Software Testing: A Life Cycle Perspective
How long can you wait for CMMI® Compliance?

Manage your software and systems engineering projects in guaranteed compliance with the CMMI NOW!

processMax 2i and 3i include all the necessary policies, procedures, guidelines, criteria, templates, and forms in role-based, step-by-step instructions, ready for use—everything you need for compliance with the CMMI-DEV at Level 2 and Level 3 respectively. Integrated with robust document management, workflow, and automated measurement and reporting, processMax is the intranet web-based solution for effective and efficient management of your software and systems engineering projects. processMax enables you to start operating in compliance with the CMMI immediately! We guarantee that organizations using processMax will not fail a formal appraisal led by an independent SEI-licensed appraiser.

pragma SYSTEMS CORPORATION
www.pragmasystems.com
877.838.PMAX
info@pragmasystems.com

GSA Schedule Contract No. GS-35F-0559S, processMax is a registered trademark of pragma SYSTEMS CORPORATION. Although processMax makes use of portions of "CMMI for Development, Version 1.2," CMU/SEI-2006-TR-008, copyright 2006 by Carnegie Mellon University, neither the Software Engineering Institute nor Carnegie Mellon University have reviewed or endorsed this product.

Copyright © 2008 pragma SYSTEMS CORPORATION.
This issue looks at software testing from a variety of perspectives but, when considering them in totality, some key points emerge. One point is that software complexity, as well as the need for flexibility, demands that we consider new approaches to testing. Author John Scott speaks to this issue as it relates to designing and testing open networked systems.

Future combat systems increasingly rely on interlinked systems (hardware, software and communications components) to gain advantage over adversaries. This interlinkage creates interdependencies among applications, components and systems which can lead to non-intuitive failures and potentially undesirable systems behaviors. These second, third or nth order effects are not predictable under current system design and management practices. If network centric warfare is the goal, current methods for design, testing, validation and verification must change from a group of stand-alone tasks to an integrated framework of technologies and processes that change as a system grows and evolves.

Design of an open networked system is different than anything DOD has previously undertaken. Previous warfare components were meant to operate within a specific operating plan with developed boundaries around how a component would operate within an operation. Network-centric systems, on the other hand, are specifically intended to interoperate with other systems. This distinction is important: acquiring components which are meant to be used in unintended ways will require a different design philosophy. Existing practices for design and testing of networked systems are not fixing the problem. In fact, they may be making it worse. Furthermore, if these issues are not solved in concert, the costs of developing new systems will escalate, as will risk to the war-fighter, resulting in ‘brittle’ systems and technology incompatibilities. The Department of Defense is concerned because as these systems become more interconnected and complex, the ability to understand (and control) their behavior with traditional methods is limited.

Another key point is summed up nicely by testing expert, Robin Goldsmith, who refers to testing as the “forgotten phase” of software development. He indicates that even though testing constitutes about half of development time, most formal academic programs that purport to prepare students for software careers don’t even offer courses on testing [1].

Author Brendan Wright, whose article describes competitive testing at TopCoder as a solution for achieving quality, has this to say about testing shortfalls.

All software systems of any normal complexity require careful attention to achieve quality. While quality demands range greatly depending on the intended usage of the system and the targeted quality, it must be planned or it will not be achieved. For software engineering a large part of quality assurance is driven by the testing efforts that occur throughout the software development lifecycle or (SDLC).

A good quality assurance plan is proactive and decomposes test efforts in conjunction with decomposition of design and development efforts. Unfortunately these activities are resource intensive and tend to be the first to get short changed when time or money is constrained. (Think of this as top-down driven non-testing.) At the smallest unit of decomposition a single developer is writing an object or function and the QA plan calls for some amount of unit testing - preferably automated. This kind of testing tends to be somewhat tedious and often feels pointless - “I just wrote it, I know it works, I don’t need to test it.” (Think of this as bottom-up driven non-testing.) The effect of both of these types of testing short comings leads to an over reliance on manual testing. Unfortunately manual testing tends to be limited in nature, as many QA organizations are shared across multiple projects and don’t have a full command of the systems requirements and architecture, and may further lack the skill sets to properly leverage scripting and automation. These factors all come together to create a chronic ‘under testing’ in the industry, which helps explain the difficulty in achieving quality in many systems and products.

Consider a hypothetical software project of ten resources that is behind schedule. Originally staffing had one full time QA engineer doing independent testing of completed functionality, another software engineer producing automated tests at the integration level and required unit testing of all code by the remaining team members. Additionally, one day every other week was dedicated to code review of any trouble spots or areas undergoing major change. Everyone who has worked in the industry knows what comes next: the team starts relaxing the rigor of testing tasks in the SDLC to get the last few functions done because of external and internal
pressures and when the product ships the quality is not acceptable. Certainly this is not always the case, but it happens with enough frequency that alternate approaches must be considered.

Author Capers Jones makes the point that there is value in tracking defect removal efficiency and “cumulative defect removal efficiency” as methods for evaluating testing effectiveness. He asserts that in spite of the existence of dozens of books and hundreds of articles on testing, there is very little information about how many test cases might be needed, how many test personnel might be needed, and how many defects are likely to be found and removed via testing. His article provides quantitative information on testing as well as other forms of defect removal such as inspections and static analysis.

Authors Elfriede Dustin, Bernie Gauf and Thom Garrett note that “simply stated, there is more software to test, with increasing complexity, more often, with fewer people. The current manual testing methods cannot keep pace.” They describe Automated Software Testing (AST) as a key technique that addresses some of the challenges software testers face today.

References
Network Centric Warfare, Total Systems Design & Testing

IMPLEMENTING A MARKET BASED APPROACH FOR ACQUISITION REFORM WOULD CHANGE THE OPERATIONAL PARADIGM FROM ONE OF PROCESS AND CONTROL TO ONE OF ACCOUNTABILITY AND EXECUTION, SAVING TIME AND MONEY OVER THE LONG RUN.

by John Scott, Defense Consultant

The Department of Defense (DOD) is moving away from stand-alone, component-driven, stove-piped, closed systems to system architectures that are interconnected, interdependent,1 reconfigurable and open systems-of-systems. This is particularly true with information technology intensive systems, which are the backbone of network-centric warfare (NCW) systems. Current doctrine is pushing network-centric warfare as a way to increase the lethality of the fighting force, specifically to “conduct simultaneous, distributed and parallel operations synergistically across the levels of warfare.”2

Distributed elements need to be able to seize the initiative, exploit success, and reconfigure themselves in response to dynamic threats and opportunities, guided by the commander’s intent. Further objectives include creating a “future Joint Force [that] will be capable of conducting and supporting distributed non-linear operations in a singular battlespace.”3

However, there is a shortfall with current design and testing methods as applied to geographically distributed systems. The new systems that DOD envisions for the future are systems that are open to new configurations and connections rather than being designed for only one set of missions. The design goal for new systems should be openness, modularity and reconfigurability to meet emerging and unknown future threats.

