STAT COE-Report-02-2017

Automated Software Testing Practices and Pitfalls

Authored by: Thomas Pestak
William Rowell, PhD

April 2017

Revised 30 September 2018

The goal of the STAT COE is to assist in developing rigorous, defensible test strategies to more effectively quantify and characterize system performance and provide information that reduces risk. This and other COE products are available at www.afit.edu/STAT.

STAT Center of Excellence
2950 Hobson Way – Wright-Patterson AFB, OH 45433

Table of Contents

Revision 1, 30 Sep 2018: formatting and minor edits.

Executive Summary

This document focuses on automated software testing (abbreviated AST in this document, but not to be confused with the Association for Software Testing). As the name suggests, automated software testing means nothing more than automating any part of the testing of software at any stage in the software development process. Any software testing that can be automated can be tested manually. The goal of AST is the same as the goal of automation in a production line: to optimize throughput and quality by improving the speed of each stage and the repeatability of each process. In the world of software testing, automation can reduce the time it takes to uncover design flaws or trace bugs. It can also improve software quality by reducing uncertainty. This is accomplished by checking a greater percentage of the software or system under test (SUT) for errors (increased coverage), especially by way of negative testing. In many cases, AST frees up human testers to focus on manual exploratory (and context-specific) testing, which can be better suited to finding faults.

This document seeks to provide information and insight into the planning, architectures or implementations, and test design strategies for AST, and to describe how AST folds into the larger issue of software economics. AST can be as simple or complex as the SUT it is testing, and while the goals may seem obvious and the benefits guaranteed, successful AST is difficult to achieve for many reasons. At the risk of sometimes reading like a "worst practice," this document seeks to shed light on the pitfalls to successful implementation of AST, many of which are non-technical in nature. In general, the AST exemplars aligned resources and training with the goals of the business case. Many of the success stories were sandboxed pilot programs and/or the second time trying, using lessons learned from previous attempts.

Background

The First Programmable Computer

It is worthwhile to understand how software testing and AST fit into a larger vignette. In 1943, the Mark 1 Colossus computer became operational. This would be the world's first fully operational programmable digital computer and had five parallel processors capable of performing digital logic. The motivation for the Colossus was to automate W.T. Tutte's "Statistical Method" which could solve for the starting wheel positions of the German Tunny Lorenz cipher in World War II (Gannon, 2006). Knowledge of the cipher's starting wheel positions was needed to decrypt each message, and since the Germans changed the starting wheel positions each day, automating the "Statistical Method" was necessary to stay ahead of the Tunny machine (Copeland, 2008). Programmable computers then, like now, are tasked with **automating mathematical operations**, as shown in Figure 1. The Colossus computer was programmed not with software, but with a panel of switches and jacks that would set the algorithms to be used. The first computer to execute software instructions was the ENIAC just a few years later in 1948. Before the ENIAC was even finished, its engineers realized that the task of creating and setting up new calculations was difficult, error prone, and time consuming. These engineers needed a way to automate the hardware using controllers that could read and execute instruction sets from a computer program. One year after the initial ENIAC was released, a modified version was able to run computer programs. The task of the ENIAC was to perform Monte-Carlo simulations of neutron decay during nuclear fission. Today we refer to these computer programs and their associated non-executable data as *software*. Software then, like now, is tasked with **automating programmable computers**.

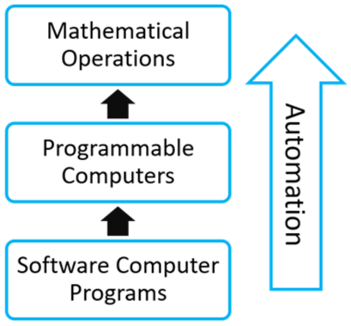


Figure 1: Automation in computing

Automation in Computing

The essence of computing is automation. But that automation is not limited to the execution realm (like the control of hardware from instruction sets or the inversion of matrices); it is also used in building and testing. As computer hardware was built less by hand soldering wires or installing vacuum tubes and more by semiconductor foundries, automation became mandatory for testing hardware designs. Figure 2 describes the process of test generation to expose faults.

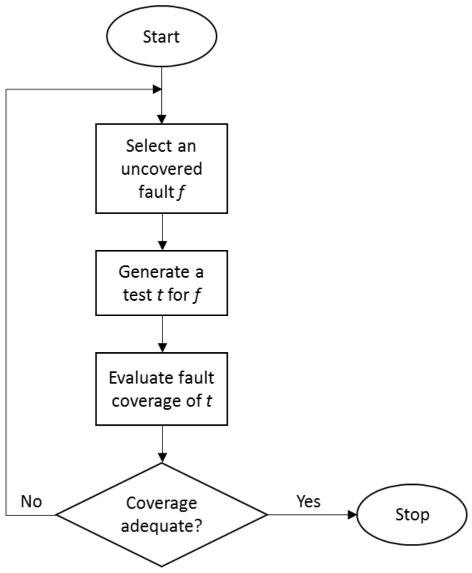

Start
Select an uncovered fault *f*
Generate a test *t* for *f*
Evaluate fault coverage of *t*
Coverage adequate?
No
Yes
Stop

Figure 2: Generic test generation procedure (adapted from Murray and Hayes, 1996)

Similarly, as software design has grown in complexity, scale, and levels of abstraction, incorporating automation into the testing of software has become an attractive endeavor. Today, a portion of software testing is performed automatically by syntax checkers and compile-time operations performed by compilers. But the phrase "software testing" generally refers to a human tester performing run-time testing for errors to **validate** and **verify** the correct operation of the software.

Validation and Verification

Validation is defined as: "*Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled [ISO 9000]"* (Standard Glossary of Terms Used in Software Engineering, 2011).

Verification is defined as: "*Confirmation by examination and through provision of objective evidence that specified requirements have been fulfilled [ISO 9000]*" (Standard Glossary of Terms Used in Software Engineering, 2011).

The wording in the ISO 9000 seems like a minor distinction without a difference. Williams uses the following questions to illustrate the difference between validation and verification. Validation asks the question "Are we building the correct product?" while verification asks the question "Are we building the product correctly?" (Williams, 2016).

Oracles

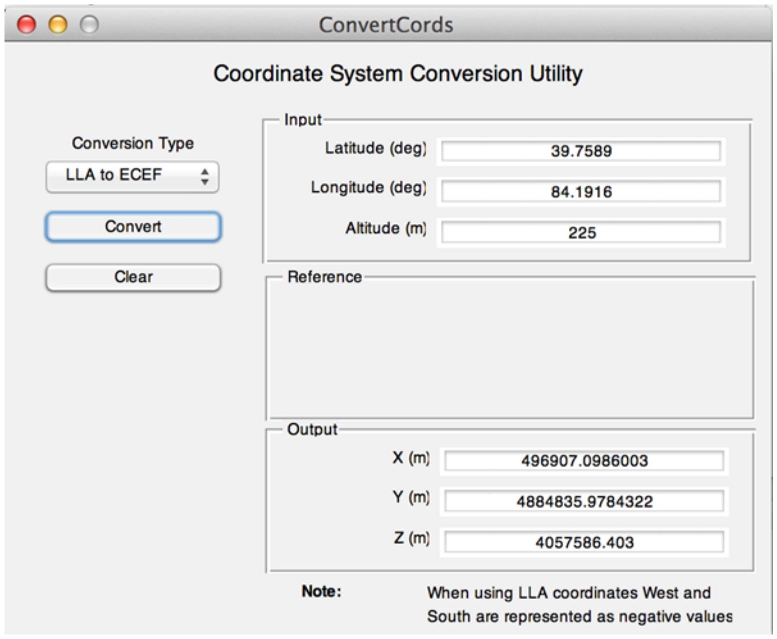
Testing for validation should confirm that the software contains the feature set and operates according to the requirements established before development began. In general, most types of testing used in validation are **positive tests**. In software testing, a positive test is defined as a test where valid input is provided and the observed results are compared to the output from an **oracle**. An oracle is a mechanism for determining whether a program has passed or failed a test (Kaner, 2005). Examples of common oracles include:

- Specifications and documentation
- Other products (for instance, a second program that uses a different algorithm to evaluate the same mathematical expression as the product under test and is considered an accurate source of correct behavior)
- Heuristic oracle that provides approximate results or exact results for a set of a few test inputs
- Statistical oracle that uses statistical characteristics
- Consistency oracle that compares the results of one test execution to another for similarity
- Model-based oracle that uses the same model to generate and verify system behavior
- Human oracle (i.e. the correctness of the system under test is determined by manual analysis)

Positive Testing

Consider a simple application that converts 3-dimensional coordinates between different coordinate frames. A positive test would be to type valid coordinate data into the input text fields, click a button labeled “Convert,” and observe whether the values in the output text fields match that given by the oracle.

