SYM-AM-15-097

PROCEEDINGS
OF THE
TWELFTH ANNUAL ACQUISITION RESEARCH SYMPOSIUM

THURSDAY SESSIONS
VOLUME II

Use of Automated Testing to Facilitate Affordable Design of Military Systems

Valdis Berzins, NPS
Paul Van Benthem, SPAWAR
Christopher Johnson, SPAWAR
Brian Womble, DASN (RDT&E)

Published April 30, 2015

The research presented in this report was supported by the Acquisition Research Program of the Graduate School of Business & Public Policy at the Naval Postgraduate School.

To request defense acquisition research, to become a research sponsor, or to print additional copies of reports, please contact any of the staff listed on the Acquisition Research Program website (www.acquisitionresearch.net).

Use of Automated Testing to Facilitate Affordable Design of Military Systems[1]

Valdis Berzins—is a professor of computer science at the Naval Postgraduate School. His research interests include software engineering, software architecture, reliability, computer-aided design, and software evolution. His work includes software testing, reuse, automatic software generation, architecture, requirements, prototyping, re-engineering, specification languages, and engineering databases. Dr. Berzins received BS, MS, EE, and PhD degrees from MIT and has been on the faculty at the University of Texas and the University of Minnesota. He has developed several specification languages, software tools for computer-aided software design, and fundamental theory of software merging. [berzins@nps.edu]

Brian Womble—is the deputy for open architecture in the Office of the Deputy Assistant Secretary of the Navy for Research, Development, Test, and Evaluation. He spent the first half of his career working as a system developer in the telecommunications industry in Dallas, TX. In 2001, Womble joined Lockheed Martin in Manassas, VA, on Open Architecture efforts within the U.S. Navy Submarine ARCI program. He joined the Navy civilian workforce in 2009 and is now leading efforts to transition the Naval Enterprise to Open Systems Architecture through an Open Business Model. [brian.womble@navy.mil]

Abstract

Efforts to develop and implement automated test capability within the Department of Defense have resulted in the development of a number of tools. Literature from 2007 references use of automated testing to reduce the design cycle time of software, and it has been noted as one of several key components included as part of the more comprehensive plan for transforming the business and technical approaches to become more responsive to Fleet readiness requirements with the goal of providing more agile, integrated capabilities for the Navy by increasing supportability, standardization, system interoperability, network security, and Joint alignment.

This paper describes efforts to implement software systems automated test capability and an analysis of the results of the effort. The paper examines how well the automated test capability performed and evaluates the impact on system development time compared to systems developed using more traditional methods. In addition, a review of lessons learned and recommendations for further enhancements are discussed.

Overview: The Testing Challenge

Infinity Is a Big Place

The common slogan is that testing can demonstrate the presence of errors, but cannot guarantee their absence. This is valid in almost all situations. The exception is software services, for which the input space is small enough for exhaustive testing. Inputs include input parameters, initial states, and data read from files or input data streams. Practical limits for exhaustive testing are roughly one 32 bit input, which would require about 4 billion test cases. If we assume a 4GHz processor that takes 10 clock cycles per instruction on the average, that would take 10 seconds times the number of instructions

[1] The views presented is this paper are those of the authors and do not necessarily represent the views of DoD or its components.

needed to execute the service. For a service with a 32 bit input, that takes 1,000 instructions per execution—time for exhaustive testing would be about three hours; if the service needs a million instructions per execution, time for exhaustive testing becomes about four months.

For most practical services, the size of the input space far exceeds those estimated above, and in many cases, it may be unbounded, or bounded only by the capacity of the hardware. Since the size of the input space is exponential in the length of the input, this gets ridiculously intractable very fast: For services with just two 32-bit inputs, multiply the exhaustive testing time estimates above by 4 billion. Many practical services have much larger input spaces than that. Even though real computing hardware consists of finite state machines, the number of possible states in our machines is so large that it is not practically distinguishable from infinity.

There Will Be Bugs Left Behind

A consequence of the above analysis is that essentially all practical software systems are delivered with remaining imperfections.

However, not all faults are created equal: Every fault results in a failure for some subset of the input space. Since these spaces are finite, although very large, the failure spaces can be measured by the number of points in the failure space, or by the fraction of the input space occupied by the failure space. The latter fraction can be interpreted as the failure probability or failure rate associated with a given fault. Although the exact numbers involved are generally too large to be determined exactly, they can be estimated within given error tolerances by statistical sampling methods, and they can be used as a conceptual tool for classifying faults according to the associated failure rates.