Emergent behaviors within a network can be useful or disastrous, subject to circumstance and the dynamics of the system. In addition, the network must tolerate a large degree of behavior variability without failure. To some extent, we want to modulate and influence the emergent behaviors present within a complex networked system. Only in this manner will collections of people, ideas, technology and processes be fully brought to bear against new problems.

Testing is currently applied as a milestone for components of combat systems. Testing validates the hardware construction, but cannot test the capabilities the hardware will provide once integrated into future military operations.

This reductionist view of design and testing is part of a greater problem with systems engineering: Simply designing and testing individual components (or even small groups of components) will not ensure that a dynamic network comprised of millions or billions of components will function as envisioned or enable adaptation of the network to future threats.

The problem of testing or validating these geospacally distributed systems for failure (or stress) becomes combinatorial when dealing with communications between components or groups of components. For example in Figure 1, if each hospital uses its own database schema the number of schema’s each hospital has to coordinate becomes combinatorial – as the number of databases goes up linearly, but the number of potential connections goes up with the square.

Emergent behaviors within a network can be useful or disastrous, subject to circumstance and the dynamics of the system. In addition, the network must tolerate a large degree of behavior variability without failure. To some extent, we want to modulate and influence the emergent behaviors present within a complex networked system. Only in this manner will collections of people, ideas, technology and processes be fully brought to bear against new problems.

Testing is currently applied as a milestone for components of combat systems. Testing validates the hardware construction, but cannot test the capabilities the hardware will provide once integrated into future military operations.

This reductionist view of design and testing is part of a greater problem with systems engineering: Simply designing and testing individual components (or even small groups of components) will not ensure that a dynamic network comprised of millions or billions of components will function as envisioned or enable adaptation of the network to future threats.

The problem of testing or validating these geospatially distributed systems for failure (or stress) becomes combinatorial when dealing with communications between components or groups of components. For example in Figure 1, if each hospital uses its own database schema the number of schema’s each hospital has to coordinate becomes combinatorial – as the number of databases goes up linearly, but the number of potential connections goes up with the square.

![Figure 1 Combinatorial, Schema Problem: If every hospital uses a different database schema, the interchange problem is combinatorial](image)

2 Joint Warfare and Conflict Resolution Perspectives
3 SecDef Donald H. Rumsfeld, Joint Operating Concept, Nov. 2003.

Data & Analysis Center for Software (DACS)
Furthermore, because networks of systems interact over time, the problem is even harder, since it isn’t possible to isolate a dynamic network for testing. One solution to the database schema problem is for everyone to agree to a single database format. But what happens when requirements, designs and systems capabilities are changing faster than the ability of groups to formalize decisions?

System testing has become a black-box affair, where the individual components are tested and then integrated into a system. Individually testing these components leads to optimization based on their individual functions not necessarily on the needs of the system as a whole. The problem is larger than just the ability to test individual components or an entire system. It may require a process change for how systems are acquired. Testing occurs in a larger DOD acquisitions ecosystem that exists to acquire, conceive, design, construct and maintain military systems. Any solution for validating and testing distributed systems must take into account the needs of these processes.

**Scope of Problem**

Existing design techniques do not allow for the evolution of complex systems. In fact, pressure to evolve often creates huge problems and spiraling costs, especially during upgrades or changes to a currently operating system.

One example of current practices failing on a massive scale is the Northeast power outage of August 2003. The interconnectedness and complexity of the Northeast power distribution system is extremely high, with electricity moving through the grid as need arises. The final report on the outage details four core causes of the blackout, three of which are pertinent here:

1. Inadequate system understanding;
2. Inadequate situational awareness; and
3. Inadequate reliability of diagnostics.

A key point is not that the failure can be traced to one individual component, but instead that the failure was caused by a confluence of environment, technology and process. Current system design techniques have created a system that is growing beyond our understanding, much less our ability to control. The report outlines proposed fixes, but these suffer from the same reductionist engineering paradigm and are a

Figure 2 - Possible Network Structures of an Air Transportation Network

(manufacturing, food, banking, etc.) that are somewhat insulated by changes in demand or supply due to network topologies, the NAS is a more complex mix of consumers, producers and suppliers, any one of which can disrupt travel service for all at a moments notice. The ‘brittleness’ of the NAS system is the Achilles heel of industry and the national economy - witness recent events, including:

- 9/11 add security incidents at airports (e.g. November 2001 Atlanta airport, July 2002 Los Angeles airport). One incident can cause major disruption or a halting of service for the entire country.
- Federal Aviation Administration: the effort to upgrade the Advanced Automation System to improve air traffic control system failed leading to a loss of 3-6 billion dollars between 1982 and 1994 and was abandoned without improvement of the system.

These examples of where the failures and vulnerabilities reside in systems can be well understood a-priori: But, what happens when there are massive networks of systems operating fully or semi autonomously? How can a design and testing process prove, certify or ensure that the systems of tomorrow operate as envisioned and meet the needs of the user community? This problem will only continue to grow as systems increase in size, scope of missions, interconnectedness and as systems change and evolve.

---


6 Ibid, Transportation

7 A system is defined as ‘brittle’ if small changes in its operating conditions, inputs, or other factors cause the system to rapidly fail and rather than continue operations and degrade slowly.
Trends

There are two important factors that should be taken into account as DOD moves towards developing new NCW systems: operational needs and the acquisition communities’ ability to meet those needs.

Collectively, these trends drive systems development. An example of increasing need is shown in Figure 3, which portrays the shortfall between the number of transmissions the National Security Agency intercepts each day and the number actually processed by analysts.

NSA isn’t even close to being able to keep up with the daily intercepted transmissions traffic. The current solution is to use new technology to filter and prioritize (and store), but the long term solution will not be sufficient unless there is a change in systems (people, process and technology) architecture.

One possible solution for NSA is to open the system and treat everyone like an analyst, enabling users to review and classify data that is important to them. Testing of distributed systems could one day be the same: users of military systems would run (or run autonomously) various combinations of tests to validate that sets of dynamic systems are operating within acceptable bounds for successful mission completion.