In this case, the positive test passes if the observed behavior is as intended. Did the application convert from Latitude, Longitude, Altitude (WGS84 Coordinate Frame) to X, Y, Z (Earth-Centered, Earth-Fixed Coordinate Frame)? More specifically, are the observed values for X, Y, and Z in the Output pane the values that were expected based on the oracle – be it a human calculating the values using a calculator and the correct formulas or another (known to be accurate) application with the appropriate conversion algorithms like www.gpsvisualizer.com? If the actual output matched the expected output, the test passes and **validates** a requirement. This is an obvious requirement (that the software titled Coordinate Conversion converts coordinates correctly); however, consider a design requirement that all output conforms to a fixed width. A positive test to validate that requirement would fail in the screenshot shown in Figure 3, as the X, Y, and Z outputs have different widths.


ConvertCords

Coordinate System Conversion Utility

Conversion Type: LLA to ECEF

Input
Latitude (deg): 39.7589
Longitude (deg): 84.1916
Altitude (m): 225

Reference

Output
X (m): 496907.0986003
Y (m): 4884835.9784322
Z (m): 4057586.403

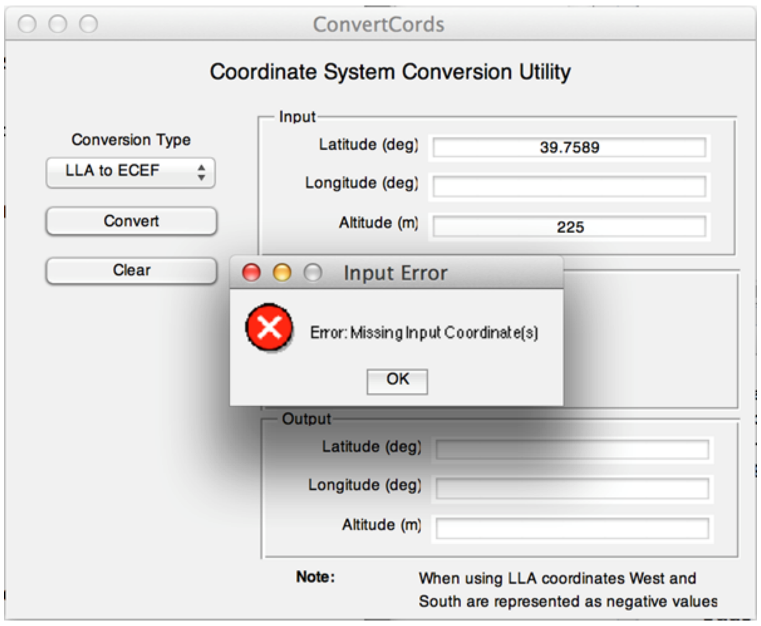
Note: When using LLA coordinates West and South are represented as negative values

Figure 3: Example of a failed positive test due to varying output lengths

Negative Testing

Successful positive tests for validation are not sufficient evidence of verification, or, that the product is built correctly. Defects or faults (bugs) can easily exist within software and go unnoticed by tests for validation. **Negative testing** is one method to attempt to observe a failure in order to uncover faults. In software testing, a negative test is defined as a test where invalid input or improper or unexpected behavior is applied to the SUT to observe if it can gracefully maintain its required functionality. Using the example above, does the Coordinate Conversion gracefully deal with the user failing to enter data in one of the input fields?

Figure 4 shows an example of missing input entered in by the user. As seen in the screenshot, the application handled the lack of input data gracefully. It neither crashed nor produced values in the output fields (which would be garbage values), and it notified the user of the mistake.



Input

Latitude (deg) 39.7589

Longitude (deg)

Altitude (m) 225

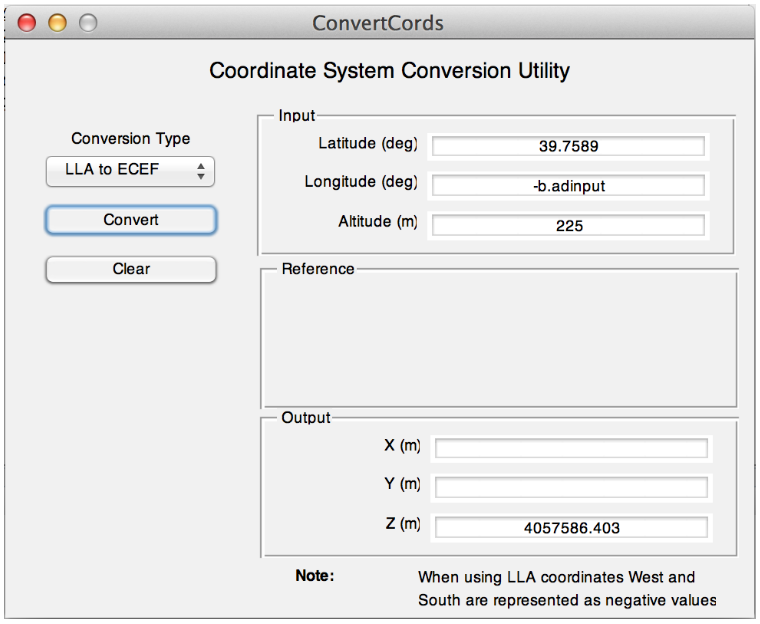
LLA to ECEF

Error: Missing Input Coordinate(s)

Note: When using LLA coordinates West and South are represented as negative values

Figure 4: Example of a negative test handled gracefully

What about a negative test for *bad* input data? Assuming the oracle expects that the application similarly provides a message box to indicate the input error, the negative test depicted in Figure 5 would fail, uncovering a fault or defect in the code. However, the application did not crash and it did not report garbage values. The value for Z is correct in this case, so whether or not the application handled the bad data gracefully *enough* is determined by the oracle for the test.



Latitude (deg) 39.7589
Longitude (deg) -b.adinput
Altitude (m) 225
X (m)
Y (m)
Z (m) 4057586.403

Note: When using LLA coordinates West and South are represented as negative values

Figure 5: Another example of a negative test, perhaps not handled gracefully *enough*

There are many different types of negative testing scenarios for dealing with input fields, such as:

- How the SUT handles empty data
- How the SUT handles improper data type input (e.g., string instead of integer)
- How the SUT handles data boundaries (integer overflow)
- How the SUT handles unreasonably bounded data (-1 entered in a field titled: Age)

Myth of Complete Testing

The above examples (Figures 3-5) illustrate the types of testing that can be performed on a rather simple application. An oft-stated intention of software testing is to "completely test" a program. For nontrivial programs, this is an impossible goal. It is important to understand that goals or claims of testing that use the word "complete" e.g., *complete* code coverage or *complete* path coverage are almost always referring to a *complete exercise* of either requirements (validation), or a *complete exercise* of all functions of code or even lines of code. This sounds comprehensive, but it does not mean every function is run through an exhaustive gauntlet of negative tests and time-varying tests. It means that each line of code is exercised at least once in some way by a test – usually a positive test. There are too many possible inputs and too many branching paths to test completely. Consider that in addition to testing all valid inputs, complete testing would require:

- All invalid inputs, which include anything that can be entered on a keyboard
- All edited inputs, which include all possible character entries, deletions, and new characters
- All variations of input timing

In addition to testing against all possible inputs, complete testing requires testing all possible states of the program, which requires all possible paths through the code. Kaner et al. (1999) describe a real-life state machine with only six possible states. Seemingly easy enough to "test completely," a latent fault existed that was only exposed if the program transitioned between State4 and State5 30 times before finally transitioning to State6.