The faults with the highest failure rate are those that produce a failure for all possible inputs. These are the faults with the highest impact on quality of service, and fortunately they are also the easiest to detect, since any single test case will detect all of them simultaneously. Faults with lower failure rates are increasingly difficult to detect via black box testing.

At the other extreme are the single-point failures: faults that result in a failure for only a single point in the input space, and produce correct results for all other input values. This category of failures is statistically invisible in practice, since a number of test cases close to exhaustive testing would be required to have an appreciable probability of detecting them in the absence of additional information about the fault.

Critical Bugs Are Must Fix

Failures are also not created equal: Some failures have more severe consequences than others. The critical bugs are those with the most severe consequences. Exposure is a combination of failure rate and severity of consequences. Severity of consequences can be measured with an abstraction known as “risk,” which can be estimated subjectively as a function of severity and failure rate, which can be interpreted as the expected likelihood of failure and relative overall cost to the Enterprise. In DoD contexts, severity of consequences has additional dimensions that include human injury and loss of life, in addition to financial loss. Risk is widely characterized as

$$R[f]\ (s, fr). \qquad (1)$$

MIL-STD-882E (DoD, 2012) recognizes that severities and failure rates for particular hazards are rarely known exactly, and provides an approximate method for ranking hazards by degree of risk exposure that depend on subjective qualitative assessment of severity and failure rates based on informally defined ranges. The standard also provides guidance on the level of authority needed to accept residual risks in each category of risk exposure. The practical result of this guidance is that the highest risks must be mitigated by measures that reduce the severity, failure rate, or both. This includes fixing the known faults with the highest risk exposure.

The weak point of current practice involves the word “known” in the previous sentence—all too often, faults become known only after at least one associated failure has occurred, along with associated undesired consequences. We would prefer faults with potentially severe consequences to be detected prior to fielding and actual occurrence of any failures due to those faults. This can be done by risk-based testing, which is done by automated testing whose intensity is determined by a risk analysis (Berzins, 2014).

Finding Critical Bugs Requires Cheating

Critical bugs may be statistically invisible. This is often the case for faults that are deliberately placed in the code by malicious insiders, such as Easter eggs and back doors. Such malicious additions to code are likely to be explicitly designed to produce statistically invisible single-point failures (see There Will Be Bugs Left Behind section). Such items are likely to be placed in services whose input spaces are much larger than the maximum size feasible for exhaustive testing.

For example, an Easter egg could be placed in a spreadsheet that would only be activated if a particular key was entered in a particular cell and all of the other cells were empty. Suppose that a cell can hold 10 characters and the spreadsheet can have 100 rows and 100 columns (most spreadsheets can accept much more data than this). If the testers know that only one cell is non-empty, then the number of possible input states is 10,000 * $2^{80} > 10^{28}$. The probability of detecting the Easter egg by black box testing using 4 billion random test cases (roughly the largest practical amount) is approximately 4 *10^{-19}, which is less than the likelihood of winning the grand prize in your favorite lottery twice in a row. Without knowing anything about the pattern that triggers the Easter egg, the number of possible input states becomes $2^{800,000}$ and the probability of detection by intensive black box testing would be less than $10^{-239,990}$, which is less than the chances of winning the grand prize in the lottery every day for the next $10^{26,650}$ years, even if we increase the number of test cases to the number that could be executed by all the computers in the world working for a century. To put that in perspective, the length of the winning streak is about $10^{26,640}$ times the age of our universe. This example is intended to illustrate that “statistically invisible” means “impossible to detect by black-box testing.”

Clear box testing can do better than black box testing in such cases, by using traditional coverage criteria, such as ensuring that every statement in the program has been executed for at least one test case. Running the usual test cases and keeping track of which statements have been executed, which can be done via instrumentation capabilities optionally provided by many compilers, will expose the rare paths in the code. Difficulties associated with covering the remaining statements include finding test inputs that exercise particular statements and determining whether remaining statements are in fact unreachable code. Although both problems are algorithmically unsolvable in the general case, for the kind of code encountered in practice, constraint solvers can succeed in synthesizing suitable test inputs for the majority of the cases and for identifying some of the unreachable code. The remaining code can be small enough to be singled out for human inspection.