Systems design has much the same problem. Creating processes to predict future operating environments or more importantly, predict the ways the warfighter may wish to reconfigure a system is essentially a hopeless task – the number of possible configurations increases exponentially as new components are added.

These trends foreshadow that DOD will need systems (acquisition, design & testing) that can rapidly respond to changing threats and new operating environments, by incorporating:

- Open and modular architectures, that
- Reconfigure quickly, and are
- Geo-spatially distributed.

These issues should be solved in concert to meet the operational needs, limit the costs of developing new platforms and decrease the risk to the war-fighter.

Continual Disruption & Design

Colonel John Boyd was an innovator in the practice of how warfare is fought and is credited for the strategy leading to a quick victory in the first Iraq war and changes in the way our military fights battles. Boyd combined the philosophies of science, engineering and technology into the art of warfare.

Boyd makes the case that since there is incompleteness in our understanding of reality; there are inherent problems with designing and building systems based on that altogether incomplete picture of reality. In Boyd shorthand, the requirements and decision processes DOD uses to decide to acquire a system are inherently flawed, since the future base assumptions are wrong. This problem is only compounded by the time it takes to develop and refine requirements and progress through the approval process.

DOD is designing and building supersystems that are network centric warfare systems, but since we lack a robust foundation for acquisitions decisions, our transformation to NCW is essentially a leap of faith. The methodologies used to design, build and test NCW systems must change.

Boyd’s analysis leads to a key question: what types of systems should be built? Based on DOD’s vision of NCW, we can infer that future systems should be:

- Assembled quickly to meet the current threat.
- Systems should be constructed as simply as possible with open interfaces.
- Systems (and all componentry) should be modular.

---

8 OFT presentation, Transformation Defense, MORS Meeting, Jan. 27, 2004
Boyd established that having a higher rate of speed of decisions (as expressed in the OODA (observe, orient, decide, act) loop than your enemy wins battles and wars. Speed is a competitive advantage, not just on the battlefield, but also in the acquisitions and planning process. Somehow, these lessons should be incorporated into the NCW systems acquisitions process.

**Open vs. Closed Systems**

In Boyd’s paper “Destruction and Creation,” Boyd makes the case that the destruction and creation enabled by open systems will enable us to meet new threats, “unstructure, restructure, unstructure, restructure… the result is a changing and expanding universe of mental concepts matched to a changing and expanding universe of observed reality.” Creating a DOD acquisition, design and testing process that would allow for the enablement of open systems would solve some of the problems associated with responding to new and unknown threats.

An open system is one which exchanges energy with its environment, whereas in a closed system, there is no transfer of energy with the environment. Acquired DOD systems seem to have been operating under the principle that all systems should be closed, with no exchange of energy (or information) between systems, except on a very limited (i.e., stove-piped) and controlled basis. In contrast, open systems allow for a greater range of possibilities for adapting to a changing environment.

The trade-off is between adaptive potential and predictive capability. Open systems are easier to evolve, but they can be more difficult to control and accurately influence behavior. Closed systems can be tested, whereas tests of open systems are artificial or partial. In order to carry out any meaningful tests on an open system, artificial constraints must be placed on it which make it closed for the purposes of the test. This means that no uncontrolled inputs are allowed which might distort the behaviour of the test, rendering it unrepeatable.

**Current Systems Design**

Current design methodologies functions on the principle of analysis and creation of components and synthesis/integration of a system. While reductionism works when attempting to understand systems, it isn’t necessarily the best methodology for designing networked systems.

Systems engineering (SE) provides a methodology to create technological systems while using the systems approach. But current systems engineering practices for designing, building, operating and supporting future combat systems treat testing, verification and validation as milestones to pass rather than opportunities for continuous improvement. Classic SE processes (Figure 4) assume that systems requirements for evaluation and testing will be created (and followed) during the conceptual design phase.9 But performing testing as a milestone activity doesn’t ensure that the system will meet specifications or ensure the correct specifications are tested, i.e., requirements creep. Other methods, like spiral development where systems are incrementally improved and deployed over time, have the same issues of testing and V&V as milestones and not as an opportunity for improvement.

**Existing Methods**

The market for design and testing methodologies is very fragmented with tools that are used to test individual components and a reliance on good design to handle any incongruencies or problems for how the components interoperates.

Commercial design is typically handled the same way as in DOD: reductionism and analysis followed by synthesis and integration of components. The one major difference in private industry is the ability to make or readily buy components, enabling it to move much more quickly than DOD. DOD has no feedback mechanism for rewarding bottom-up innovation which often can better meet future needs more reliably than top-down management direction.

**US Government**

The process of identifying capabilities needed for the
warfighter is a multi-layered process that involves the combatant commands, OSD and the services. One problem with these processes is that they do not explicitly link the needs of commanders to approval of new acquisitions programs or approval must be given so far in advance that by the time a system is delivered, the need has changed. In some respects, the interests of the people in the field and the people in the offices are not aligned, since the incentive and risk/reward structures for these stakeholders are not the same. The process of allocating scarce resources via OSD-mandated management processes has slowed change and adoption of new capabilities in the military.

The US Government has a few initiatives to examine how large systems are designed and tested. These efforts tend to focus on highly specific problems: how to carry out computer intensive processing of simulations (the National Science Foundation) or how large systems can be certified for service (National Institute for Standards and technology).

**Private Industry**

Private sector companies do not have DOD’s problems providing capabilities because companies can use the marketplace to outsource or quickly acquire the capabilities they need. Private companies also tend to view formalized design and testing methodologies as extra cost, not an advantage, except among information technology companies, who accept it as an asset or a differentiator, but only as it applies to core functions for what the companies actually make money on. The approval process used by private industry for decisions to initiate a project focuses on three things: does this project save the company money (and have a return on investment) or make the company money, and what are the risks involved. Private companies also optimize their business operations around a few goals, whereas DOD isn’t able to optimize or maximize one or two resources and instead must retain spare capacity.

Companies also tend to have newer and more up-to-date technology and to squeeze as much value out of previously purchased systems as possible. Massively multi-player online game (MMOG) companies face some of the same issues that DOD faces from a design and testing perspective, specifically how to build the command and control infrastructure (both front and backend) to run these games.

One theme that pervades the MMOG market for testing is to test early, often and automate as much as possible. The testing of these systems is integrated from the beginning of the design process, when components are tested before being added to the systems. The system is continually tested afterwards.

Testing software code and services is the simpler part of testing; the harder part, much more akin to testing NCW services, is testing content for players. Since content is continually being added to games (much like NCW systems will be) there is a constant need to test, since content can change the game for players. For instance, if there is an economy within a MMOG, inserting new items for players to buy and sell can have unintended consequences.