For a more contemporary example, the video game speed-running community recently exploited a redundant controller input polling loop to create a buffer overflow, allowing them to 'beat' Super Mario Bros 3 for the Nintendo Entertainment System (NES) in two seconds. In order to exploit the loop, they had a robot cycle through 6,000 button presses on the NES's controller without any button being pressed twice in succession (Orland, 2016). The speedrunners had come across online documentation discussing the function of the redundant controller input polling function. They used those release notes to dream up the sequence of valid inputs required to create the buffer overflow and to then execute machine code to 'break' the game. By now it should be self-evident that a tester could dream up an unlimited number of *negative* tests that can be applied to even a simple software application. For those that remain skeptical of the inherent limitations of software testing, Kaner et al. (1999) dedicate an entire chapter to delivering the bad news, using real world bugs (some that were catastrophic) to illustrate the myth of complete testing.

Test Coverage

Many studies have estimated that software testing accounts for at least 50 percent of the overall budget on any software development effort (Beizer, 2003). Since complete testing in the strictest sense is impossible, the resource allocation for software testing is arbitrary as there is no finite amount of testing that can guarantee complete testing of software. In practice, the actual cost of software testing is determined by how much it costs to **reduce uncertainty of the software quality to the appropriate amount** for that application. Test coverage (also sometimes called code coverage) refers to the types of metrics used to qualify the effectiveness of a testing effort by quantifying (at various levels of depth and granularity) the amount of the SUT that has been exercised by testing. The National Institute of Standards and Technology's (NIST) Computer Security Resource Center outlines some of the better known coverage metrics (Computer Security Division).

- **Statement coverage**: This is the simplest of coverage criteria and is the percentage of statements exercised by the test set. While it may seem at first that 100 percent statement coverage should provide good confidence in the program, in practice, statement coverage is often considered nearly worthless. At best, less than nearly full statement coverage indicates that a test set is inadequate.
- **Decision or branch coverage**: The percentage of branches that have been evaluated to be both *true* and *false* by the test set.
- **Condition coverage**: The percentage of conditions within decision expressions that have been evaluated to be both *true* and *false*. Note that 100 percent condition coverage does not guarantee 100 percent decision coverage. For example, *if (A || B) {do something} else {do*

something else} is tested with [0 1], [1 0], then A and B will both have been evaluated to 0 and 1, but the *else* branch will not be taken since neither test leaves both A and B false.

- **Modified condition decision coverage (MCDC)**: This is a strong coverage criterion that is required by the U.S. Federal Aviation Administration for Level A (catastrophic failure consequence) software; i.e., software whose failure could lead to complete loss of life. It requires that every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to independently affect the decision outcome, and that each entry and exit point have been invoked at least once.

Software Testing Techniques

While it is beyond the scope of this document, the field of software testing employs many advanced techniques to improve the quality of testing. The Scientific Test and Analysis Techniques Center of Excellence (STAT COE) has published two best practices on software testing: *Combinatorial Test Designs* (Bush and Ortiz, 2014) and *Scientific Test and Analysis Techniques for Software Testing* (Ortiz, 2015). The research of Dr. Charlie Colburn at Arizona State University and Drs. Rick Kuhn and Raghu Kacker at NIST is at the forefront of improving software testing techniques. See Section *Further Reading* for additional resources. Since complete testing is impossible, pragmatic testing is about finding shortcuts to reduce the number of tests, employing a hacker's mindset when testing software, and increasing the efficiency of test generation, test execution, and test evaluation. This brings us to automation and the role it can play in improving software testing.

What Can Be Automated?

The software development life cycle (SDLC) generally speaks to testing as a discrete stage that occurs after requirements and development but before release and then feeds back into development in the form of debugging. This is a waterfall paradigm that has existed for decades and is still followed for many software development efforts as displayed in Figure 6. In this model, there are four types of testing that are delineated: implementation, unit, integration, and system testing.

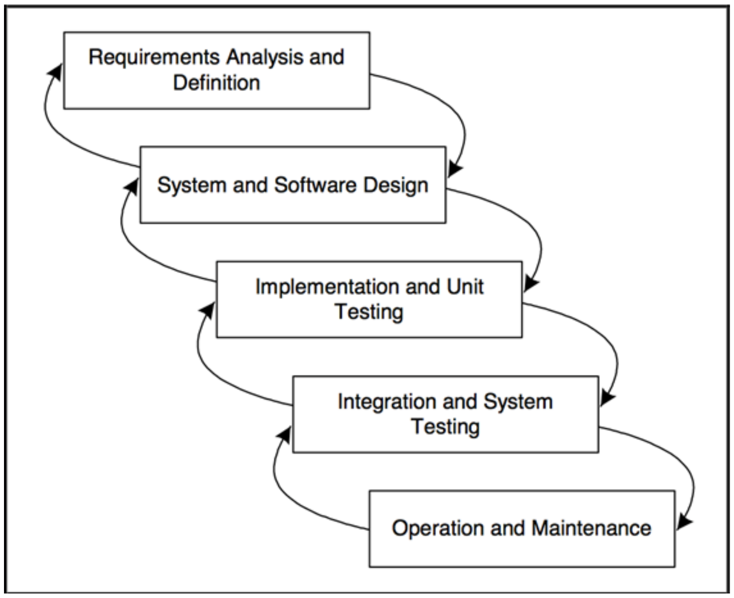

Requirements Analysis and Definition
System and Software Design
Implementation and Unit Testing
Integration and System Testing
Operation and Maintenance

Figure 6: Software development lifecycle (Tassey, 2002)

Different organizations have different categories of tests for these *testing stages* of the software development process. The general stages of testing are defined by Jones (1997):

- Subroutine or Unit Testing
- New Function Testing
- Regression Testing
- Integration Testing
- System Testing

Unit Testing

Unit testing is used to validate the smallest functional units of code – be they termed subroutines, functions, methods, or modules. Unit tests are written by software developers and they contain no working knowledge of the rest of the program. As such, they require faked environments to run and generally use stubs (sometimes called mocks or fakes) to take the placeholder of dependencies that would otherwise be needed to exercise the module of code under test. Unit tests are more useful for developers to ensure robust code vs testers; however, unit testing can be automated as regression testing to maintain a peace of mind that new functionality doesn't break established code. The automation of unit testing is one of the hallmarks of Test-Driven Development (TDD), which is discussed at length in the STAT COE Report-3-2017: *Automated Software Testing State of DoD and Industry*. Quality unit tests expose bugs earlier than any other form of testing, making it the cheapest form of debugging. However, the number of exposed bugs uncovered by the *automation* of unit testing for regression testing is much less than running manual tests at the system level, which exercises the *relationships* between units of code. At the system level, faults are more likely to be hidden since the unit tests have already been written and conceivably run at least once to ensure proper behavior of the units of code. That said, having quality unit tests makes code refactoring a much safer endeavor.

New Function Testing

New function testing is simply checking a new piece of code for possible bugs, which can be done with a quality unit test or any other test that isolates the new functionality.

Regression Testing

Regression testing is more of a process with a goal than a specific stage. Regression testing seeks to check if any recent changes to the code base have exposed bugs in any other previously stable parts of the code base. One form of regression testing is running every unit test after adding or refactoring code. **Smoke testing,** which is sometimes referred to as “Build Verification Testing,” is like safe-mode testing, where a subset of tests is run to validate the most important functions of the SUT. This subset is usually much smaller than exhaustive regression testing and can be used to decide if the current state of the SUT is stable enough to proceed with further testing. Exhaustive regression and smoke testing are outstanding candidates for automation. The actual tests that make up the regression and smoke testing are likely to be executed many times throughout the SDLC. By automating these tests, testers can proceed to test new functions or create new exploratory manual tests knowing that the code base is stable or in the case of smoke testing, that the main functionality is stable.

