focuses on these topics:

- Best practices for planning your automated test effort
- Best practices for planning your manual test effort
- Uniting your automated and manual test efforts
- Optimizing your test efforts during the QA cycle
- Using Retrospectives to improve future testing efforts

Best Practices for Planning Your Automated Test Effort

Many companies run their regression test cases manually, so when does it make sense to begin automating your regression test cases? In most cases, it's a good idea to consider automating your test cases when you can no longer run the regression test cases on each build created. For example, if you are doing daily or weekly builds of your code for the quality assurance team, and you cannot quickly run your full regression test cases with each build, it is time to consider automating them. When investing in automation, spend your time wisely using best practice approaches.

Best Practice 1 – Hire a Dedicated Automation Engineer

Many teams experiment with automation by trying to use an existing manual tester or programmer, then when the automation effort fails, they scratch their heads to figure out why. It's simple: an automation engineer brings years of experience that reduces re-work and is dedicated so that other manual testing and programming tasks do not interfere with their role of automating tests. If costs of another head count are an issue, consider the statistics: standard industry results show that it's 30-100 times less expensive to find defects in QA than once your software is released to customers! Most companies find that head count concerns dissipate as they start testing their applications daily and reducing QA time while improving software quality.

Best Practice 2 – Start Small by Attacking Your Smoke Tests First

Don't try to automate everything under the sun. Instead, start small – a good place to begin is by automating your smoke tests. Smoke tests are the basic tests you run on a new build to

ensure nothing major was broken with the new build. This group may include only 20 or 25 tests, but by starting with just these, you quickly to see immediate impact from your efforts.

Best Practice 3 – Automate Your Regression Tests

Once you have automated your smoke tests, move on to your regression tests. Regression tests ensure that the new build has not broken existing features. Automating your regression tests may involve automating a large number of tests, so take a methodical approach and focus on the areas of highest impact:

1. **Frequently-Performed Tests** – Start by automating the regression tests that are frequently performed. Automating a test that gets run once a release cycle isn't nearly as impactful as automating a test case that is run 100 times during a release cycle.
2. **Time-Consuming Tests** – Some tests take hours to run: they involve setting up database table entries, running user interface tests, then querying the database to ensure the data was handled correctly. When done manually, these tests can take hours, so automating these tests can free your day up for less time consuming tests.
3. **High-Precision Tests** – Look for tests that require a high degree of precision. By definition, these test cases have a low tolerance for error: if a mistake is made when running the test, you have to scrap everything and start over again. Such a test may involve complex mathematical validations, or a complex series of steps you have to follow to execute a test case that, when interrupted, forces you to start over again. Once these tests are automated, you will get more consistent results and reduce the stress of running them manually.

Best Practice 4 – Intelligently Organize Your Automated Tests Based on Project and Team Size

If you have a small team with one automation engineer, a few manual testers and a few programmers, the likelihood that you will need to split up the automation test effort between team members is small. Keep the structure simple by organizing your automation tests with folders inside a single project, with each folder housing the tests for a functional area of your software. It is also a good practice to have a “common” folder that contains common test cases that can be re-used by calling them from test cases that reside in your functional area folders. Examples of re-usable test cases are those for logging in and out of the software, sending emails, etc.

If you have multiple product lines and automation engineers, you will have issues if the automation engineers need to access test cases from within a single project because they will have concurrency and source code checkout issues. To prevent these problems, create a project suite that contains multiple projects (one for each product line, one for common tests, etc). Within each project, organize them with folders that are separated by functional area so that you can quickly find test cases that relate to areas of your software. By having multiple projects, automation engineers can check those out separately without the worry of overwriting someone else's work.

Best Practice 5 – Keep Your Tests Protected with Source Control

Ever lost your hard drive or overwritten something by mistake? We all have done this, and

recovering from it can be simple or can be impossible. By using a source control system (like Subversion, Perforce, ClearCase, TFS, etc.), you can prevent loss of data. As you make changes to your test cases, check them into your Source Control system and if you ever need to roll back to the prior version, it is simple to do.



Want to learn more about this topic? [Watch this movie on how to use your Source Control system during testing.](#)

Best Practices for Planning Your Manual Test Effort

While automated tests are great for reducing time spent running regression tests, you still need manual tests for testing new features or enhancing existing features of your software. When planning out your manual test effort, best practices dictate that you take a methodical approach that produces great results and is repeatable.