MMOG’s have it a little easier than NCW systems; their systems may be complicated, but they are not necessarily complex. Since the code exists in one place (a cluster of servers), it is possible to take the entire set of code offline and perform any multiple of tests on it. MMOG’s are also built new and from the ground up, so designers are able to purchase and add whatever the latest and greatest is available in the marketplace. MMOG’s also rely to a great extent on super-users and beta testers to play the game and point out inconsistencies of issues and problems. These testers typically aren’t paid, but do receive status in the online game community.

**Alternative Methods**

Network centric systems are inherently different than stand alone systems in the operation and value proposition: their value lies not in how well one piece is able to complete its job, but in how well the constellation of pieces contributes to mission success. Another key feature with NCW systems is the necessary extent of DOD control over the distributed network. As we have seen with the Internet, understanding how to influence network behavior will be key for NCW to succeed.

A key lesson from Boyd is how to gain tactical advantage through innovation and disruption. These lessons should be applied to the DOD acquisitions process as well. Constant challenge of our assumptions and design principles is the key to building NCW systems.

Cutting down on the amount of time it takes to deploy systems would help. One solution is versioning and modularity. By deploying systems in pieces with different versions (and smaller block purchases), they can be put to use much more quickly than existing systems acquisitions methods. By focusing on designing systems that are modular, DOD can develop new concepts of operation to meet emerging threats. Speed is a competitive advantage in battle; it is only natural that we should push Boyd’s lessons into the acquisitions process.
Here are some examples to illustrate the need for a different mindset:

**Marketplaces, Niches & Hierarchies**

<table>
<thead>
<tr>
<th>Example/ Issue</th>
<th>Alternative Thinking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joint Tactical Radio System (JTRS): a $5 billion program to replace and deploy approximately 750,000 software enabled radios. A recent GAO report found that within JTRS, none of the critical 20 technologies were mature and that the number of drawings had tripled since 2004 implying that design isn’t stable.</td>
<td>Develop an architecture for versioning of the radios, allowing rapid deployment of current technology. This would have solved a need in the field and developed a user-base to gather feedback, test and deploy new versions.</td>
</tr>
</tbody>
</table>

In a relatively short timeframe, OSD was able to increase production (see Figure 6) of new armored Humvee’s from less than 20 a month (May 2003) to 500 a month (March 2005). Currently 61% of Humvee’s in the Iraqi theatre have been up-armored (this includes up-armor kits). This ability to use the commercial marketplace to rapidly respond to new threats (higher transaction rate) is the competitive advantage of the US. Instead of locking into one solution, DOD could have chosen to leverage the international marketplace investigating acquisition of Casspir, a South African built mine protected vehicle (MPV), recognized as one of the best urban assault vehicles in the world. The military is moving toward a services-based architecture, in which pieces can be connected in unintended ways to form new solutions to unimagined problems. The challenge is to build a marketplace that allows the Services to advocate their own systems (encourages competition of ideas) while ensuring the command can piece together robust solution sets to counter future threats.

The gap between a commander’s ability to carry out joint missions and the bureaucracy’s ability to acquire the fighting force could be addressed by the use of market mechanisms for the DOD acquisitions process. The fact that needs are not being met on the ground is evidence that our joint concepts are out of sync with actual capabilities. Centralizing the acquisitions process has not solved this problem and will not as long as our military continues to grow in size and complexity.

Top-down hierarchies imprint their characteristics onto the levels and systems encompassed below and within them. In contrast, bottom-up properties emerge from the interactions between components.

The acquisitions structure should more closely match the desired characteristics of the NCW systems DOD is trying to build. NCW systems should ultimately become a marketplace of capabilities open to users, who should be able to pick and chose those capabilities they require to complete a mission. A marketplace consisting of competitive and cooperative ideas and systems would ensure that the tenets of network centric warfare are embedded in DOD systems.

Ultimately, the future successes of the US military will depend on how quickly we are able to innovate both on and off the battlefield. A marketplace designed to churn through new ideas and kill bad ones can more effectively provide the joint capabilities needed by the commands. By building a marketplace, DOD could also bring to bear modern financial instruments, such as options theory, modular design methods, etc.

Implementing a market based approach for acquisition reform would change the operational paradigm from one of process and control to one of accountability and execution, saving time and money over the long run. Market mechanisms may not solve all of DOD’s acquisition issues, but they will help to create joint capabilities that make the military better suited to rapidly changing missions than it is with current acquisitions schemes. Research has shown that markets are much better at meeting needs and reacting to future change.

---

10 GAO-05-301 Assessments of Selected Major Weapons Programs
11 http://en.wikipedia.org/wiki/Casspir
than hierarchical systems of resource allocation.13

Marketplace Design

Architecture for a DOD marketplace would have multiple heterogeneous buyers and sellers, each competing to construct, obtain or coordinate the delivery of services. Buyers would contract with competing sellers for the right to provide a specific set or groups of products or services.

One way to set-up a marketplace would be to create an architecture in which individual Services and combatant commanders could cooperate to develop new systems. Rules that would be necessary in a DOD marketplace include:

- Standardization of cost data across DOD to allow users to assess delivery of capabilities across services.
- Allocation of funds to buyers
- Rules about how groups are able to invest in capabilities of other groups.
- Interoperability rules

One key to creating a viable and robust marketplace is the design of the rules that govern how resources are shared, bought, sold and acquired. These rules operate within and between hierarchies affecting how programs receive funding and govern how commanders make their wishes known through their transactions. The rules should be designed to account how they affect the whole market. One rule may govern how services and products are priced, while other rules may control the allocation of limited resources among participants with competing interests. However, pricing doesn’t have to be the only driving force. Other rules such as delivery, quality, etc. can be designed into a system to guide behavior.

Information Technology

Command and control systems are the glue that holds military forces together. One strong idea14 a federated grid of computing services combined with service-oriented architectures. “Service grids are not just a technical construct; they represent an emerging business ecology as well. As distributed service technology architecture becomes more pervasive, a rich mixture of services – some developed internally by companies and others offered by specialized third parties – will come together to create a robust market, shaped by federations of providers.”15

Instead of trying to devise and understand prior to construction all possible uses of NCW systems, service grids allow the network to form ecosystems of services and providers. In this model OSD’s management role becomes more of an orchestra conductor influencing action versus top-down control. One advantage DOD does have is control over its networks, so it can use lessons learned by private industry, not to invest or develop ideas that would produce less than desirable results.