Software and Hardware Are Never Finished

Successful systems always have long lists of pending change requests, including repairs for discovered faults and requested enhancements to functionality. In the Navy, such changes are typically implemented in technology upgrade cycles that occur every two or every four years, depending on the program. Each change has the possibility of introducing new bugs into the system, and each new release must therefore be re-tested. This implies that tests must be repeated many times during the lifetime of a typical system.

Improving Affordability by Automated Testing

In practice, testing accounts for a substantial fraction of the cost of developing each new release. An online game called the Massive Multiplayer Online War-game Leveraging the Internet (MMOWGLI) was run in two rounds during 2013. This web-based game involves large numbers of distributed players who interacted to encourage innovative thinking via crowd-sourcing, generate ideas for solving problems, and plan actions that carry out identified solutions.

The second round of the MMOWGLI game addressed the issue of reducing cost without reducing system quality or capability. Several of the highest-ranked action plans produced by the game included automated testing and retesting as part of the strategy for affordably ensuring system quality (Schmidt, 2014).

Testing Is a Design Requirement

Hardware Is Designed With Test Points

Computing hardware, particularly integrated circuits, is designed to include special interfaces for testing. The purpose of these interfaces is to provide observability and controllability of internal states of the circuit. These are necessary because internal points on the chip are physically inaccessible and because the yield of manufacturing processes is less than 100%. Uncontrollable variations in manufacturing conditions, such as imperfect alignment of lithography masks and imperfect printing and etching due to dust particles in the air and working fluids, result in fabricated geometries that deviate from the ideal design. Some fraction of these result in chips that do not behave as designed. Successful sales depend on rapid acceptance testing that separates the functional chips that can be sold from the damaged ones that must be discarded.

Testing of digital hardware is easier than testing software for many reasons, including the following:

- *Uniform state representation*. For the purposes of testing, circuit state can be usefully represented as fixed-length bit vectors. When the circuit is in testing mode, all internal state cells are configured into a long shift register that can be sequentially output through the pins for observability and input from the pins for controllability. This enables open loop testing, where each test sets the internal state to a specified value, executes chosen operations, and then the internal state is read out for analysis. This avoids the problem of finding input sequences that will drive a possibly faulty circuit into prescribed initial test states, reduces time to design test cases, and speeds up the actual execution of the tests. In contrast, software states are typically sensitive to the meaning of the data, which varies widely between applications. This precludes a one-size-fits-all solution to observability and controllability of internal states.
- *Known expected outputs*. Since hardware tests are looking for deviations from the designed behavior, expected outputs can be derived using a uniform and conceptually simple process: Simulate the logical design on the test inputs, and calculate the expected results. This process is typically

completely automated. Since software tests are looking for design faults, finding expected outputs is a much harder problem that does not have an easy, uniform solution and generally requires human creativity for each new application.

- *Effective error models*. The processes that introduce manufacturing defects are well understood and produce defects that are easy to characterize. The most common defects are voids in conductors (manufactured circuits lack connections that are present in the design) and bridging between adjacent conductors (manufactured circuits have extra connections that are not present in the design). The vast majority of hardware faults can be effectively detected by test sets that expose all single stuck-at faults, and practical algorithms for automatically constructing such test sets are known.

Software has much more complex failure patterns, and complete test sets for detecting such patterns are not algorithmically computable in the general case.[2] Observability and controllability for internal states of software are discussed in the next section.

Architecture Assessment for Testability

Software architectures can have a great impact on the effort required for system testing and the effort required to employ automated testing. Recent efforts by the Technical Reference Frameworks Working Group sponsored by ASN RDA have developed a structured set of testability levels to help assess these effects, shown in Table 1.

[2] This is a consequence of Rice’s theorem, a well-known undecidability property.

Table 1. Testability Levels

	Level	Description
0	inadequate	Does not meet requirements for any of the higher levels
1	syntactic	All services and data elements provided by each procurable component have published interfaces/data models that provide names and type signatures.
2	semantic	Published interfaces include precise definitions of the meaning of the services/data, including units, connection to real world objects, and requirements on outputs and final states resulting from all services.
3	robust	Published interfaces include all assumptions and restrictions on inputs and states, triggering conditions for all exceptions, and expected results after exceptions.
4	observable	All system attributes relevant to checking the requirements are observable either via the published operational interfaces or published augmented testing interfaces.
5	measurable	All properties needed to check the requirements have clearly defined measurement and evaluation procedures.
6	decidable	Pass/fail decisions for all test cases can be made entirely by automated procedures, without need for subjective human judgment.
7	unbounded	Any number of random test inputs can be automatically generated and corresponding test results can be automatically checked for all services.