Integration Testing

Integration testing checks whether connecting multiple code components together exposes bugs. Compare integration testing to unit testing, which completely isolates each code component. The scale of integration testing can be two components or the entire end-to-end system. However, integration testing still implies that the environment in many ways is faked to isolate some subset of the code base. In that sense, integration testing is still a form of white-box testing where the tester has the access to the internal systems. As a general rule of thumb, increased isolation of the components being integrated plus a controlled environment (using fakes and limiting shared resources) ensures that the tests are more reliable. Miller (2015) provides some key best practices for automating integration tests.

System Testing

System testing exercises the entire SUT through its publicly exposed interfaces. In other words, system testing uses tests that are characteristic of possible user/customer behavior. Any testing described as “black box testing” is usually system testing. There are many ways to automate system testing. It is important to note that in many cases, the test execution involves the user interfaces and thus is limited to the real-time operation of the SUT. Because of this, any efficiency gains from automation do not necessarily manifest in time savings of test *execution* over a manual approach. However, through automation, data can be input more quickly into text boxes or button clicks can happen more rapidly. In many case studies, other factors and overhead actually made the automated testing slightly slower than manual test execution. However, the benefits of automation are that it releases the tester from the mundane task of entering data over and over. Mundane testing tasks can lead to tester fatigue, so automation mitigates those types of human errors. Additionally, automating the system testing can allow tightly integrated logging and recording of not only test results but the state of various system objects. Even simple things like screenshots or short video capture can be linked to the test cases and gives the tester much more insight about an error. Additionally, with an excellent logging system in place, tests can be executed overnight without the presence of a tester.

Other Candidates for Automation

AST is not limited to executing tests. One of the most beneficial uses of automation throughout the software development process is in analysis. Take, for example, a program that receives message traffic over a data link and after some conditioning logs it to a data file. While a useful test case might expect traffic or a log file upon the completion of some function, the data in the file probably cannot be predicted accurately in a system test. In a manual case, a tester might perform a cyclic redundancy check (CRC) or peruse the file for anomalies (assuming it is readable by humans). If, however, during the process of testing, multiple files of this nature are generated, it can quickly become a burden for manual testing and is a prime candidate for an automated analysis tool. Automation can also be leveraged to trace test cases to requirements.

Prioritizing what to automate is an important step for any newly fledged AST effort. Figure 7 provides a sample checklist similar to an AST effort that was completed for an Air Traffic Controller System of Systems (Graham and Fewster, 2012).

Table 1: Checklist for prioritizing test cases to automate

	Assessment Criteria	Notes
1	How many times will the test be repeated?	
2	Does execution require more than 1 person?	Automation can accomplish tasks that require more than 1 person
3	How much time is saved by automating this test?	Don't forget time for test generation and analysis
4	Does this test exercise the critical path?	Will it become more important over time?
5	Are the requirements being verified by this test low risk or unlikely to be impacted by change?	Is the benefit worth the maintenance cost?
6	Is the test likely to be reusable by other projects?	Anticipated shared costs and benefits down the road is attractive
7	Is the test susceptible to human error?	Overly complex
8	Is there significant idle time between test steps?	
9	Is the testing mundane or highly repetitive?	Eliminate tester fatigue
10	Does the test require domain expertise?	Automation can minimize reliance on domain experts

Before you Begin

In a survey of over 700 test professionals, 70 percent of respondents said they believe that automation for software testing is a high payoff endeavor; however, they were not sure why that was or how automation fit with their project (Dustin et al. [2009]). This sort of uninformed optimism requires a deeper understanding in order to proceed with a business case for implementing AST.

Understand the Benefits of AST

The goal of AST is to improve efficiency and effectiveness throughout the software testing lifecycle. In many cases, AST can achieve this goal by enhancing the efforts of manual testers. As an example, AST can replace the tedious and repeatable process of entering various data into text fields, freeing up a tester to focus on designing new and better test cases or working to implement better software testing techniques. Additionally, the repeatability of AST mitigates various human errors on the part of testers. Once implemented, AST can reduce the uncertainty of testing efforts.

Understand the Limitations of AST

Good automation of software testing is no substitute for good software testing. Automating poorly throughout test cases just means faster and more repeatable tests of marginal utility. Having the manual testing process in order is a prerequisite to any successful automation effort. Also, some of the inherent challenges in manual software testing cannot be overcome by automation: complete testing is still impossible. Within the larger understanding of software economics, automation is not always considered a holy grail. According to Brown et al. (2013), in most cases, productivity gains from automation are between 5 and 25 percent of the overall software development effort. See Figure 8 for Brown et al.'s Chart on the Economic Impacts of Software Development Decisions.

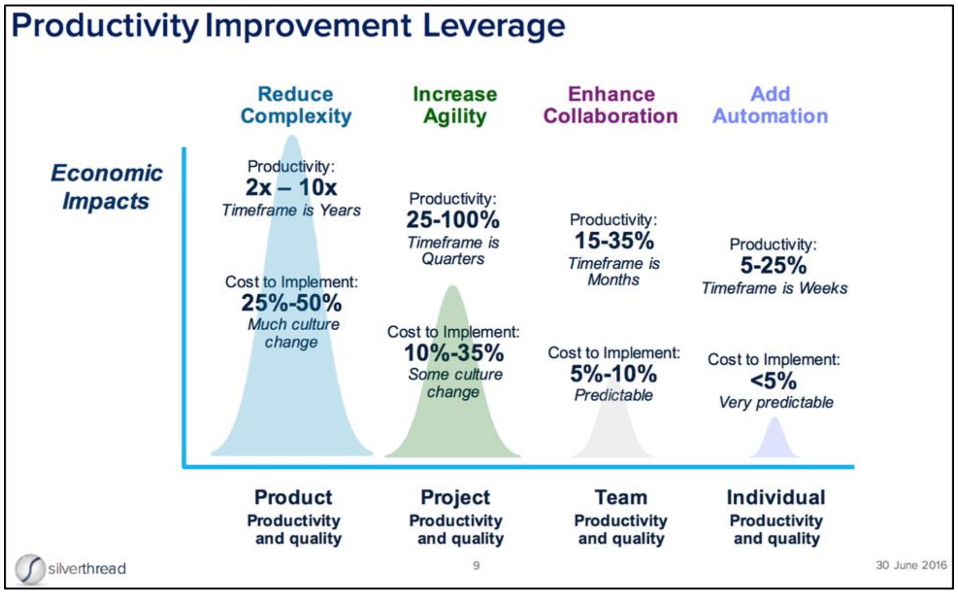


Figure 8: The economic impacts of software development decisions (Brown et al., 2013)

Understand the Challenges *Created by* AST

AST poses some unique testing challenges that could otherwise be mitigated by an experienced manual tester. As an example, consider a manual tester presiding over a battery of test cases. The tester will run a test, monitor the SUT, and in the event that the test fails, will write up a bug report. Ascertaining whether or not the test has revealed a bug is known as the "**oracle problem**." Kaner (2012) provides a focus group-like description of the "oracle problem" and how it uniquely affects automated testing. To demonstrate the difference between a human tester executing a manual test and a test execution tool running automated tests, consider a test to verify that a calculator program adds $2 + 2$ correctly. A human might recognize that a certain test uncovered a bug because it took the calculator a few *minutes* to produce the value 4 – an unacceptable amount of computing time. An automation framework without some sort of expectation of timeliness spelled out in the oracle would pass this test, failing to report what a human would recognize as an obvious symptom of a bug. Alternatively, imagine an automated test that fails because some other program, system, or hardware created a resource conflict (perhaps opening a file). AST would report a failed test but had a human been inspecting the SUT during the execution of the test, he or she could have realized that the SUT behaved appropriately even if the outputs did not match the oracle. This lack of situational awareness over the testing can create a false sense of security when tests pass, and can engender inappropriate lack of interest over presumed false alarms when tests fail.

Additionally, for AST to be useful at helping testers trace failed tests to the underlying bugs, the "testability" of code needs to exist within the SUT or a monitor recording the application state and diagnostic information needs to be part of the testware. Improving testability could involve refactoring the code so that every component logs its state, the state of the data it is reading or writing, and provides timestamps and other relevant information to aid in debugging. Testability can also refer to different components being exposed to test cases. If a requirement for testing is some code coverage metric, then the software developers will have to create hooks to expose otherwise untestable parts of the source code.