Within marketplaces, there will always exist inequities. OSD might perceive that a need wasn’t being met, and they (as an outsider) could invest in a new capability and push for its adoption, just as any other Service or commander would. In this proposed service based world, there will also be a need to develop an archive capability to store (for use) applications or services that are no longer being supported by other groups. OSD could fill this gap or enable other groups to create services to fill the gap and receive support from users. In short, DOD systems needs have grown beyond the point where centralized top-down control is effective. The sooner that market mechanisms are put into place to distribute and encourage the ‘hive mind’ the better.

Lifecycle Methodologies

Open Source & Collaborative Tools

One method that has been shown to work in the private sector is the open source software movement. The general rules for open source software are that software that is derived from other open source software be open as well.

Open source is an excellent method to use as a distributed operation to meet a predefined group goal. As a method of software development, open source has shown itself to be an evolutionary process that can produce very complex pieces of software code. Open source may also be able to limit possible introduction of Trojan software into complex software systems; since the source code is open to all, ambiguous or useless code can more easily be found and removed. System vulnerabilities are also easier to find using open source because there are multiple people reviewing and revising code.

Open source is not limited to just a library of code and designs. Generally it requires a collaborative site where users, designers and engineers can gather to discuss and review new and existing designs. Tools such as wiki’s16, blogs and

---

13 “How Economists Can Get Alife,” Leigh Teaves
14 Service Grids, multiple articles by John Seeley Brown, Martin Milani and John Hagel
discussion boards could be implemented to ensure a distributed network of designers is working to deliver the greatest value to the government and more importantly, that the transient knowledge of how these systems are developed is captured, stored and processed for use later (all subject to appropriate levels of classification).

Deploying open source methods for DOD could be an effective means of evolving highly complex and valuable pieces of code, using the legal rights and authorities which DOD already possesses. In May of 2003, former OSD-CIO John Stenbit sent out a memo detailing how open-source software could be used and acquired for DOD. It has been two years since a DISA-commissioned MITRE Corporation study found that open-source software is already used widely within DOD. DOD already owns the software it pays to develop, and has the source rights - this is standard practice on every government contract for software development (vs. procuring products like Windows, etc.).

The next step is to treat DOD’s government source rights as open source within DOD as a matter of practice, since the legal rights already exist. If the goal for Network Centric Operations is to increase interoperability and increase the speed with which capabilities can be created and modified, transforming the “technology base” in this fashion is the most viable and effective option.

One DOD experiment is DARPA’s J-UCAS (Joint Unmanned Combat Air Systems) program. The J-UCAS program (see Figure 7) has selected two systems developers, Northup Grumman & Boeing, and a referee (Johns Hopkins Applied Physics Lab). DOD and industry understand how to develop and build airplanes. What is less understood is how to develop software that enables the UAVs to semi-autonomously coordinate their behaviors.

The J-UCAS contractual arrangement calls for both companies to develop different unmanned platforms in tandem, while collaborating and competing on the development of the control software. This is an extremely important experiment in business process, because the deliverable will be source code that DOD can readily reuse and/or adapt on other aeronautics projects.

Not all software implemented in the program will have government-rights-only code. As the J-UCAS director noted in a recent meeting, the largest cost of weapons platforms is the development of software-based systems. Using open-source methods will, over the long-term, make it more practical for the program to produce better code and systems while saving the government money.

Testing & Operations Strategy

Information technology is a catalyst that is forcing fundamental changes to how operations are carried out and how systems are designed. Rigorous testing of IT components should still occur, but the code should be shared and built upon open-source methods. Testing of integrated components will find only a small number of incompatibilities both with the systems and with other connecting systems, but it is worth completing.

Testing and monitoring of geo-distributed systems should be woven into the entire life-cycle of any proposed NCW system. A great deal of money, time and effort goes into testing systems (roughly 30% of development costs). We should be able to leverage those resources beyond simple one-time testing of a particular system or components. Instead, the tools used for testing should carry over (where possible) into a health and monitoring suite of tools. This makes better use of resources and ensures that new tools for monitoring systems build upon previous development.

Independent red teaming of systems could occur in two ways: indirectly and directly. Indirect implies that new software code can be developed from the foundation of code that has already been scrubbed in an open-source process. Direct red teaming means that as systems are developed (in the design and

---

16 A wiki is a web application that allows users to add and edit content, http://en.wikipedia.org/wiki/Wiki

17 May 9, 2004, AIAA Net-Centric Sub-Group Meeting, Dr. Mike Francis, DARPA
construction phase) the code be open to criticism, comment and general stress testing to see if it functions as envisioned. After delivery, a NCW system should be actively probed by machines and humans to understand how the NCW system is behaving (to develop models of behavior) and find those configurations where the system breaks. All data generated during these exercises should be fed back into the open-source system.

Conclusions

As we move from the industrial age into the information age, the nature of operations in warfare is changing. Instead of massing forces, forces are now networked and distributed, increasing options for a commander. We need new tools to understand, create and design technologies and operations in a distributed and networked battlefield.

At a recent conference18, an attendee observed that troops in Afghanistan and Iraq were greatly enjoying the capabilities provided by the robotic PakBots (by iRobot Corporation) and developing new uses beyond what the systems designers had conceived. But there was a problem: no logistics supply chain to provide support. The attendee bemoaned the fact that we shouldn’t deploy technologies into the field without a litany of check-offs or approvals. In the end, these industrial area processes obstruct the introduction of valuable new solutions. It is a much simpler problem for a half dozen engineers to figure out how to support and upgrade a technology already in use (and in demand) in the field versus trying to predict the needs of the troops in 5 to 15 years.

Being able to rapidly field new ideas and keep options open longer will prevent DOD from locking into one set of solutions or capabilities. Utilizing open-source methods within DOD would help decision-makers focus on mission capabilities instead of which specific solutions should be purchased. Systems developed using open-source methods allow new concepts to be more easily assembled and recombined for new and unknown missions.

Network-centric warfare, while often portrayed as a set of technical infrastructures or CONOPS, requires network-centric processes to achieve DOD goals. Processes, be they industrial-age acquisitions, testing methodologies, or open-source market mechanisms, are the DNA of what gets built, and the systems we get are inevitably an expression of that DNA. Without changing the genetic code of processes for development and testing of network-centric systems, we cannot transform the systems themselves. On the contrary, the cultural DNA of industrial age business process, which financially rewards closed systems, exclusionary development practices and milestone testing methodologies, will systematically produce antibodies to thwart network-centric ways of doing business. This immune response is ultimately baked into the systems themselves.

Instruments (open-source development methodologies and internal marketplace methods) hold promise as network-centric processes to promote the systems we envision, and allow DOD to meet future threats.

