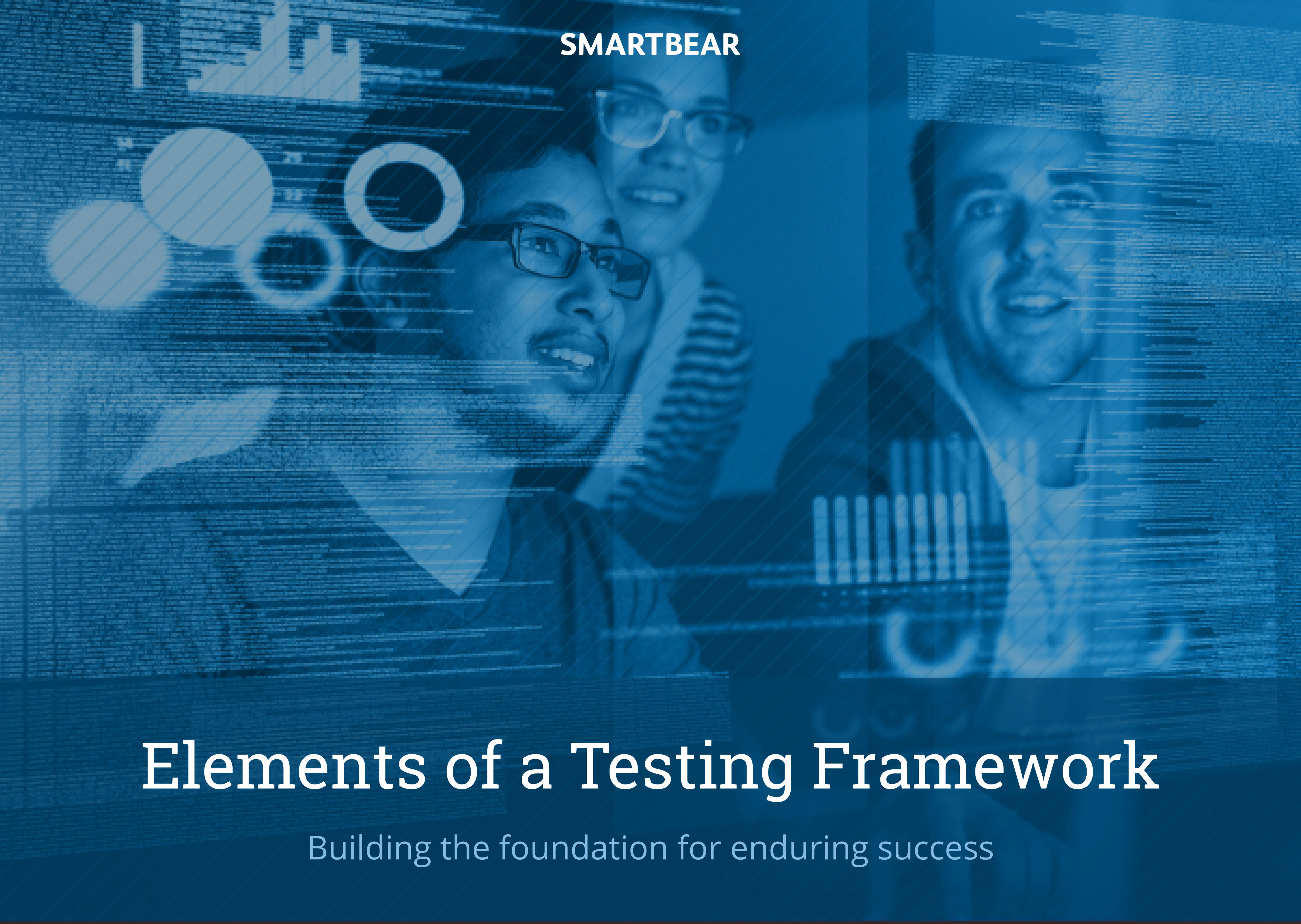
SMARTBEAR



Elements of a Testing Framework

Building the foundation for enduring success

TestComplete QAComplete

Content

Setting up testing frameworks is one of the most challenging parts of the software delivery lifecycle. It is a time-consuming and complex process. Building the right one however, can help teams test more efficiently, reduce test design and maintenance efforts, and provide a higher return on investment (ROI) for those looking to optimize their processes.