Best Practice 1 – Great Testing Starts with Great Requirements

Ever worked on a software project that spent as much time in the QA phase as it did in development? One where the end result was lots of re-work, missed deadlines and frustrated team members? Much of this re-work can be eliminated by first producing great requirements. By great requirements, we're not talking about heavy requirements that fill up bound notebooks and elicit siestas during team reviews. A good requirement has three attributes:

- Succinct yet descriptive narrative
- Explicit list of business rules
- Prototype - a mockup or wireframe of the functionality

Consider the requirement to the right - it is a poor requirement because the narrative does not really describe how the feature should work, the business rules are incomplete (they list the fields but not any behaviors of the fields) and there is no prototype associated with the requirement to provide visual and functional guidance, so it is not clear what this screen will look like once it is completed.

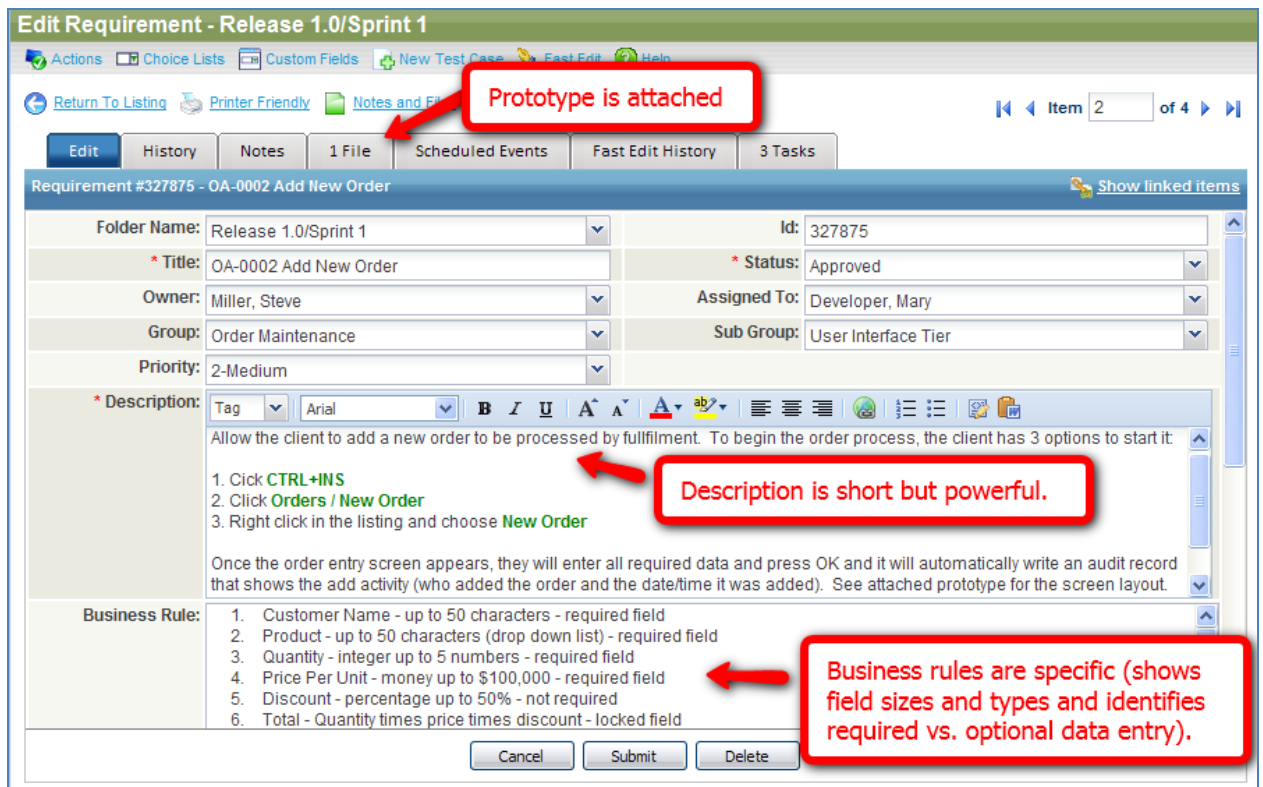
Requirement: OA-0002 Add New Order

Description: Allow adding of a customer order. Orders can come from many places. They can come from downloads from the website, Google searches, trade shows, etc. Regardless of how the order comes in, process it the same way. Invalid orders should not be processed.

Business Rules: Allow entry of these fields: Customer Name, Product, Quantity, Price Per Unit, Discount, Total, Date, Address, Credit Card Info.