Each level incorporates the requirements for all lower levels. The rationale for Table 1 can be explained as follows:

1. The externally observable behavior of a system consists of the services it provides to other systems. To enable independent testing of the system and its components, at a minimum the names of those services, the types of input data that each requires, and the types of data that each produces must be available to the testing team so that the services can be invoked by associated testing procedures. At level 1, this information should be specified as part of the system architecture to enable testing at each granularity level.
2. Although validation testing can be done at level 1, by relying on stakeholder review of each test output to judge adequacy of demonstrated behavior, verification testing requires level 2. Level 2 requires the architecture to include documented requirements for each service that are sufficiently precise to enable the testing team to make pass/fail decisions regarding the test outputs for each test case and service required at this level. This includes precise definitions of the properties of the real world that affect the requirements but may not be directly observable by the software. For example, a safety requirement in an aircraft control system typically specifies a minimum acceptable separation between aircraft. The requirement applies to the actual physical separation between the planes, rather than to the data visible to the control system software, which may differ because they are derived from sensors that can fail or produce inaccurate results. Level 2 may require human judgment for pass/fail decisions on test outputs, but those judgments can be made by the testing team, without requiring stakeholder participation in every test.

3. Level 3 includes documentation of all constraints, restrictions, and exceptional conditions associated with each service under test, not just the expected normal case behavior. This information is needed to check robustness of system operation, and can be used as a guideline for designing test cases focused on this issue. Level 3 implies complete coverage of the requirements, including both what the system is required to do and what it is required to avoid doing, regardless of whether inputs are in a “reasonable” range. For example, architectures at testability level 3 should include requirements on system input that would guard against SQL injection attacks; this information would support development of test cases that check what system behavior would result from such attacks.
4. At level 4, all system attributes relevant to checking the requirements are observable via standardized interfaces that are part of the architecture. This enables the software to be tested without modification by the test team, for example, without the need to manually add instrumentation code, and it enables test cases and test scripts to be portable across development and testing environments. Instrumentation is an issue in modern designs that use information hiding and object-oriented structures to limit access to internal system states. Access to some of these attributes may be needed for testing purposes, although they may not be needed during system operation, and their presence during system operation may not always be desirable due to the possibility of introducing cyber vulnerabilities. Therefore the testing interfaces may be excluded from the fielded version of the system, but they must conform to the documented standards whenever they are present. If instrumentation code is needed, architectures with testability level 4 include clearly documented standards for those testing interfaces. Ideally those standards enable automated instrumentation of the code in a repeatable manner that does not require human coding effort.
5. At level 5, all requirements are defined precisely enough so that all pass/fail decisions can be made based on defined criteria and measurement methods, without any expert human judgment required. Level 5 may still require human effort to apply the criteria, but they must be repeatable by anyone following a detailed written procedure.
6. At level 6, all requirements are defined precisely enough so that all pass/fail decisions for test cases can be made automatically, by software, firmware, or hardware. This implies that execution of test cases and assessment of test results can be completely automated at this level. Such automated tests can be repeated quickly at minimal cost, although up front setup costs for creating the automated decision procedures would be required. Level 6 differs from level 5 in that all of the criteria and measurement methods have been defined down to the granularity of basic operations that can be automated. Auxiliary instrumentation and communication links for physical attributes may be needed to support automated tests. For example, information from onboard GPS receivers may be needed to test separation between planes for the case in 2 above.
7. At level 7, test inputs can be randomly generated at need, based on probability distributions called operational profiles, and all pass/fail decisions can be automated. The difference between levels 6 and 7 is that 6 can be met via a list of expected outputs for a fixed set of test cases, while 7 requires a general pass/fail checking procedure that works for all possible inputs. This

level implies that additional test cases can be created without any human effort, which makes the marginal cost of additional test cases very small. At this level, very large test samples are affordable. This implies very high degrees of statistical confidence in system reliability can be achieved relative to the system workloads characterized by the operational profile distributions.

An additional test capability that can be provided at level 6 or 7 is built-in-test (BIT), which means that completely automated test capabilities are integrated into the deployed system, and can be invoked out in the field. Such capabilities may be used to ensure that all systems are functional prior to a mission or to recover from some types of equipment failures. If included in the architecture, such capabilities should include procedures for remedial action and be targeted at the most frequent expected failures. For example, a well-designed built-in test should be capable of diagnosing which part has failed, and issue instructions regarding what needs replacing and how to do it, or automatically switch to a backup system and issue a warning about the degraded status of the system. A very simple example of such a built-in test capability is a warning that a battery needs replacement or recharging, based on internal sensors. The benefit of including such capabilities is the ability to recover from some system failures out in the field, without the cost and delay of returning equipment to a home base for repair.