Teams today are being tasked with releasing software faster than ever while simultaneously improving the quality. Utilizing a well-structured framework enables teams to achieve both goals by speeding up test creation, expanding test coverage, and providing code that is reusable and easier to maintain. We understand though, that getting started is a painful process. That's why we created this eBook. We want to help teams like yours build a successful testing framework by providing the essential pieces you need.

In this eBook, we'll go back to the basics and cover:

- The Definition of a Testing Framework
- Defining the Important Components of the SDLC
- How to Make Informed Decisions About Your Testing Suites
- The Key Elements Vital To Every Framework

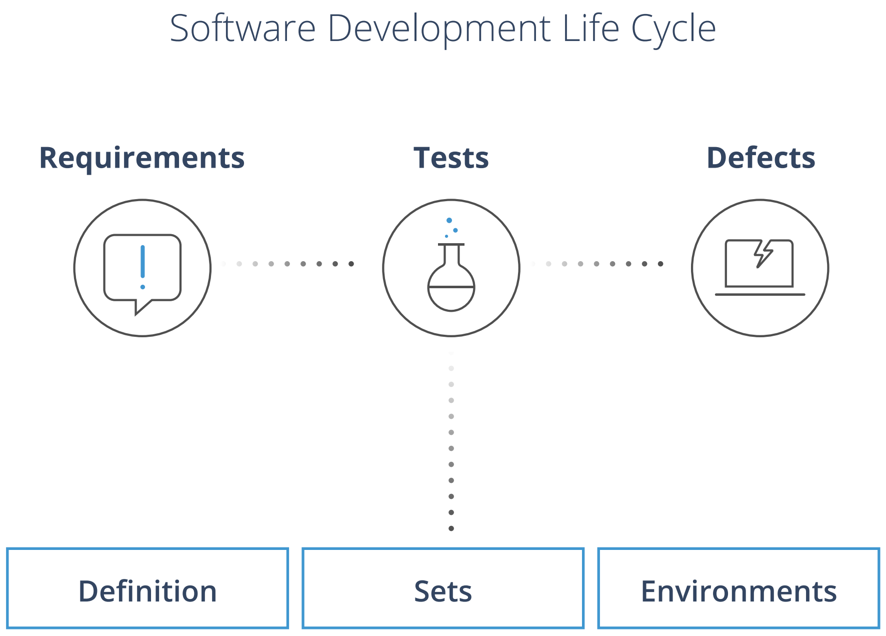
What is a Testing Framework?

The SDLC Mind Frame

Before diving into how to build a framework and the top tips and tricks you'll need, let's head back to the beginning and review the basic building blocks of the software development lifecycle: requirements, tests, defects, and releases.

Requirements in the development lifecycle are the descriptions that indicate what an application is supposed to do. They are the starting point for any team looking to build software. To lay out your requirements, you should ask yourself the following questions:

- What do you ***want*** to make?
- What do we ***need*** to make that happen?
- ***How*** should it behave?



Is it a form that needs to be filled out? Maybe it's a mobile game? Regardless of the type of software you're building, requirements are vital to the success of your product. It's where both your application and framework start to get built, so it's important to spend the time mapping them out.

The second piece of the lifecycle are the tests, which can be broken down into three pieces themselves: definitions, sets, and environments. Once you know what you want to build and how to build it, you'll want to ensure it behaves as expected. For each test, this will be outlined by a test definition. This could be applied to a single test, but you'll probably also have test sets with multiple definitions therein. Finally, test environments will outline where you're running your tests, from browser types to operating systems, resolutions, and any combination of the three (i.e., Chrome, iOS etc.).

Finally, once you've defined your tests and run them, you'll encounter those pesky little issues known as defects. As proactive as we try to be in preventing them, they are inevitable. Your test will find a bug and when you've reached that point, it will be crucial that you're able to log it properly and trace it back to the correct requirement.

This SDLC mindset is essential to have in place prior to creating your testing framework. It's critical that you're able to trace defects post-test execution back to requirements. Developers will be able to fix them faster, so you can retest sooner, and it will allow you to label requirements as "broken as designed." This in turn directs the design of the new tests, thereby ensuring proper test coverage and reporting for all changes made. Finally, you'll be able to list all of the changes made to your requirements with each build.

Any great framework can build your test plan for you from this list, as it links requirements with their tests. So, what is a testing framework?

The Definition of a Testing Framework

A testing framework is a set of guidelines used for creating and building test cases. It's the combination of practices and tools that are designed to help teams test more efficiently. These could include coding standards, methods for handling test data, or a reporting structure for the application under test.