Set Clear, Deliberate, and Attainable Goals

Having smart and clearly defined goals for automation (not *testing* – this is an important distinction) is the first in a line of necessary (but not sufficient) practices to achieve successful AST. This seems obvious, but a lack of clearly defined goals for automation is dooming the initiatives from the beginning. For example, a goal of software testing is to expose faults, or "find as many bugs as possible." **This may be a poor goal for AST**. One of the most common uses for automation is the overnight execution of regression tests to ensure that changes to the code base did not break something that was previously working (or regress). In most cases, these types of tests are ill-suited to find bugs. One reason is that unit tests are generally written as *positive* tests – meant to exercise the function of the smallest units of code, not run it through a battery of negative tests. Another is that unit tests are written, not auto-generated. Therefore, a developer will certainly test his or her code with the unit test at least once before moving on. The automated regression testing of unit tests, then, is less likely to find bugs than manual exploratory testing performed by experienced testers. This does not necessarily mean that the automation effort was a failure, but it could be interpreted as such by management if the goals of automation are not aligned with the strengths of automation and the needs of the project. In one of the case studies by Graham and Fewster (2012), the automation team reported that only 9.3 percent of bugs were found using automated tests. However, the effort to automate was considered a wild success because the goal of the automation was to free up time for testers to perform manual exploratory tests. After three release cycles with the automation in place, the team reported that 58.2 percent of bugs were found using manual exploratory (mostly negative) tests.

Commit from the Top Down

It is tempting to think of automation as simply "faster testing" or "saving time" – platitudes that are difficult to evaluate. This is a major pitfall. Effective AST requires investments up front: new resources, culture changes, enabling skillsets, and robust architectures. In short, there is not much of a business case for automation with limited up-front investment or a lack of patience. Treating automation as a side project is almost guaranteed to lead to disappointing results. Because of this pitfall, the goals of any automation effort must be matched by a willingness of leadership to invest and lead from the top down. Very few efforts to automate software testing succeed with a bottom-up approach. In many case studies, the consequences of a lack of commitment from leadership included misplaced personnel and team-wide frustration as the testers were asked to become coders and the coders to become testers. In this scenario, the quality of both functions suffered.

Research the Return on Investment

It is a good practice to estimate return on investment (ROI) at the outset of an automation effort. One pitfall is to assume that the investment portion is limited to the cost of a commercial AST tool. There are other significant investments, including:

- Research and development of an overarching testware architecture, of which tools are usually sub-components
- Designing automation solutions that address the project needs while being robust enough to be useful down the road
- Maintenance costs of the testware. Remember, software is used to automate software testing. Easier maintenance down the road requires a better architecture (more investment) up front. The principles of software economics apply to Testware just the same.
- New project management processes. This is part of the culture change that needs to happen for successful AST.
- Pilot programs. Automation tools for software testing have existed for many decades. By now, many have realized that an adequately funded pilot program sandboxed from other software development efforts is the most fruitful way to inject useful automation into software testing.
- Refactoring legacy code. In order for legacy code to become testable using automation, it very often needs to be refactored to be more modular or at least have injection points (hooks) for the automated scripts to insert test vectors for unit and integration testing.
- Personnel overhaul. New hires are often required for a successful implementation of AST and existing software developers and testers may require training and certification.

The European Space Agency (ESA) underwent an overhaul of software testing to include tightly integrated automation to test their Multi-Mission User Services. Upon completion, they reported an up-front cost of 812 hours to implement test automation using a model-based approach. Lindholm (2017) provides a good overview of the process that is referred to as **Model-Based Testing**. Manual testing to create and run the same tests required just 315 hours. The difference is almost entirely based on the added investment needed to automate. However, due to serious time savings in the execution of tests and the maintenance savings (which they found to be significant) enabled by model-based test creation, the break-even point between the manual and automated approaches occurred after the fourth test cycle. A test cycle in this case was required to achieve 100 percent requirements coverage (positive testing). One thing to note is that the ESA used an in-house test execution tool called "Test Commander" in their testware that they had previously developed to automate efforts on another software project. They reported that this tool required about one person-year of development effort. If that extra 2,000 hours is included in the upfront investment of the AST effort, it slides the ROI break-even point to the 15th test cycle (Graham and Fewster, 2012).

Establish and Maintain Communication

In the example of the ESA, the test automation experts had experience automating other software projects and had experience with most of the tools (commercial and in-house) that they used to design their testware. Embarking on AST with little to no experience necessitates not only appropriate goals and the backing of leadership, but also a constant line of communication between developers, automators, and all levels of management stakeholders. Imagine taking on the ESA initiative and eventually arriving at the unforeseen conclusion that you need to develop an in-house test execution tool because none of the commercially available tools suit your project. Constant communication and expectation management is critical to avoid disappointment and sticker shock.

Start Small

In case studies and testimonials, many efforts to do AST succeeded because they were preceded by a pilot program. In this case, a pilot program is great way to reduce uncertainty surrounding the implementation of AST. The extent of the technical challenges is not always understood beforehand and AST implementation can often conflict with the deadlines and milestones of the software development process. In some implementations, the delayed payoff from AST is neither well-understood nor accepted by the team and/or its leadership. AST is also a group effort, and a pilot program is an opportunity to monitor group dynamics and evaluate the resilience or resistance to some of the culture changes required for effective AST.

Understand Competing Incentives

In many cases, developers need to architect or refactor their software to make it more testable. This is more work without obvious benefits to the developer, whose goals are usually adding features to the code base. Similarly, many testers view automation as a threat to their livelihood and are not fond of having to write any code or struggle with testware. Finally, a new role is required for implementing AST:

the role of the test automator. In many cases, a group will not have any experience with such a role and there will be a lot of uncertainty about how that role fits into the team. Consultants are often brought in to assist with this task, but quite often the long-term solution is to employ a test automator that maintains the testware. Because of all these things, getting the recipe for AST right the first time for a specific team or application is daunting. The best play is to sandbox off a small team so they can gain experience and confidence in automating tests and use the lessons learned on low hanging fruit to devise a reusable testware.

Prevent Automation Burdens

It is very important to use automation of software testing as an enabling technology for developers and testers and not a burden. To be effective, automated tests must be reliable and repeatable. If they are "overly brittle" (too many false positives) the tester is going to be overly burdened trying to discern if a test failure is "real" or not. Over time, this will induce a lack of faith in the automated tests. Additionally, if there is no traceability between tests and requirements or code, even if there is confidence that a failure was real, it can be very difficult to know *why* a test failed. Certain types of tests are more reliable than others. It is often better to have a reduced set of automated tests with less code coverage than an expansive test suite that is not reliable.

Roles and Skillsets

Google as an Exemplar

Google names their software tests by scope and not function and are called: "Small," "Medium," and "Large" tests. The scope in most cases dictates the functionality. They define them as follows**:**

Small tests cover a single unit of code in a completely faked environment. **Medium tests** cover multiple and interacting units of code in a fake or real environment. **Large tests** cover any number of units of code in the actual production environment with real and not fake resources. This may seem a bit ambiguous but it aligns very nicely with the roles that Google has identified as necessary for automated software testing. Those roles are: Software Engineer (SWE), Software Engineer in Test (SET), and Test Engineer (TE). The roles are defined as follows:

The **SWEs** are the traditional developers who spend the vast majority of their time writing and reviewing code. They write a good amount of test code, mostly small tests, but participate in medium and large tests as well.

The **SETs** are software developers as well, but their focus is on testability of code and maintaining the test infrastructure. They write the unit testing frameworks and enable the automation of software testing. They refactor code to make it more testable. Their concerns are quality and test coverage over adding new features of increasing performance and supporting the SWEs.

The **TEs** are more akin to traditional manual testers in that they test on behalf of the user. They organize the testing work of the SWEs and SETs, interpret test results, and drive test execution, especially of large tests.