It is not necessarily useful to require every system service to have automated or built-in-test capabilities. Automated testing is generally beneficial only if the tests will be repeated often enough so that reduced marginal costs outweigh the extra setup cost of developing the automated test capabilities for each requirement. This is illustrated further in the following section.

An additional situation where automated testing is beneficial is in mitigating severe system risks, where high statistical confidence in particular system properties is required. That generally requires large numbers of randomly chosen test cases, which becomes affordable only at testability level 7. For example, guidance in MIL-STD-882-D (DoD, 2000) suggests that failure rates for mishaps with catastrophic consequences should not exceed 〖 10^{-6}, which requires roughly 20 million test cases if the probability of sampling error leading to a false positive conclusion must also be no more than 10^{-6}. See Berzins and Dailey (2009) for details on determining the number of test cases required to meet given statistical confidence levels.

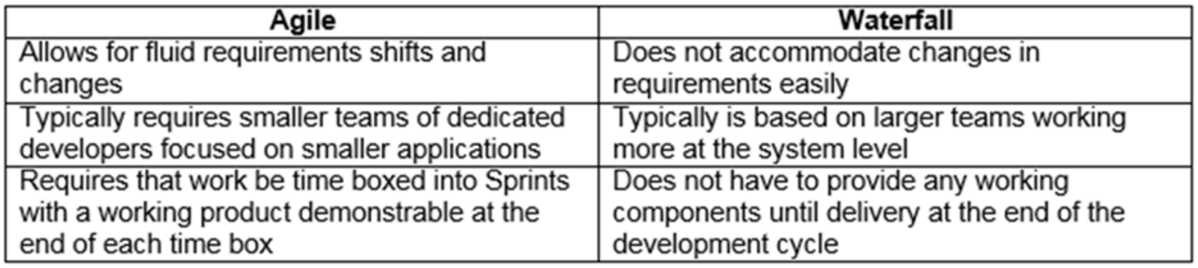
Experience With Automated Testing

Space and Naval Warfare Systems Center Pacific's Command and Intelligence Systems Division of the Command and Control Department is the author of a software development and acquisition initiative that is gaining momentum across the Navy and DoD. This initiative is no new big bang/silver bullet; it simply focuses on lowering the cost and risk of government-developed software by demanding closer government control of the baseline, focusing testing where needed, and streamlining processes to deliver capability faster by relying on agile development methods.

Historically, DoD software was developed utilizing Waterfall or Spiral development methods. A prime contractor would be awarded the contract to go and build applications, integrate them, and return once completed, for a major Development Test (DT). This process typically lasted anywhere from 12–36 months, and after testing, as much as 48 months, before a requested capability or version of software was fielded. Often the software would be released reliant on hardware that would not be backward supported, causing additional issues with the installation and fielding of the applications. The DoD strove to change this process and leverage the agile method to enable delivery of those applications faster.

Agile is a total paradigm shift for programs or development teams. Agile and Waterfall differ in many ways. A literature search reveals sources documenting many of the differences. Table 2 outlines several of the differences that are most relevant to DoD programs.

Table 2. Agile/Waterfall Compare and Contrast



Agile	Waterfall
Allows for fluid requirements shifts and changes	Does not accommodate changes in requirements easily
Typically requires smaller teams of dedicated developers focused on smaller applications	Typically is based on larger teams working more at the system level
Requires that work be time boxed into Sprints with a working product demonstrable at the end of each time box	Does not have to provide any working components until delivery at the end of the development cycle

There are many other differences between the two methods, but the last one in Table 2 is intriguing and one to explore.

In most DoD programs that require software, the applications are built and tested at the contractor facility while development occurs. The tests, called Contractor Tests (CTs), are typically the contractor's best guess as to how the products are going to be incorporated into a system and used in a workflow to support the user's needs. When contractors have the ability to interact with the end users to get a good feel for the end use workflow, testing is better and the products typically work more efficiently, but are still subject to major defects based on architectural changes or dependent application modifications that might hamper the applications from working correctly.