References

Alexander, T., Massively Multiplayer Game Development 2 (Game Development), Charles River Media, 2005
Bar-Yam, Y.; 1997; Dynamics of Complex Systems; Perseus
Bar-Yam, Y. 2002 Brief, Enlightened Evolutionary Engineering / Implementation of Innovation in FORCEnet, Report to Chief of Naval Operations Strategic Studies Group
Bar-Yam, Y., Unifying Principles in Complex Systems, in Converging Technology (NBIC) for Improving Human Performance, M. C. Roco and W. S. Bainbridge, Eds., in press
Boyd, J., Sept. 3, 1976, Destruction and Creation

18 2005 NDIA Test & Evaluation Conference
Acknowledgements

This work was supported under contract CLIN: N66604-05-M-0395 – Network Centric Warfare, for the Naval Undersea Warfare Center and the Office of Force Transformation.

About The Author

John Scott is a consultant for the Department of Defense and emerging technology companies in the Washington DC area and works for Mercury Federal Systems, Inc. He holds a MS in Systems Engineering from Virginia Polytechnic and State University as well as a BS in Mechanical Engineering from Lehigh University. He previously has held senior positions in technology companies as well as engineering consulting roles and positions in the US Navy Program Executive Offices. Previously, he has publicly presented at many US Government and Defense conferences as well as O’Reilly & Associates technology conferences.

Author Contact Information

Email: johnmscott@mindspring.com
Satisfy Your Quality Obsession

Software quality and reliability are mission critical. The size, pervasiveness, and complexity of today’s software can push your delivery dates and budgets to the edge. Streamline communication, improve traceability, achieve compliance, and deliver quality products with Seapine Software’s scalable, feature-rich application lifecycle management solutions:

- **TestTrack Pro** — Development workflow and issue management
- **TestTrack TCM** — Test case planning and tracking
- **Surround SCM** — Software configuration management
- **QA Wizard Pro** — Automated functional and regression testing

Designed for the most demanding software development and quality assurance environments, Seapine’s flexible cross-platform solutions adapt to the way your team works, delivering maximum productivity and saving you significant time and money.

GSA Advantage®
GSA Schedule 70 Contract GS-35F-0168U

Visit [www.seapine.com/gsa](http://www.seapine.com/gsa)
Competitive Testing: Eyes on the Quality Prize
OUT OF THE CHAOS OF COMPETITION EMERGES A MORE STABLE, PREDICTABLE PROCESS THAN A TRADITIONAL PROCESS THAT SEeks TO CONTROL THE UNCONTROLLABLE.

by Brendan Wright, TopCoder, Inc

This article will describe the nature of TopCoder’s model for software development; explain how to leverage a virtual workforce, competitive processes, and component reuse in both top down and bottom up testing activities; and examine the application of these concepts. Refer to Sidebars 1-3 respectively for an overview of the TopCoder environment, types of testing conducted within the TopCoder model, and a detailed description of the scope of competitions used by TopCoder.

Testing activities are expensive in the Software Development Lifecycle (SDLC). Unfortunately the value of testing (or cost of not testing) is easy to overlook, and so the testing activities, which are critical to ensure quality, are often shortchanged. This is not surprising for a traditional approach to completing the work required by the SDLC, where engineering resources are staffed, and therefore constrained. In seeking to improve the value of testing, and the overall follow through on well known valuable testing activities, non traditional approaches are needed. The nature of a virtual workforce combined with a robust methodology, for example TopCoder’s competitive, open processes and component driven SDLC, can bring several positive factors into play. The discrete atomic tasks (i.e. competitions) are engineered processes that require completion of the “menial” tasks (e.g. unit testing, code review) to receive payment. This prevents the backsliding of the testing effort required to ensure quality in each low level task. Due to the scalable nature of the virtual workforce test plans can be designed, implemented and maintained in parallel with the ‘important’ development, and the costs associated with developing these are lower. This is due to the competitive and global nature of a community such as that of TopCoder. Finally an SDLC that can actually deliver on a reuse program at the component level helps deliver much of the system in pre-tested blocks, thus lowering the overall cost of testing in any given system and ultimately enabling a faster time-to-quality for development.

What makes up a Software Development Platform?

At TopCoder there are three main differentiators from a traditional SDLC, a Virtual Workforce, Competitive Processes, and a Reusable Catalog. Collectively these things (and the systems required to manage and interact with them) make up the software development platform, henceforth “the aspects of the platform”.

The virtual workforce is characterized by its global nature and scale (the consistently growing 200,000 plus members hail from almost every country where there is an internet connection) and its elastic capacity. (Note that all work completed competitively is produced through competition by TopCoder members. Specifically, during a competition the members submitting solutions are “competitors”; the members evaluating solutions against scorecards are “reviewers”). The elastic capacity is seen at both an aggregate level and an individual level. At an aggregate level the community is able to respond to new demands rapidly. Several years ago a new product line of competitions, branded “studio” was introduced. Studio competitions are focused on the creative skill sets (such as image creation, human factors, and idea generation) initially relatively few members competed in this area, but now the supply of members in the studio area allows for all creative work to be run through these competitions. Individually talented members tend to adapt to the demand based on where they can maximize earnings for their skill set. Developers who only have worked on Component Development may try out and excel at assemblies if they see the grass is greener.

Competitive processes are used to manage the work completed by the virtual workforce. By using competition (instead of assigning one person to one task) several benefits are achieved. Firstly for any given task multiple people are involved in solving it, which results in a broader range of solutions and ideas. Of course some people are better than others. Competitive testing involves rewarding the best solutions (and people) and this encourages the best to come to the surface. Competitive testing also provides an avenue for sharing the process with a wider audience, which is key in software development. Finally, competitive testing is a great way to build a virtual community, and TopCoder is built around a community. It was founded on the belief that people working together in a virtual environment can produce better results than a traditional model. In TopCoder’s case the community and the virtual workforce have grown exponentially.

The Reusable Catalog is another key aspect of TopCoder. The Catalog is a system of reusable components that developers can use to complete their work. The Catalog is a tiered structure that allows developers to layer components together to build complex systems. The Catalog is accessible through an API, which allows developers to interact with the components in a flexible way. This allows developers to create custom solutions that are tailored to their specific needs. The Catalog also provides a way to share components with other developers, which is beneficial for both developers and users. Overall, the Reusable Catalog is a powerful tool that helps developers build better systems more efficiently. It also helps to ensure the quality of the systems, as developers are more likely to reuse high-quality components when they have access to a large and diverse set of reusable components.