Utilizing a framework that standardizes processes like these, enables teams to rapidly create tests and will provide more test coverage. The code developed will also have a higher degree of reusability due to the necessary step of separating test data from logic. This practice is vital to building a successful framework and we'll cover the array of pieces you'll want to separate later on.

How to Build a Test Framework: The 3 D's

Now that we've reviewed the software delivery lifecycle and the definition of a framework, we'll walk through how to build one with an example from a recent SmartBear project.

1. Define

The first step in building your framework ties back to the beginning of your SDLC – your requirements. As mentioned earlier, each requirement should have at least one test case. When building your framework, you'll start by identifying and detailing the common workflows throughout the application under test. If you're using user stories, or the high-level description of various software features from the end-user perspective, you're already a few steps ahead.

In our example, our goal was to build an ROI calculator on a new website for our test management tool, QAComplete. First, we needed to define the requirements. In doing so, we asked ourselves who would be visiting the site and the calculator, and why would they be using it? User intent indicates they were probably a prospect interested in the product and were trying to determine whether or not QAComplete was the right investment for their team.
If you're already familiar with user stories, the prospect in this example equates to the 'role' in user story.

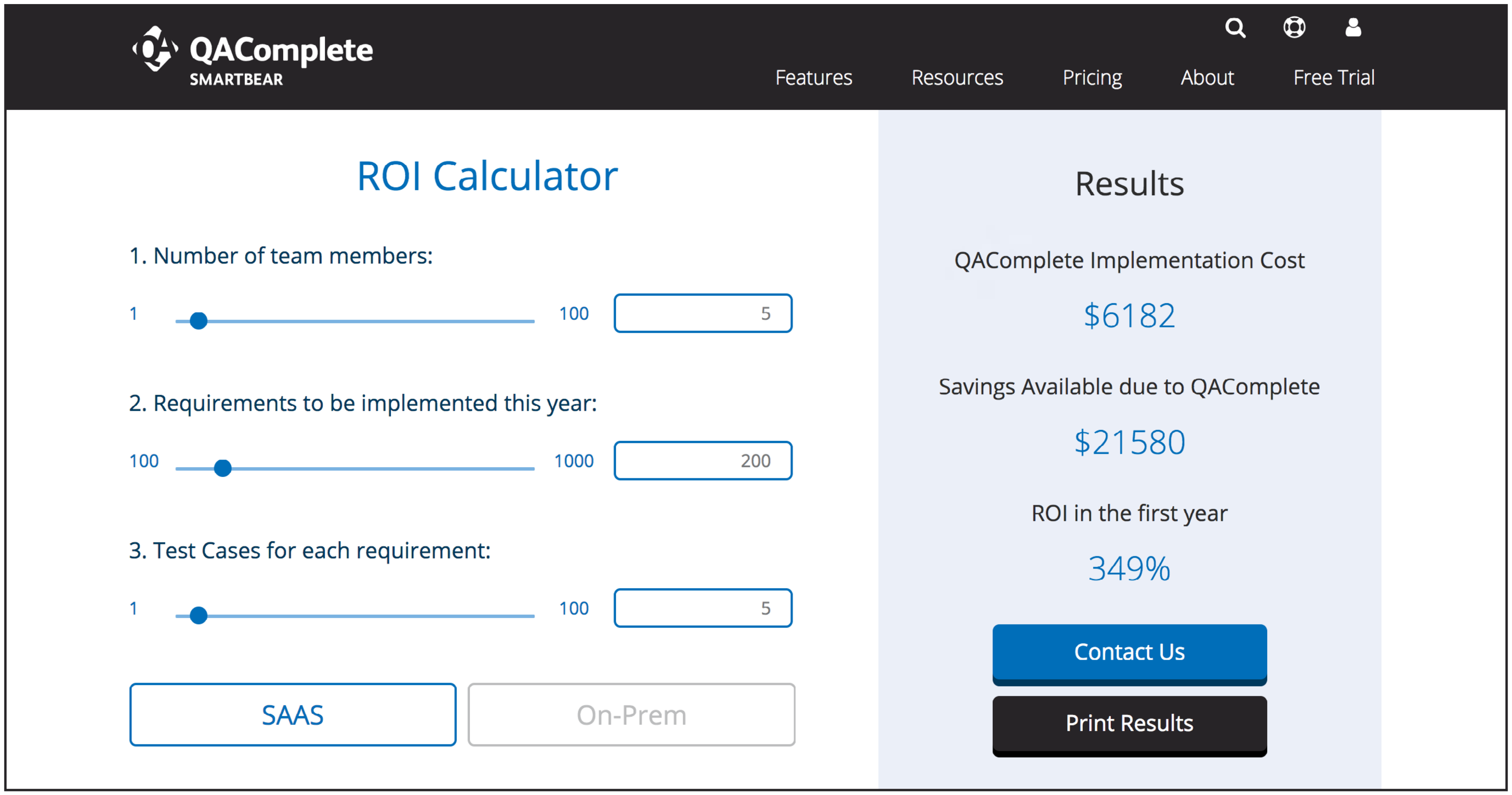
Perspective, which tells you the 'why,' or explains the reasoning for a user action, is often missing in frameworks. Utilizing user stories and defining the 'role' will fill in gaps in information and highlight unwritten requirements.