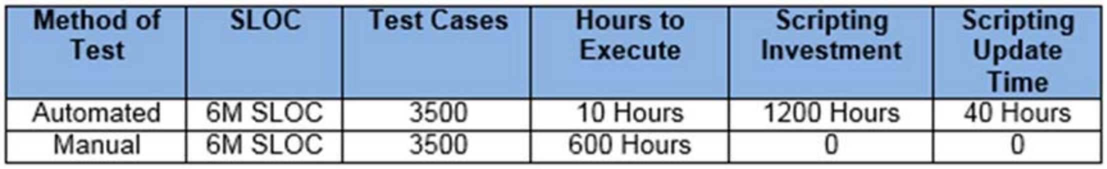
From this point, once built and tested in the developers' facility, the software is sent to the Integration Facility. The Rapid Integration and Test Environment (RITE) method infers that the Integration is conducted by a government facility or trusted agent, such as a Federally Funded Research and Development Center (FFRDC). Once the software is delivered to Integration, Integration Tests (ITs) are conducted. This process, according to the RITE method, is a graduated set of tests that begin with scanning and analysis of the source code delivered to the government-managed repository. The RITE Process requires source code be analyzed to ensure that it is built with suitable quality,[3] as defined by the program manager and Integration Team. Source code scanning, a waterfall technique adopted for use by agile, and "White Box" type testing of source code is a fundamental change to the way that DoD integration activities have conducted tests in the past. The ability for DoD entities to now look down to the source code line that is the root cause for the defect and point remediation to that specific line of code is very powerful and extremely helpful in managing precious taxpayer dollars in the ownership of software intense systems.

[3] Quality measurements are based on industry standards by referencing items such as SQALE, ISO, or IEEE documents as determined by the program manager.

The Integration Team then takes the software, once scanned and compiled, and begins deployment into a system development sandbox. These sandboxes allow the components to be "bolted" together and subsequently tested as dependent components, providing more of the end system execution environment. Automation testing at this stage is conducted based on CT artifacts that have been collected in previous deliveries that go into generating the Automation Regression Library. The beauty of this is that as system capability and applications mature, so does the automation regression library. Many would argue that automation and the development of scripting automated testing procedures is a heavy investment for many programs to undergo. It is true that the investment is substantial initially, but after the program has the foundation scripts generated, the investment in time is minimal to keep them updated delivery to delivery.

In some programs, our goal for automation was set to cover 65% of our testing via Automated Regression Scripts. As we quickly learned, this took approximately six months of dedicated, full-time effort for two personnel to script up the 65% of foundation scripts that we would utilize for a software intensive program, and then a dedicated 20% of two people's time each month updating the Regression Library. With a productivity factor of 100 hours per month for a test engineer for a given Sprint, 20 of those hours would be used to do nothing more than update the Regression Library. The payoff is that using these scripts allowed a program with a code base of over 6 million SLOC to run over 3,500 test cases in a period of less than 10 hours. This same set of test cases was run manually by six test engineers 10 hours per day for 20 consecutive days. See Table 3 for a breakdown.

Table 3. Automation vs. Manual Test Comparison



Method of Test	SLOC	Test Cases	Hours to Execute	Scripting Investment	Scripting Update Time
Automated	6M SLOC	3500	10 Hours	1200 Hours	40 Hours
Manual	6M SLOC	3500	600 Hours	0	0

By reviewing Table 3, one can see the investment in scripting hours over the period of six months to get adequate regression suite of test.

Additional analysis shows that this investment is quickly paid off in two months or Sprints of dedicated testing that would have otherwise had to be done manually, and that the Return On that Investment (ROI) is 560 hours times the number of months or Sprints required after that. This enables the test team to repurpose the 560 hours that would have gone to repeat the regression test, adequate time to focus on new functionality or fix defects from one delivery to another. The ability to focus assets on new functionality facilitates the ability of the testing to cover a greater breadth of system capability and help decrease the possibility of major defects being released once the system is delivered. The goal is to achieve as close to 100% test coverage as possible.

On any delivery, the software produced typically satisfies anywhere from 1 to 200 requirements. This can require a proportional number of test cases to plow through in the space of the 30 days allocated to a Sprint. The only foreseeable way to test this new developed capability sufficiently, while at the same time ensuring that no previous capability has been broken as a result of the newly delivered code, requires dedication of hours from the test team, affordable only through automation. GUI Drivers, such as AutoIT, and test orchestration platforms, such as Test Complete, enable linking of automation scripts to generate predictive test workflows that mimic real world task execution. Testing can be planned based on certain requirements or mission components linked together to operate in certain environments. Additionally, increasing occurrences of executed instances or increasing frequency of web service calls can generate load to test performance of a system.