So what does a better requirement look like? Consider the requirement below:

it has a very descriptive narrative, has an explicit list of business rules and has an attached prototype.



Picture above courtesy of SmartBear Software's [ALMComplete](#), a tool for managing requirements, test cases and defects.

Best Practice 2 – Create Positive and Negative Tests

When creating your test plan, ensure that you have positive test cases (those that ensure the functionality works as designed), negative test cases (those that ensure that any data entry and uncommon use issues are handled gracefully), performance test cases (to ensure that the new release performs as well as or better than the prior release), and relational tests (those that ensure referential integrity, etc.). [Click here for a detailed discussion and related best practices.](#)

Best Practice 3 – Ensure Test Cases Have Requirement Traceability

When creating your test plan, the best way to ensure you have enough test coverage for each requirement is to create a traceability matrix that shows the number of types of test cases for each requirement. By doing this, you will quickly spot requirements that do not have adequate test coverage or missing test cases.

Best Practice 4 – Publish Your Test Cases to Developers Early

Once your testers have completed their test cases, publish them to the programmers so that they can see the tests that will be run. Your programmers should review the test cases to ensure that their code will accommodate logic for each of the tests; this simple tactic will dramatically reduce re-work during the QA phase.



Want to learn more about this topic? [Watch this movie to learn suggestions on how to include developers in the test case creation process.](#)

Uniting Your Automated and Manual Test Efforts

Let's imagine that your automation engineers have automated your smoke and regression test cases and your manual test engineers have created a complete set of test cases that have great traceability and test coverage. The development team has just shipped the first version of the code to the QA team and plans to perform daily builds each day of the testing cycle. Here are some best practices to keep the QA phase operating efficiently.

Best Practice 1 – Schedule Your Automation Runs Daily During the QA Phase

Now that you have set up your automation test cases, it is important to run them each day so that you can quickly discover if the new build has broken any of the existing functionality.

When doing this, there are a couple of approaches you can take. If your builds are being done by a continuous integration tool (such as [Automated Build Studio](#), Hudson, Cruise Control, etc.), then you can launch your automated tests from the continuous integration tool.

If you are doing builds manually or if you prefer to have the automation launch at a specific time, you can schedule them to launch using a scheduling tool such as SmartBear's Software Application Lifecycle Management (ALM) tool, [ALMComplete](#).

Automation Schedules

[Return To Listing](#)

* Automation Type:	TestComplete														
* Host Name:	SM-Laptop														
* Project Suite:	Orders App Testing														
* Project:	Regression														
* Time Run:	6:00 PM														
* Date Range:	02/03/2011														
* Run Every:	<table border="1"> <thead> <tr> <th>Mon</th> <th>Tue</th> <th>Wed</th> <th>Thu</th> <th>Fri</th> <th>Sat</th> <th>Sun</th> </tr> </thead> <tbody> <tr> <td><input checked="" type="checkbox"/></td> <td><input checked="" type="checkbox"/></td> <td><input checked="" type="checkbox"/></td> <td><input checked="" type="checkbox"/></td> <td><input checked="" type="checkbox"/></td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> </tbody> </table>	Mon	Tue	Wed	Thu	Fri	Sat	Sun	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Mon	Tue	Wed	Thu	Fri	Sat	Sun									
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>									

A good scheduling tool should be able to launch the automated tests on a specific machine at specific times each day of the week and then log the results of the test run on dashboards, so that you can easily see how many automated tests ran, how many passed and how many failed. You also want to be able to see which tests failed, so your programmers can check the code to fix any issues the new build caused.

Best Practice 2 – Create Reproducible Defects

Nothing drains time in QA like reporting defects that are not reproducible. Each time a tester reports a defect that is not reproducible, it takes more time for the programmer to report that it is not reproducible, time for the tester to re-document how to reproduce it, and more time for the programmer to try again.



So how do we solve this? The best way is to publish a narrated movie that shows what you did to reproduce it. Do this with a free product called Jing (<http://www.jingproject.com>) that allows you to highlight your application, record the keystrokes, then produce a movie (with narration if you narrate it with a headset) that is accessible via a URL. Include the URL of the movie on the defect you send to the programmer and the programmer has everything they need to see your defect in action!