1 See http://soa.sys-con.com/node/713244?page=0.1
2 Before Studio, TopCoder staffed numerous creative resources to complete this work, as the studio membership developed these staffed resources transitioned from completing the work, managing the studio competitions, greatly increasing their leverage in terms of projects completed and customers supported.
Sidebar 1 - TopCoder’s History and SDLC Model

Since 2001, TopCoder has been building a commercial online platform for software development – a virtual software factory. TopCoder has built along with this platform, in parallel to it, a virtual workforce in the form of a global programming community with more than 210,000 members —independent developers attracted to a professional and collegial environment which offers a variety of incentives for involvement – financial, educational, notoriety and simply fun.

Discreet tasks in the SDLC are mapped to competitions (see Sidebar 3 for details). Competitions are defined by their inputs (competition specifications,) and their outputs (deliverables). Almost all competitions share some common features:

1. Competition specifications (along with the rules of the competition) are posted to the TopCoder site (sometimes access is restricted to members with signed NDAs or background checks)

2. Members opt-in by registering to compete.

3. Open interaction between competitors and managers: The here the manager is whoever is responsible for the competition, usually a TopCoder employee, another member, or a customer.

4. Submissions are posted by competitors.

5. Submissions are screened to ensure they have the minimum requirements, this is done by a peer in the member base, by automated inspection of the submission, or a combination of both.

6. Submissions are reviewed by other members (i.e. peers) using a standard scorecard (in some cases, usually creative work, the customer may perform review)

7. Winners are selected based on the highest scored submission. Often the reviewers will have identified some final fixes that the winner is required to complete before they can receive full payment.

A typical customer will license the TopCoder platform for a one year period and pay a monthly subscription fee. Platform Managers – TopCoder employees that help customers with usage of the platform – then help customers map new work into a series of competitions. This mapping is called a game plan and identifies (roughly) what competitions are needed and when. As competitions are run the game plan is continuously updated. Customers pay a standard fee per competition; this fee includes the costs of the members (including first and second place prizes, as well as review fee). TopCoder then pays the winner typically 50 percent of the prize when a winner is selected, and the remainder is paid after any final fixes are completed. Members can also win prizes for their overall body of work: members can win money for their overall body of work in Digital Runs, they can win reliability bonuses for always completing work they sign up for, and they can qualify for the TopCoder Open where $150,000 in prizes are routinely awarded.

Although the above steps are relatively simple, there is a great deal of complexity within this process. Significant problems had to be considered and solved before any of this methodology would work: How do you make it fair? How does the SDLC break down into smaller steps? Who should be allowed to do reviews? How do you provide the right incentives for people to compete? As these broad questions have been resolved and solutions increasingly refined, TopCoder has applied similar lines of questioning to each type of competition: Where does the competition spec come from? What’s the right set of deliverables? What questions should be asked during review? How should they be weighted? Through process engineering, feedback from customers, and input from members, a successful quality assurance approach should constantly be evolving and engaging all stakeholders and should continue to refine the competitions it runs. As TopCoder has continued to move more and more of the SDLC tasks to the member base, more lines of competitions have come into existence. Originally only software components were built through competition, but then integration work was being done by assembly competitions, and now competitions exist for developing business requirements (conceptualization), architecture and design, creative tasks, and of course testing. The strength of the TopCoder model comes from competition and community. The competition helps keep pricing competitive, increases the discrete quality of any deliverable produced, and eliminates points of failure in the overall project, by having multiple resources competing for any individual task. The community brings together a globally distributed talent base to fuel the competitions. In the end the SDLC can be thought of as a fractal - each competition is a bit of randomness following a standard pattern, and in aggregate the SDLC is an ordered shape. Out of the chaos of competition emerges a more stable, predictable process than a traditional process that seeks to control the uncontrollable. For customer testimonies regarding the TopCoder process visit the following:

http://industry.bnet.com/technology/1000470/aol-finds-topcoder-tops/

http://www.computerworld.com/industryttopics/financial/story/0,10801,98834,00.html

---

1 See http://www.topcoder.com/dh

2 See http://www.topcoder.com/wiki/display/tc/Component+Design+Reliability+Ratings

3 See http://www.topcoder.com/tc?module=Static&d1=tournaments&d2=tco09&d3=overview&d4=overview
competing, so a task is not dependent on an individual's success or failure. The risk of something outside an individual's control preventing that individual from completing the task is unacceptably high, but when paying for level of effort there is no alternative. It is cost prohibitive to pay multiple resources for their time, assign them all to the same task and only use the best solution. Another benefit of competitive processes is the upward pressure on quality; the normal incentives associated with competition push each submitter to deliver the best possible solution. Another important aspect of competition is openness about the requirements to compete, the rules for picking winners, and the results of the competition itself. No one likes an unfair competition and openness helps ensure competition remains fair.

The reusable catalog is primarily made up of Java and .NET software components. These represent standard libraries, widgets, frameworks, and tools that are often redeveloped for each application in a traditional model. In order to ensure reusability of the asset, the standards for a component are well defined and decomposing each system into components is an artifact of the SDLC. Beyond software components, other assets, such as common architectures, algorithms, web services, etc. can be added to the reuse catalog whenever doing so makes sense.³ Some considerations are the cost of maintenance of the asset, if the asset is proprietary to a customer, and what is required for the asset to be reusable (e.g. creating standards for documentation, adding non-application specific API elements, etc.).

The virtual workforce, competitive processes, and catalog form a positive feedback loop. It would be very difficult, both from a cost and management perspective, to hire two employees and then have them compete on writing parts of the system, but with a virtual workforce competition is not only possible, it actually attracts new members. Without the scale of a virtual workforce and the quality and reliability delivered by competition, it would be extremely difficult to build new software applications as reusable components and ensure usage of preexisting components in an application – but both of these other aspects make the catalog thrive. And in turn having a software catalog means less new software is built for each new system, reducing the build costs of customers, which means customers can run more work through the platform, which again helps grow the workforce. Having anyone of these aspects is useful, but having all three makes the platform greater than the sum of its parts.

Using the Platform’s Tools to Test

Testing can be incorporated into a mature and disciplined SDLC in two manners, both well known to software engineers - “top down” and “bottom up”. By top down I mean the act of decomposition; by bottom up I mean aggregation. An SDLC should be designed to support the development of systems and the subsequent maintenance. At the top level there are several areas that are likely seen as related, but relatively independent such as development and testing. The breakout of testing at the top level leads to the top down incorporation of testing activities. As the other tracks are broken down, especially design and development, they eventually become fine grained steps that can be handed off to a resource (an engineer, a competition). The incorporation of testing into these tasks (independent of the top down testing) is the bottom up incorporation of testing activities. Both approaches are required for effective QA in an SDLC. Sidebar 2 lists types of testing that are incorporated within the competitions and Sidebar 3 describes the various types of competitions and the processes inherent in some of them.