One of the fundamental aspects of the RITE process is “focused testing.” Focused testing is accomplished in three phases. The first phase uses automated code quality analysis tools to scan software for Software Quality defects. These defects are relayed back to the developer upon identification for quick remediation. The second phase uses test scripts provided by the developer that are the foundation of automated functional testing to test services and component functionality build to build. The final phase uses human resources to manually test new features and develop automated scripts which will be used in future iterations of the focused testing process on successive software builds. Figure 1 shows the RITE recommended Testing Flow of Software through the Continuous Integration process.

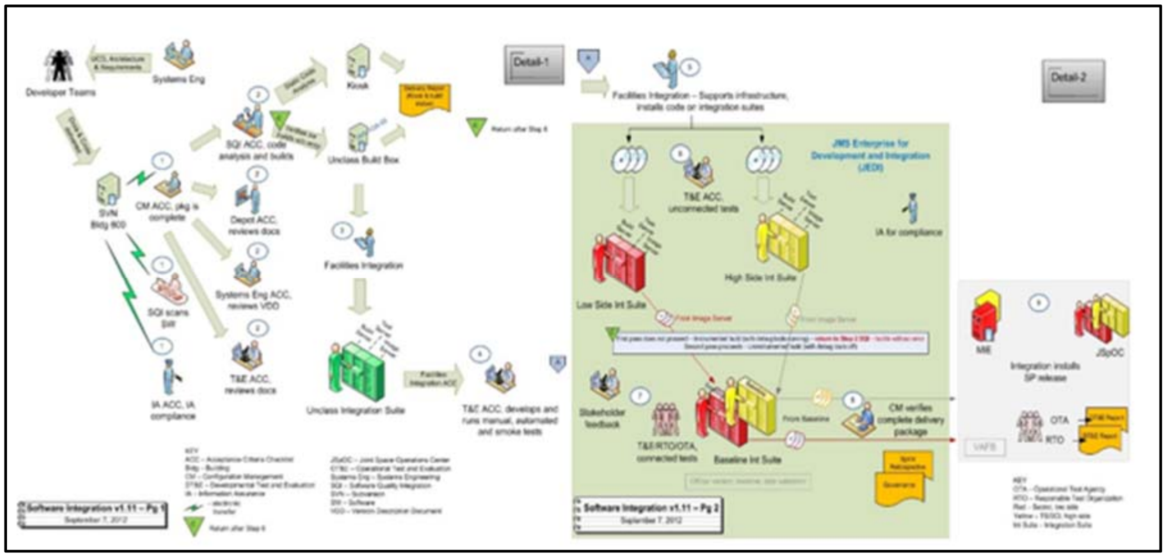


Figure 1. Continuous Integration Process Flow (Anchor A Continues the Process)

Figure 1 also shows the flow of test-required dependencies as a software component evolves through integration. Rigorous testing requires a certain level of graduation from the lowest level, source code, to the higher components, and through to the System level. This graduated testing helps catch defects sooner, and enables fixes prior to fielding where the corresponding costs are much higher.

See Figure 2 for a sample Defect Fix Cost Chart. Figure 2 shows that as the Product Lifecycle progresses, the costs of fixing defects rise. This is typically where DoD programs suffer. Programs that subscribe to the RITE process, working from contracts that generate certain Quality Requirements for software to be developed and received, would lower the rates of defects being produced. In addition, the focused testing element of RITE also helps detect early on any defects that could find their way into the delivered source code. Finally, the use of an evolving automated regression suite, run repeatedly through the development and integration process, helps in decreasing the time spent in the model conducting a formal Development Test at the completion of a version release.