Second, we asked ourselves where the calculator would live and if it would require a new environment. In this case, it was going to live in a web browser. However, we needed to take this a step further. Yes, we needed a web browser, but was it going to be a mobile browser? How were people going to consume it? Where would users be accessing this calculator?

The key in defining your workflows is to ask yourself those same questions.

- Who is using my application?
- Why are they using it?
- What context are they going to be using this application in?

Take a look at the picture below. In this calculator, there are four inputs and three pieces of data, or outputs, revealing the ROI of buying QAComplete. There are also multiple ways to consume that information – a 'print' button and a 'contact us' button.



ROI Calculator

1. Number of team members: 1 — 100 | 5

2. Requirements to be implemented this year: 100 — 1000 | 200

3. Test Cases for each requirement: 1 — 100 | 5

SAAS | On-Prem

Results

QAComplete Implementation Cost
$6182

Savings Available due to QAComplete
$21580

ROI in the first year
349%

Each output meets our general requirement of providing an ROI and we purposefully designed the calculator with a limited number of steps to cut back on variability. Our desired user path (with minimal variation) looked like the following:

1. User selects values for the 4 inputs or leaves the defaults
2. User reviews the 3 pieces of returned data
3. User clicks 'Print'
4. User clicks 'Contact Us'

This is where our test workflow started. Now that we have this, we can break it down further into the individual actions.

2. Decompose

The next step in building your testing framework is decomposing your workflow into atomic pieces, or the small, separate actions associated with each step. Keep in mind this is not referring to units, or unit testing. Unit testing is designed to test specific or niche functions to ensure they behave per your requirements. In this stage, we're looking at it from the user perspective and not taking the backstage actions that are triggered into consideration. You'll break down your workflow in a similar manner, which we outline below:

- Actions that provide an input
- Results of those actions
- Workflow logic
- Exit points from the workflow

Take a look at our ROI calculator again. We've highlighted our inputs, outputs, and exits. In this example, we had multiple types of inputs, each of which could be entered in one of two ways (i.e., sliding or by typing), as well as the ability to toggle back and forth between the two. The calculator also has three outputs and two exit strategies.

Getting to this level of granularity does require in-depth knowledge of the application under test, but is vital to the decomposing stage of building your framework. You've now framed the journey you want your users to embark on.