Being a leading software company, Google's journey towards AST began with a disruptive decision: testers needed to be proficient programmers, and programmers needed to become quality testers. This idea was met with derision and frustration, so a small group at Google proceeded to hire candidates as part of a pilot program. Over time, the program succeeded and one of its main initiatives was to create a company-wide training certification program called "Test Certified" where every team had the opportunity to compete for prizes and bragging rights. They were all being trained to become part of a company-wide directive towards shared responsible for quality, as opposed to using testing at the end of a development cycle to try to "test in quality." While few Department of Defense (DoD) organizations have the need or the ambition to completely overhaul their way of doing business the way Google did, there is a universal truth for all efforts to implement AST: you need distinct skillsets to build the architecture/framework/testware that may require targeted hiring Whittaker et al. (2012).

Recommended Roles for AST

There are four distinct roles that are required to implement AST, although one person could manage more than one role. They are: Test Automation Architect, Test Automator, Test Executor, and Test Creator.

Test Automation Architect

The **Test Automation Architect** is most likely the role that does not currently exist if a group is embarking on an effort to implement AST. The Test Automation Architect is the person who designs the overall structure for the automation or selects and adapts the framework used to achieve a good testware architecture.

Test Automator

The **Test Automator** (which could be the same person as the test automation architect) is responsible for designing, writing, and maintaining the automation software, the scripts, the data, the oracles, and any additional tools or utilities (Graham and Fewster, Chapter A.5, 2012). Both the Test Automation Architect and the Test Automator need to have excellent programming skills and the Test Automation Architect should be familiar with Systems Engineering Principles.

Test Executor

The **Test Executor** is the person that uses the testware to create, initialize, and record the results of automated test scenarios.

Test Creator

The **Test Creator** is the closest to the traditional tester role. This is the person that understands the SUT more than any of the other roles and has the testing skillsets required to implement the best (manual) testing techniques. If the testware makes it easy to write automated tests then the Test Creator will more than likely fulfill the role of Test Executor as well and just be called the "**Tester**." This is the case with most DoD organizations.

A number of skillsets are useful or required to successfully implement AST. They are: Program Management, Systems Engineering, Software Development, Configuration Management, and Design of Experiments. We describe each of these skills in more detail below.

Program Management

- Gathering requirements for AST initiatives
- Establishing processes, metrics, and monitoring progress and outcomes
- Prioritizing the scope of AST and where it will and will not be injected as part of the SDLC
- Establishing the success criterion for the AST effort
- Evaluating manual testing efforts to estimate ROI
- Hiring new employees to fill AST roles and/or bringing in SMEs as consultants
- Procuring commercial AST tools
- Resource management
- Nurturing good communication between the AST team, the software developers, and upper management
- Constantly monitoring progress and documenting all lessons learned (most initial AST efforts look a lot different than their successors)

Systems Engineering

- Subject matter expertise on the SUT
- Develop the test plan and expected results (oracles)
- Understand the interfaces and their specifications
- Prioritize test cases to be automated

Software Development

- Identifying operating system (OS) and programming languages used by the SUT
- Identifying any tools used in manual testing and their ability to be repurposed for AST
- Designing and developing the software or software modifications to implement the testware
- Designing and developing test scripts

Configuration Management

- Developing a configuration management plan for AST
- Locking down baselines for the SUT, testware, and environment with configuration control
- Managing source code version control, product release schedules, bug reports within an AST framework

Design of Experiments

- Understanding various testing technique (boundary value, equivalence portioning, risk-based testing, orthogonal classification, combinatorial testing)
- Evaluating efficacy of AST implementation and making recommendations
- Helping prioritize test cases to be automated based on the AST effort requirements

Certification, Standards, and Training

ISTQB

The International Software Testing Qualifications Board (ISTQB) offers certification as a Test Automation Engineer. See: http://www.istqb.org/certification-path-root/test-automation-engineer.html. The syllabus can be found here: http://www.istqb.org/downloads/send/48-advanced-level-test-automation-engineer-documents/201-advanced-test-automation-engineer-syllabus-ga-2016.html

ISO 29119

The International Organization for Standardization (ISO), in conjunction with the International Electro-technical Commission (IEC) and Institute of Electrical and Electronics Engineers (IEEE) released ISO/IEC/IEEE 29119—ISO 29119 for short—a set of standards governing testing processes, documentation, techniques, and keyword-driven testing designed to create uniform testing practices within any SDLC or organization. Parts 1, 2, and 3 of the standard were published in September 2013, with Part 4 published in 2015, and Part 5 published in November of 2016 (International Organization for Standardization).

BBST

The Association for Software Testing offers training courses for Black Box Software Testing (BBST). https://www.associationforsoftwaretesting.org/training-2/courses/

Tips for Good Testware

Testware includes tools, resources, scripts, generators, analyzers, and any other components of an AST architecture. The testware is the glue that holds everything together. One of the main functions of testware is to seamlessly link test cases to the test execution tools that apply them to the SUT.

In most cases, the job of testers is to develop good tests. It is easy to lose sight of that when embarking on an AST effort. In many cases, a team will acquire a test execution tool and expect the testers to get to work automating their tests cases. But if the testers are not developers they will be unfamiliar with the scripting language of the tool. If they decide to struggle through it, they will most likely produce poor automation code that will become very difficult to reuse or maintain. This is where testware, the glue, comes in.

Abstraction

Any useful testware will include a layer of abstraction between the automation scripts that execute the test cases and the cases themselves. This allows testers to write tests in a domain specific language with which they are comfortable. One of the jobs of the testware is to translate those into test cases for the test execution tool for automation. In this way, good testware acts as a compiler from one language (the testers' language) to another (the test execution tool's language).

Additionally, good testware is built to be flexible to include different tools. Very often, AST efforts involve switching commercial tools or the team decides to build their own. If a testware is too tightly bound to a specific tool it may have a shorter lifespan or be less reusable. This concept of abstraction layers is shown below (Figure 9).

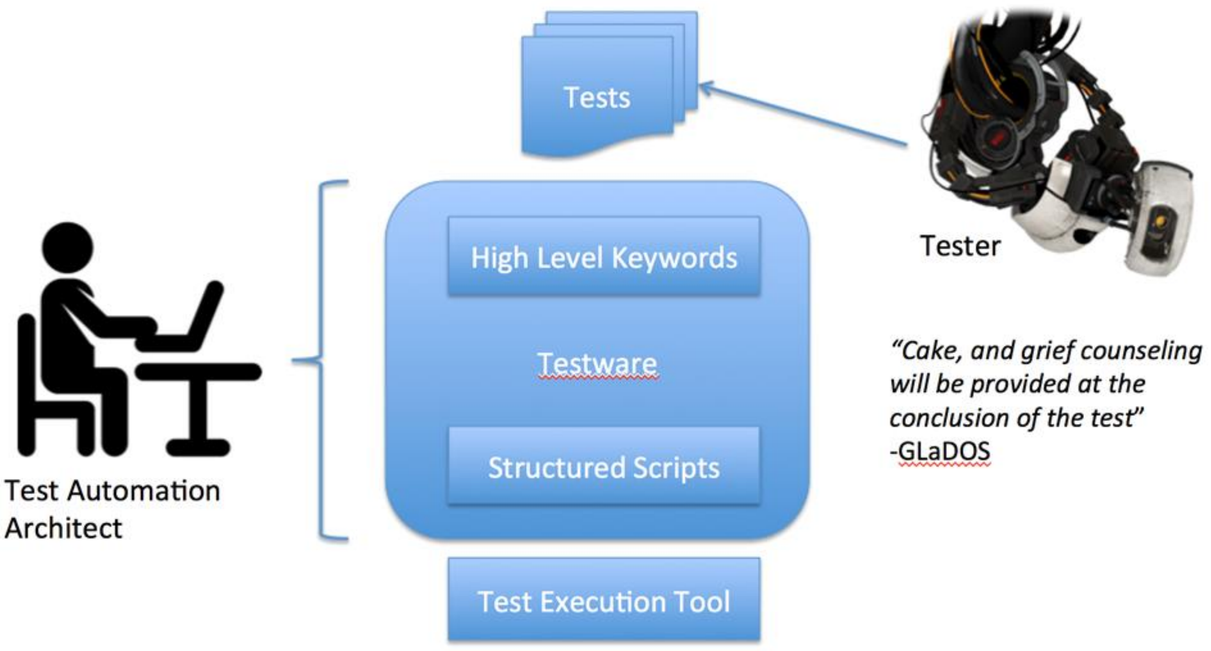

Tests
Tester
High Level Keywords
Testware
Structured Scripts
Test Automation Architect
Test Execution Tool
"Cake, and grief counseling will be provided at the conclusion of the test"
-GLaDOS

Figure 9: A diagram for robust testware through abstraction and delineation

Signs of Well-Designed Testware

- Testers are empowered
  - Easy for non-programmers to write tests
  - Tests are useful and not overly brittle
- Reused by other teams/programs
  - Test cases are shared
  - Testware is appropriate for various SUTs
- Maintainability

- Changes to the SUT, tool, or environment don't break everything
- Modular and expandable

Investments and Goals

It is important to decide early how "all-in" to be with respect to the testware. Certainly, there are commercial products that are useful for automating specific stages of the SDLC. Some are even open source. Creating a robust and tester-friendly testware is a very challenging and ambitious endeavor. Figure 10 illustrates the difficulty of trying to overhaul software testing.