Want to learn more about this topic? [Watch this movie to learn how to unite your manual and automated test efforts.](#)

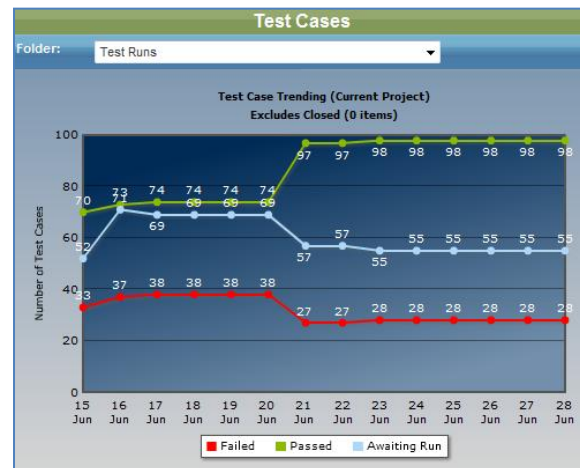
Optimizing Your Test Efforts during the QA Cycle

During the QA phase, it is important to meet daily as a team for 15 minutes (referred to by Agile shops as a Daily Scrum Meeting) to assess your test progression and to prioritize defects so that the most important ones are addressed. If your test management and defect tracking tools have dashboards that show test progression, make use of those during your meeting by presenting them interactively during your meeting (use a projector or online conferencing to review those as a team).

Best Practice 1 – Review Test Case Progression

The first indicator you should review is how much progress the QA team is making towards running all the test cases for the release.

To the right is an example of a dashboard that shows day-by-day how many test cases are run, how many passed, how many failed and how many are still awaiting run. A large number of failed test cases signal a quality problem. If you find that test cases are not being executed at a pace that allow you to finish all tests within your QA window, this knowledge allows you to adjust by adding more help or extending working hours to get it done.

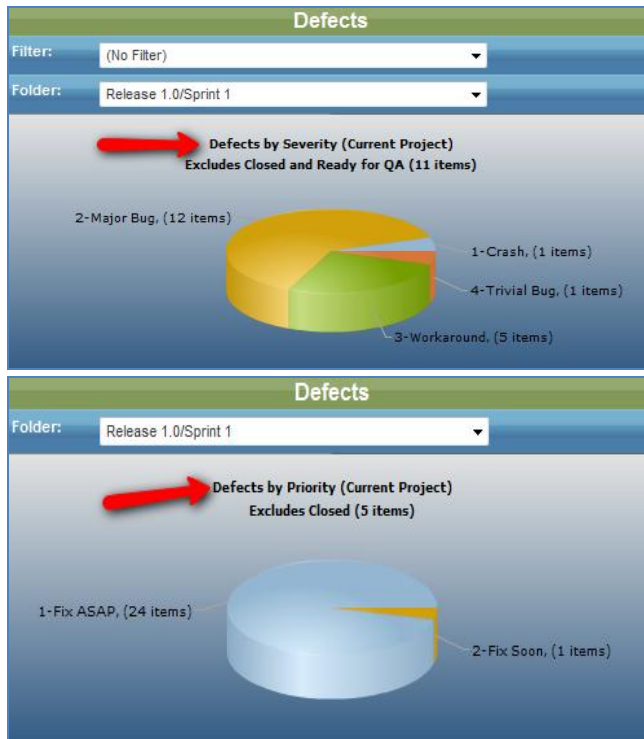


Best Practice 2 – Prioritize Defects Daily

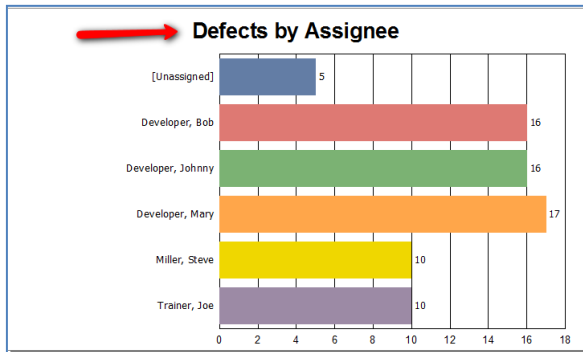
The next indicator to review is the number of defects by priority and assignee. This information allows you to determine if specific programmers are overloaded with defect work and helps you to more evenly distribute the load. When prioritizing defects, we like to prioritize them based on severity and how important they are to the software release. The key to this tactic is to objectively define your severity levels so that it is clear how problematic they are. We use these severity levels:

- 1-Crash (crashes the software)
- 2-Major Bug (with no workaround)
- 3-Workaround (major defect with a workaround)
- 4-Trivial Bug

Based on these severities, you can choose what priority they should be fixed in (1-Fix ASAP, 2-Fix Soon, 3-Fix If Time). Below are some dashboards you might consider when evaluating defect priorities:



In the graphs above, you can see that most defects are major with no workaround, which implies a quality issue. You will also see that too many defects are categorized as high priority, which means that your team needs to make tougher decisions on how your prioritize them to ensure that the most important ones are fixed first.