Inputs

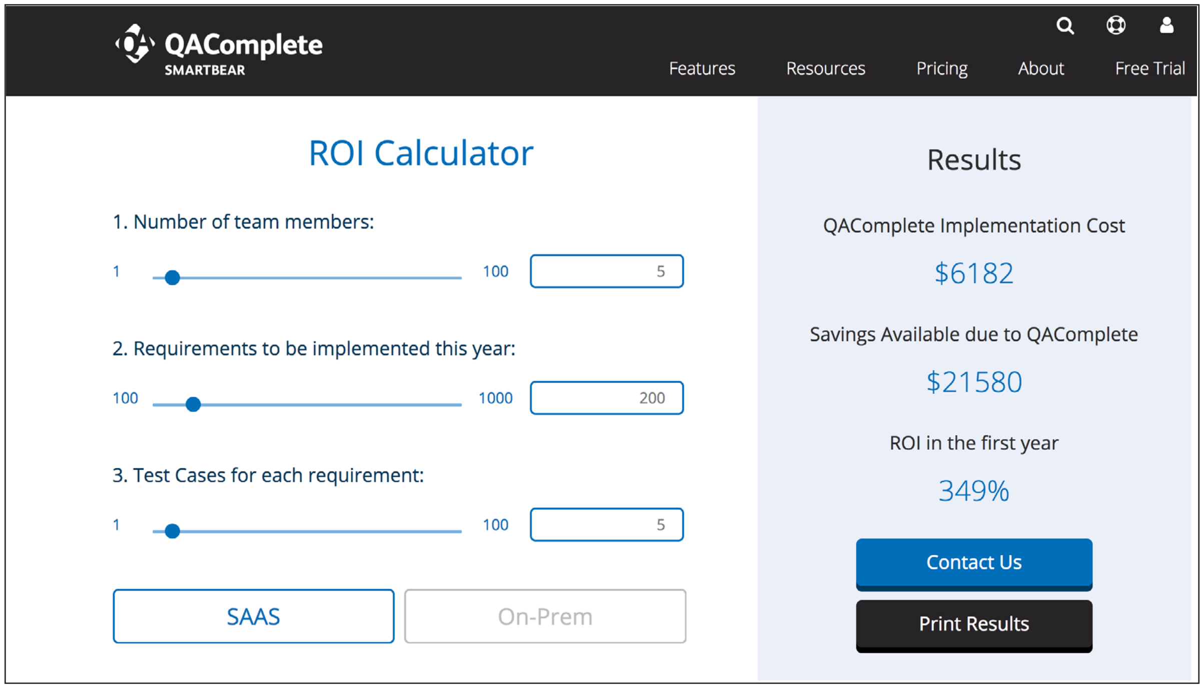
1. User selects or types in a value for team members
2. User selects or types in a value for requirements to be implemented this year
3. User selects or types in a value for test cases per requirements
4. User leaves SaaS as a selection, or switches to On-Premise, or can switch back and forth

Outputs

5. QAComplete Implementation cost in USD
6. Savings available due to QAComplete in USD
7. ROI in the first year as a percent

Exits

8. User can click Contact Us link
9. User can click Print, which will provide a printable version of 1-7



ROI Calculator

1. Number of team members: 1 – 100: 5

2. Requirements to be implemented this year: 100 – 1000: 200

3. Test Cases for each requirement: 1 – 100: 5

SAAS | On-Prem

Results

QAComplete Implementation Cost
$6182

Savings Available due to QAComplete
$21580

ROI in the first year
349%

3. Decide

The third stage in building your framework is decision making. This can be a tricky step as there are a few choices you'll need to make – which environments you'll use, the steps that can be repeated, and which parts should be automated.

Environments

We briefly touched on environments earlier in the eBook, but now is the time to decide which ones to use and what contexts to cover. You'll want to take a look at the following six items:

- Operating systems
- Browsers
- Devices (Mobile & Tablets)
- Security Permissions
- User Roles
- Conflicting Software

The internet of things (IoT) is a vast network of software. Depending on the application you're building, you could have operating systems all over the world, browsers that are constantly being updated, and devices that will be outdated with the next Samsung or Apple release.

What if you're working on a banking application or software for a government contract? You'll have to decide on security permissions and which user roles tie into different permission types. Last, you'll have to decide if your application should work with conflicting software, such as antivirus or antimalware.

Repeatable Tasks and Steps

After deciding which environments you'll use to run your tests in, you'll have to decide which tasks or steps need to be repeated. Frameworks enable a high degree of reusability among scripts and in our example, we had two –the workflow in which a user selects the slider, and the workflow in which the user opted to type in their numbers.

Once you've decomposed your major workflows, finding commonly reused steps should be much more straightforward. They'll typically be actions or checks that use the same data.

Automation

Choosing what to automate is complex and there are very few tasks or actions that have a clear-cut answer. Typically, these are actions that can be tedious and time-consuming for humans, which also makes them prone to error, such as data entry, environment set up or tear down, and cleaning up databases. Other tasks that humans can't implement well, such as testing screen responsiveness, are great for automation. It's simply impossible to have a tester sitting in front of your application with a timer, checking to make sure your mobile app works in sub one second timeframes. Tasks like these are often repeated and their implementation processes reusable. Both you and your team would benefit by automating them due to the increased speed and accuracy a machine could provide.

Non-functional testing types such as performance and load testing are also great candidates for automation. They're used to estimate how an application would handle real user scenarios and emulate bandwidth and are vital to understanding where the breaking points of an application lie. Fixing performance and load issues takes not only time, but may require additional fixes beyond code changes.