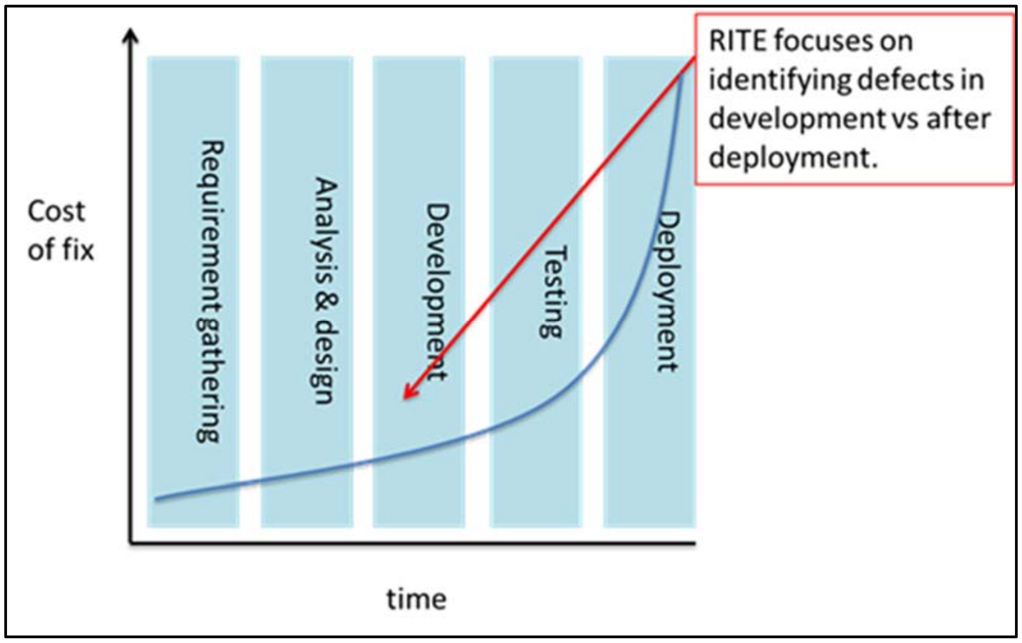


Figure 2. Defect Fix Cost

Conclusions

Automated testing has an important role to play in achieving affordable systems that reliably carry out their missions. This paper discusses automated testing, which primarily checks conformance of software behavior relative to system requirements. Other factors, such as quality of the requirements and the software architecture, are also relevant to system quality, and quality assurance techniques targeting those factors should be combined with automated testing for best results.

As explained in the Architecture Assessment for Testability section of this paper, effective automated testing depends on requirements that are both valid (capture the real needs of the stakeholders) and sufficiently well-defined to enable computing, whether particular test outputs conform to requirements or not. This is a challenge that will stress current requirements analysis processes, which typically produce natural language statements such as English descriptions of user needs. While such representations are needed for communication with people, they are insufficient by themselves for supporting automatic generation of test cases and automatic grading of test results. Natural language statements need to be augmented with more explicit representations that can support calculation of resulting truth values, such as logical assertions or the Object Constraint Language (OCL) associated with UML. This will require extra effort in requirements analysis, not only for coding requirements into these forms, but also for ensuring that the results are valid, and for refining the content of the requirements to provide sufficient definition detail and precision to carry out that encoding reliably. That extra effort is part of the initial investment needed to enable cost reduction by automated testing.

Complementary quality assurance processes are needed to ensure the quality of the software architecture and the subsystem requirements and specifications associated with that architecture. This is essential for affordably achieving reliability of large systems. A post-mortem analysis of software faults from the Voyager/Galileo programs found that the majority of the software faults were due to requirements and specification errors and misunderstanding of interfaces to external systems, not coding errors (Lutz, 1993). One of the essential quality attributes for the requirements and specifications associated with a software architecture is a degree of consistency sufficient to enable harmonious interoperability between the subsystems specified in the architecture, because its absence leads to expensive system integration problems.

References

Berzins, V. (2014). Combining risk analysis and slicing for test reduction in open architecture. In *Proceedings of the 11th Annual Acquisition Research Symposium* (pp. 199–210). Monterey, CA: Naval Postgraduate School.

Berzins, V., & Dailey, P. (2009). How to check if it is safe not to retest a component. In *Proceedings of the Sixth Annual Acquisition Research Symposium* (pp. 189–200). Monterey, CA: Naval Postgraduate School.

DoD. (2000). *Standard practice for system safety* (MIL-STD-882D). Retrieved from http://www.system-safety.org/Documents/MIL-STD-882D.pdf

DoD. (2012). *Standard practice system safety* (MIL-STD-882E). Retrieved from http://www.system-safety.org/Documents/MIL-STD-882E.pdf

Lutz, R. (1993, January). Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering* (pp. 126–133). San Diego, CA.

Schmidt, D. (2014, March). The importance of automated testing in open systems architecture initiatives [SEI Blog]. Retrieved from http://blog.sei.cmu.edu/post.cfm/importance-automated-testing-open-systems-architecture-062