Evaluating defects by assignee can indicate if a specific programmer is overloaded with work.



Want to learn more about this topic? [Watch this movie to identify ways to optimize your testing effort during QA cycles.](#)

Using Retrospectives to Improve Future Testing Efforts

Once your software release makes it to production, it is important to look back at the things you did right and the things you can improve upon so that you can take these “lessons learned” into the next development effort you embark on. This approach is sometimes referred to as a “post mortem” or “retrospective”.

Best Practice 1 – Analyze Your Project Variances

If you used tools to plan out your work efforts (programming and testing hours for each requirement) and you recorded the time each person spent on each requirement, you then have valuable information on how well your team is able to estimate tasks. Software Planner and many other project management tools have the ability to track estimated versus actual hours. Using these tools, you can capture the release analytics:

Sprint	Est Hrs	Act Hrs	Variance	% Variance
Release 9.0 – Sprint 1	441	586	145	32%
Release 9.0 – Sprint2	655	548	-107	-19%
Release 9.0 – Sprint 3	881	740	-141	-19%
Release 9.0 – Sprint 4	636	698	62	10%
Averages	653	643	-10	-1%

In the example above, notice that the first sprint was under-estimated. In sprint 2, a correction

was made (based on the retrospective) so that estimated hours were buffered and the end result was that the sprint came in under variance. By collecting the variance information release-by-release and sprint-by-sprint (if using Agile), you can make adjustments by buffering estimates in upcoming releases.

Best Practice 2 – Analyze Quality Assurance Metrics

It is also important to track how many test cases were run and how many defects were discovered during the release. Below is an example of how we tracked this information for a prior release:

Sprint	# Test Cases Run	# Defects Found/Fixed
Release 9.0 – Sprint 1	107	15
Release 9.0 – Sprint 2	129	217
Release 9.0 – Sprint 3	172	239
Release 9.0 – Sprint 4	79	533
Total	488	1,004

Best Practice 3 – Document Your Retrospective

Equipped with the analytics discussed above in best practices 1 and 2, you are now ready to hold your Retrospective meeting. Schedule the meeting and ask everyone who participated in the software release to bring into the meeting a list of three things they think the team did well and three things the team can improve upon.

During the retrospective meeting, go around the room and have each person discuss the three things done wrong and right. As each person presents their opinions, you will start to see commonalities (people will agree on the things done right and wrong), tally up the commonalities and score them.

Once done, analyze the things the team thought could be improved on and create action items (assigned to specific people) to follow up with a plan to improve those things in the next release. Once done, document your retrospective (using MS Word or something similar) and publish your retrospective in a central repository so that team members can review it in the future. If using Software Planner, you can use the Shared Documents area to store the retrospective and all team members can access it.

When you begin the planning for your next release, be sure to access the retrospective to refresh your memory on the things done well (so that you can continue to do them well) and to remind yourself of the action items taken to improve the things that were not done well.

[If you would like an example of a retrospective document, download it here.](#)



Want to learn more about this topic? [Watch this movie to learn how to use retrospectives to improve future testing efforts.](#)

Summary

So now you are armed with an arsenal of best practices to get the most out of your testing efforts. An important part of process improvement is using tools that make this process easier. In many of the examples of this whitepaper, you saw dashboard graphs that highlight important metrics your team needs to work most efficiently. Those dashboards were produced by ALMComplete, our ALM tool.

For more information on [ALMComplete](#), please visit us at www.SmartBear.com or contact us at 978-236-7900 for a free trial or a personalize demo.

You may also enjoy these other resources in the SmartBear Software Quality Series:

- [11 Best Practices for Peer Code Review](#)
- [6 Tips to Get Started with Automated Testing](#)

Be Smart and join our growing community of over 100,000 development, QA and IT professionals in 90 countries at (www.smartbear.com/community/resources/).

About SmartBear Software

SmartBear Software provides enterprise-class yet affordable tools for development teams that care about software quality and performance. Our collaboration, performance profiling, and testing tools help more than 100,000 developers and testers build some of the best software applications and websites in the world. Our users can be found in small businesses, Fortune 100 companies, and government agencies.

SmartBear Software
+ 1 978.236.7900

www.smartbear.com