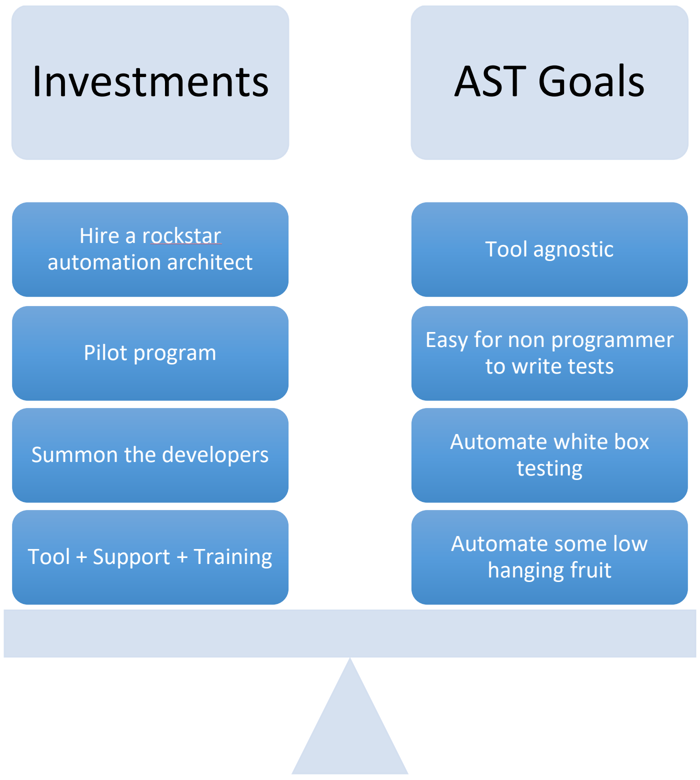

Investments
AST Goals
Hire a rockstar automation architect
Pilot program
Summon the developers
Tool + Support + Training
Tool agnostic
Easy for non programmer to write tests
Automate white box testing
Automate some low hanging fruit

Figure 10: Balancing AST goals with the appropriate investments

Testware done right will take much longer to provide a positive ROI. There needs to be a strong business case to justify the amount of resources and culture change that is required.

Tools for AST

Testing tools tend to be specialized in three ways: the application domain, the kind of interface supported, and the platform supported. Within these types, tools are typically further specialized by the kind of quality attribute they can evaluate, of which functionality, performance, and security are the most common. Listed below are many of the commercial and open source tools used to automate software testing.

Functional Testing for Enterprise Applications

- Hewlett-Packard Unified Functional Testing (Formerly Quick Test Pro): HP UFT started a 3-year roadmap in 2015 to appeal to Agile developers focused on continuous integration. http://www8.hp.com/us/en/software-solutions/unified-functional-automated-testing/
- IBM Rational Test Workbench: IBM Rational Test Workbench now appeals to development groups thanks to an integration strategy that enables deployment and control of test environments across large numbers of virtualized assets. It has focused on dev-ops and continuous integration, but not at the expense of end-to-end testing. http://www-03.ibm.com/software/products/en/rtw
- Microsoft Test Manager: Microsoft Test Manager's focus on DevOps allows it to execute diverse sets of tests: unit, Selenium and coded UI, functional non-UI, and load and performance as part of its build/release automation workflow. https://msdn.microsoft.com/en-us/library/ms182409(v=vs.110).aspx
- MicroFocus Silk Test https://www.microfocus.com/products/silk-portfolio/silk-test#
- Oracle Application Testing Suite http://www.oracle.com/technetwork/oem/app-test/etest-101273.html
- SmartBear TestComplete: SmartBear TestComplete is a lesser known AST product that is highly applicable to DoD. It is fairly easy to learn, graphical user interface (GUI) driven, customizable, works well with other tools, supports scripts written in other common coding languages (e.g., Python), though it is restrictive on the objects sharing repository for reuse. https://smartbear.com/product/testcomplete/overview/
- IDT ATRT: Innovative Defense Technologies Automated Test and ReTest is the most common tool in DoD AST. This is not surprising given the mandate to many Navy test organizations to use the product. It is currently available to all DoD organizations and on-site contractor support is available at a cost. ATRT uses a technology called bitmap capture-replay to automate GUI testing, which is a deprecated technology that succumbs to issues with brittle tests that can result in a large maintenance tail. IDT has succeeded in breaking down automation barriers for as many as 75 DoD programs. Many program offices echo IDT's claims of ATRT reducing testing cost and time; however, some have noted unforeseen difficulties and costs associated with maintaining the test infrastructure over time. http://idtus.com/products/atrt-test-manager/

Testing Graphical User Interfaces (GUIs) Natively

- Eggplant http://www.testplant.com/eggplant/testing-tools/
- Squish https://www.froglogic.com/squish/gui-testing/index.php
- Ranorex http://www.ranorex.com
- Test Studio http://www.telerik.com/teststudio

Browser Testing

- Selenium: Selenium is a widely used open-source web browser automator for testing web applications. http://docs.seleniumhq.org

- Watir https://watir.com

Web Testing

- Tellurium http://www.te52.com
- Sahi http://sahipro.com

API Testing

- SoapUI https://www.soapui.org

Unit Testing

- NUnit http://www.nunit.org
- xUnit https://github.com/xunit/xunit
- pyUnit https://docs.python.org/2/library/unittest.html
- JUnit http://junit.org/junit4/
- Google Test https://github.com/google/googletest
- PHPUnit https://phpunit.de
- TestNG http://testng.org/doc/
- Test::Unit http://ruby-doc.org/stdlib-2.0.0/libdoc/test/unit/rdoc/Test/Unit.html
- T3 https://git.science.uu.nl/prase101/t3/wikis/home

Continuous Testing Tools

- ContinuousTests http://www.continuoustests.com/
- Wallaby.js https://wallabyjs.com/
- Autotest https://github.com/grosser/autotest
- Infinitetest https://infinitest.github.io/

Other

- Automated Combinatorial Testing for Software: NIST ACTS http://csrc.nist.gov/groups/SNS/acts/index.html
- Model Based Testing: TEMA http://tema.cs.tut.fi/index.html
- Embedded: VectorCAST C++ https://www.vectorcast.com/sites/default/files/pdf/resources/vectorcast_c.pdf

Conclusion

Among many of the case studies that served as source material for this document was a common theme of uninformed optimism of the prospects of automating software testing followed by enduring frustration and in some cases, abandonment. Probably the most cited complaint among those embarking on AST was that after some amount of time, maintainability of test scripts and the testware architecture became too burdensome. When it comes to any type of software testing, it is easier to identify common pitfalls than best practices. In many cases, this is because testing is inherently about trade-offs and business decisions, not rote processes. Recall the myth of complete testing and how the amount of testing can be an arbitrary decision. AST can reduce uncertainty around the SUT during the SDLC through automated unit and regression testing. It can be used in system testing (including Black Box System Testing) to alleviate errors from human tester fatigue or to run a battery of input and performance tests on the SUT, often overnight. However, from the source material used for this document, there was a significant number of testers that found manual, exploratory testing was superior at finding bugs over automated test cases. Additionally, whether or not to automate depends in many ways on the SDLC, software architecture, deployment schedule, and tolerance for bugs.