Finally, report generation should be automated. Eliminating the need to manually generate reports that capture and consolidate test data reduces the possibility of errors and maximizes the time you have to analyze it. Generating reports monthly, weekly, and even daily can be a hassle, but with the right tool, you would be able to not only automate the report itself, but automate the dissemination process as well.

When NOT to Automate

While automation can speed up testing cycles and improve coverage, not everything should be automated. A question that pops up frequently is, "why not?" The answer is that there are actions that are better left to a manual process, as counterintuitive as that sounds.

GUI testing is complex, and while much of it can be scripted, machines have a difficult time understanding the end-user perspective. Do the colors on your web page clash? Is the text hard to read on the current background color? Is the workflow user-friendly? As long as humans interact with machines and with the applications you're developing, you'll need to conduct some form of manual testing to find defects like these.

Elements of a Testing Framework

Now that we've covered what a testing framework is, the mind frame you need to get started, and the steps needed to build one, let's dive into the pieces of the framework itself.

1. Library

The first part of your framework is your library. This is a repository of all your decomposed scripts. These should be separated into their components as you'll need to start with parts that are easy to trace and easy to build into your framework.

2. Test Data Sources

Not only will you want to separate each of your decomposed scripts into their individual components, you'll want to separate your scripts from the data. By separating your test script logic from your data, you will ensure your scripts are reusable and easy to maintain.

With data that is hardcoded into scripts, you're forced to rewrite the script when changes are made – otherwise, the test will fail. Ideally, for every set of inputs and actual results, you have at least one data source.

3. Test Environments

Next, you'll want to define the list of the environments you'll want to cover, broken down by type (i.e., operating system, browser, device). The number of devices and systems available in the next few years is only going to grow exponentially, so it's crucial to take these groupings into consideration from the very beginning of your framework building process.

4. Helper Functions

Combining your test environments with your helper functions will be the key to your success. Helper functions are scripts such as setup scripts, cleanup scripts used to scrub the databases, or scripts you use to access external information. These are the catchall for anything that isn't a user action and can save you plenty of time. For example, you'll want a setup script for a Mac device that's working with Safari, as well as a script for a Windows machine that works with Internet Explorer.

5. Modules

Once you've finalized the first four pieces of your framework, you're ready to start building the bigger parts – the modules. Each module is a combination of library items, with their helper functions, environments, and data sources, that together highlight a single product capability. Modules laid out this way will enable you to link defects more easily back to specific requirements.

Using our previous ROI calculator example, our modules looked like this:

- Workflow 1 inputs "ROI slider" using Data Source A
- ROI Calculator Checks using Data Source A
- Workflow 2 inputs "ROI type in" using Data Source A
- ROI Calculator Checks using Data Source A
- Repeat for Covered Environments: IE, Chrome, Firefox, etc.

6. Structure & Hierarchies

To finalize your framework, you'll want to develop a folder structure for your modules, in a similar manner in which you utilize dividers in a binder, or with sub categories for your inbox in your email. Each folder represents the parent / child relationships you can use. Your overarching structure should represent the format of the application under test.

Having these structures in place will enable you to track defects faster, trace them to requirements, and tie them to releases – which will pay off in the long run.

Conclusion: Benefits of a Testing Framework

A well-developed testing framework today is vital to any modern development cycle. It ensures a higher degree of agility for your team, helps you test faster, and can even enable your developers to fix bugs quicker, meaning you can retest sooner. It will help you not only expand test coverage, but ensure you're getting the right test coverage, while still allowing you to test across the board without sacrificing quality.

The structure we laid out in this eBook, from starting with a SDLC mind frame, to the steps needed to build a framework and the elements themselves, allows you to find the right balance needed to reach your goals - faster delivery cycles and higher quality software.

QAComplete

The Most Customizable
Test Case Management Tool

Get Started

TestComplete

Functional Test Automation
for Desktop, Mobile and Web

Get Started

SMARTBEAR

The Leader In Software Quality Tools For Teams

Our unique approach infuses quality and speed into your entire software development lifecycle, so teams can work together to create the best software on the planet, faster than ever.

- Accelerate Dev/Test/Ops Workflows
- Improve Quality at Every Stage
- Reduce Complexity
- Gain End-to-End Visibility
- Improve Release Confidence
- Reduce Costs

6.5M+
Users

194
Countries

22K+
Companies

Get Started

www.SmartBear.com

@SmartBear

SMARTBEAR

TestComplete QAComplete