An “Agilista” working on a headless service for ETSY[1] – where they push 50+ deployments each day – is completely immersed in automation. As a philosophy, Google releases code that has enough feature goodness to be considered useful while fully intending to continuously make improvements, fix bugs, etc. These companies developed software to scale for millions of users, and the consequences of a fault occurring are limited by the ability to immediately roll-back to a working version and the nature of the application. An Air Traffic Control software application is developed with different intentions. As such, the amount and type of software testing and the affinity of that testing to automation will differ as well. Before embarking on any venture to inject automation into any part of the SDLC, including testing, there needs to be a firm grasp on the business case, including the goals and scope of automation and a solid alignment between goals and investments.

Further Reading

Complementary STAT COE Reports

- Automated Software Test Implementation Guide
- Automated Software Testing State of DoD and Industry

Associated STAT COE Reports

- STAT for Software Testing Best Practice
- Combinatorial Test Designs

Books

- Introduction to Software Testing http://cs.gmu.edu/~offutt/softwaretest/
- Testing Computer Software https://www.amazon.com/dp/0471358460/
- Software Test Automation https://www.amazon.com/dp/0201331403/
- Experiences of Software Test Automation https://www.amazon.com/gp/product/0321754069

[1] https://www.etsy.com/

- Just Enough Software Test Automation https://www.amazon.com/dp/0130084689/

Articles

- Introduction to Test Driven Development http://www.agiledata.org/essays/tdd.html
- When to Automate Testing http://davidweiss.blogspot.de/2006/08/when-to-automate-testing.html
- The Software Testing Schism http://sdtimes.com/software-testing-schism/
- Make Testing Great Again http://xndev.com/2016/05/make-testing-great-again/
- The Oracle Problem and the Teaching of Software Testing http://kaner.com/?p=190
- Practical Combinatorial Testing http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf
- Testing Overview and Black-Box Testing Techniques http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf
- The Super Mario Bros Glitch Explained https://www.reddit.com/r/speedrun/comments/4s29rt/sgdq_2016_tas_block/d55ysqx/?context=1
- Automated Testing Best Practices and Tips https://smartbear.com/learn/automated-testing/best-practices-for-automation/

Other

- Matt Heusser – “How to talk about Coverage” https://www.youtube.com/watch?v=cVAlGRgTJ58#t=6m
- Matt Heusser – “How to Speak to Agilistas if you absolutely must” https://www.youtube.com/watch?v=DmVCdUz_M_k
- NIST http://csrc.nist.gov/groups/SNS/acts/index.html#briefings
- Dr. Charles Colburn http://www.public.asu.edu/~ccolbou/
- Cem Kaner on High Volume Test Automation https://www.youtube.com/watch?v=iP4tZ_i6eDY
- Standard Glossary of Terms Used in Software Testing http://www.istqb.org/downloads/send/20-istqb-glossary/189-extract-of-terms-used-in-the-advanced-test-automation-engineer.html
- Test Automation Guild https://automationguild.com/

References

"Coverage Measurement - Automated Combinatorial Testing for Software | CSRC." Computer Security Resource Center , NIST, http://csrc.nist.gov/Projects/Automated-Combinatorial-Testing-for-Software/Coverage-Measurement.

"Software and Systems Engineering -- Software Testing ." International Organization for Standardization, ISO/IEC/IEEE 29119-5:2016, Nov. 2016, www.iso.org/standard/62821.html

"Standard Glossary of Terms Used in Software Engineering." https://www.astqb.org/Documents/Standard-Glossary-of-Terms-Used-in-Software-Engineering-1.0-IQBBA.pdf, 11 Oct. 2011.

Beizer, Boris. *Software testing techniques*. Dreamtech Press, 2003.

Brown, Alan W., Scott Ambler, and Walker Royce. "Agility at scale: economic governance, measured improvement, and disciplined delivery." *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013.

Bush, Brett and Francisco Ortiz. "Combinatorial Test Designs." Scientific Test and Analysis Techniques Center of Excellence (STAT COE), 25 Mar. 2014

Copeland, B. Jack. "The Modern History of Computing." Edited by Edward N Zalta, Stanford Encyclopedia of Philosophy, Stanford University, 9 June 2006, plato.stanford.edu/archives/fall2008/entries/computing-history/.

Dustin, Elfriede, Thom Garrett, and Bernie Gauf. *Implementing automated software testing: How to save time and lower costs while raising quality*. Pearson Education, 2009.

Gannon, Paul. *Colossus: Bletchley Park's Greatest Secret*. London: Atlantic, 2006.

Graham, Dorothy, and Mark Fewster. *Experiences of test automation: case studies of software test automation*. Addison-Wesley Professional, 2012.

Jones, C.C. *Software quality: Analysis and guidelines for success*. Thomson Learning, 1997.

Kaner, Cem, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software Second Edition*. Dreamtech Press, 2000.

Kaner, Cem. "A Course in Black Box Software Testing ." Center for Software Testing Education & Research, 14 Mar. 2005, www.testingeducation.org/k04/OracleExamples.htm.

Kaner, Cem. "The Oracle Problem and the Teaching of Software Testing." Cem Kaner JD PhD, 11 Sept. 2012, kaner.com/?p=190

Lindholm, Joonas. “Model Based Testing.” University of Helsinki, 1 Nov. 2006. https://www.cs.helsinki.fi/u/paakki/lindholm.pdf

Miller, Jeremy D. “Succeeding with Automated Integration Tests.” The Shade Tree Developer, 25 June 2015, http://jeremydmiller.com/2015/06/25/succeeding_with_integration_testing.

Murray, Brian T., and John P. Hayes. "Testing ICs: Getting to the core of the problem." *IEEE Computer* 29.11 (1996): 32-38.

Orland, Kyle. “How to Beat Super Mario Bros. 3 in Less than a Second.” *Ars Technica*, 12 July 2016, http://arstechnica.com/gaming/2016/07/how-to-beat-super-mario-bros-3-in-less-than-a-second/.

Ortiz, Francisco. “Scientific Test and Analysis Techniques for Software Testing.” Scientific Test and Analysis Techniques Center of Excellence (STAT COE), 17 Apr. 2015.

Royce, Walker, and Dan Sturtevant. “-silverthread- Take Back Control: Improve Design Quality and Software Economics to Drive Results.” Silverthread Demonstration. 30 June 2016, Hanscom AFB, MA. Presentation.

Tassey, Gregory. "The economic impacts of inadequate infrastructure for software testing." *National Institute of Standards and Technology,* (2002).

Whittaker, James A., Jason Arbon, and Jeff Carollo. How Google tests software. Addison-Wesley, 2012.

Williams, Laurie. Testing Overview and Black-Box Testing Techniques. http://agile.csc.ncsu.edu/SEMaterials/.

Appendix A: Acronym List

Acronym	Description
ACTS	Automated Combinatorial Testing for Software
AST	Automated Software Testing
ATRT	Automated Test and ReTest
BBST	Black Box Software Testing
COE	Center of Excellence
CRC	Cyclic Redundancy Check
DoD	Department of Defense
ENIAC	Electronic Numerical Integrator And Computer
ESA	European Space Agency
GUI	Graphical User Interface
HP	Hewlett Packard
IDT	Innovative Defense Technologies
IEC	International Electro-technical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
ISTQB	International Software Testing Qualifications Board
MCDC	Modified condition Decision Coverage
NES	Nintendo Entertainment System
NIST	National Institute of Standards and Technology
O/S	Operating System
ROI	Return on Investment
SDLC	Software Development Life Cycle
SET	Software Engineer in Test
STAT	Scientific Test and Analysis Techniques
SUT	System(s) Under Test
SWE	Software Engineer
TDD	Test-Driven Development
TE	Test Engineer
UFT	Unified Functional Testing