The specifics of TopCoder’s SDLC are highly influenced by

the availability of the platform. Like most traditional models, a high level track for testing runs parallel to the development (i.e. coding) track. The flow is basically requirements map to test scenarios map to automated test cases, with manual testing verifying all tests are run, plus running any manual test cases and performing ad hoc testing. A traditional model is constrained by the nature of managing staffed resources; there will always be pressure to reduce both the quantity of hours and quality of resources spent on the tasks related to these activities. The platform approach provides a completely variable resource model; the large global pool of talent responds to market forces and is elastic in response to demand. In order to effectively harness this capacity, it is critical that tasks be based on competition and not level of effort. Resource competition ensures both successful completion of the project by allowing multiple resources to work on a single task (without raising the cost of the task), and provides a downward pressure on the prize money required to attract talent to the task. The downward pressure comes primarily from two mechanisms: firstly because an elite talent can complete a task in substantially less time than a mediocre talent, the effective hourly rate for the winner remains high, secondly the global nature of the member base automatically brings in cost savings from geography shifting. (However unlike off shoring initiatives, there is no predetermined winner based on geography.) Without competition an upward pressure on the cost of a task would be unchecked. There are other less obvious benefits that come into play too. For example when deciding whether to automate a test case: in a traditional model one would consider how hard (expensive) the initial coding will be and how expensive the maintenance of that code will be versus the cost of manually performing the test and the value of rerunning the test on demand. With the platform model the question becomes can this be automated? Since the cost of development is marginal compared to the cost of a tester manually carrying out the task, it becomes irrelevant whether the test will be rerun and if the maintenance is “too expensive.” (We'll also see how maintenance can be addressed later in bottom up analysis.) Furthermore, in a traditional model there is a choice between leveraging the tester resource or the developer resource for these approaches. If the developer has other tasks the cost of his development time is not only the resource cost, but also the opportunity cost of not completing that other task. The decision in a traditional model is “who can work on

4 It is not a goal of TopCoder to pay resources less; rather it’s to pay fair market value for quality work in a manner that allows for redundancy in the resources working on tasks. As a matter of fact the elite members tend to make substantially more than they could in a traditional corporate job. For an example look at the earnings in this member profile http://www.topcoder.com/tc/module=MemberProfile&cr=287614

Sidebar 2 - About Testing Activities and Types of Tests

**Review** – Although not always thought of as a testing activity it’s the cornerstone of the TopCoder SDLC. Review is leveraged in each competition not only to identify the winner, but also identify any final fixes required to perfect the solution. Additionally for competitions where working code is not the output, review is often the only avenue to “test” the solution.

**Automated Testing** – Test cases that are implemented via software or script. The primary advantage of automating tests is that there is only a one time cost plus maintenance to keep the test up to date, and each execution of the test thereafter is “free”. Automated tests can then be incorporated into regression tests or used by a continuous integration server to provide extra value.

**Manual Testing** – Test cases that are carried out by a person. Throughout the SDLC competitors and reviewers will use manual testing as needed to augment the required automated tests. Additionally manual testing is more heavily relied on during the certification iteration (see Figure 1).

**Unit Test** – A bottom up test produced by a developer to codify their assumptions and understanding of how a method should function, typically written in the JUnit or NUnit. Each competition that produces production software (Assembly and Component Development) requires a developer to provide complete code coverage with unit tests.

**Integration Test** – These tests probe the system boundaries by executing system functionality. These tests are identified as test scenarios during the test plan competition.

**Performance Test** - These tests probe the system performance in a production (or near production) environment under normal and worst case loads. In addition to the performance tests identified by the test plan, performance tests are also developed at component scope during review of component development.

**Ad Hoc Test** – Any test not identified by the test plan, which can be carried out at any point during the SDLC when a competitor or reviewer identifies a need. These are also more heavily used during certification to augment the SDLC tests, usually by a customer’s QA team or through crowd sourced testing (i.e. the Bug Hunt competition.)

**Acceptance Test** – User verification of the initial objectives and expectations, carried out by the intended users of the system or their proxies during certification.
this and what are they not working on if they do?” instead of “should we do this work?” The platform model thus aligns the decisions of how to decompose test cases to be greater aligned with the business.

These same themes are also apparent when looking at how testing activities can be introduced in a bottom up manner. For some time now code reviews and unit testing have been well known quality improvers for an SDLC. However, many software teams still struggle to fit the time for these types of activities into the schedule because feature sets are more visible and thus development of new functionality is given higher priority than testing activities. As described above in bringing resources to bear on the testing activities typically neglected, review can be incorporated into each task with a variable model since the work capacity is elastic to the demand. Beyond the workforce, the competition enforces the use of review – without review competitors’ submissions cannot be ranked, therefore no winner could be selected. Additionally, a competition helps enforce completion of more mundane development activities like writing unit tests. Leveraging a quantitative objective measurement, e.g. automated code coverage analysis, allows for automated screening of submissions, prior to qualitative review via scorecards. This mechanism could be used to reject a check-in by a developer in a traditional staffing model, but that just prevents the bad code from going in. It doesn’t yield alternative solutions that can be used instead of the bad code. Another advantage of competition is that the often unglamorous testing activities (e.g. unit testing mentioned above, and also maintenance of certain testing deliverables) can be incorporated into more tasks that have larger talent pools. An example of this is that component development will require a solution including component unit tests, and similarly an assembly competition integrating new functionality would require that any application level tests broken be updated. One other bottom up benefit is the reuse of software components from previous projects within new projects. This turns the act of deploying an earlier system that uses a component into a testing activity for future systems that use the same component.5

Resolving Challenges of Competitive Testing
There are certainly a number of areas where issues still arise. First and foremost is the ROI of test development. A process that develops a ratio of three lines of test code for each production line of code, certainly is not “cheap”. Another area where ROI must be considered is replication of work during the top down development of test plans and automation versus the bottom up activities of unit testing and review. Finally, the maintenance of test code can become an issue for organizations that have limited resources to work on testing activities. (After all in most organizations resource staffing is a zero sum game, and any effort spent maintaining tests is taken from creating or testing new functionality.) Fortunately, changing the calculus of fixed based resource tasking allows these concerns to be mitigated.

A global, scalable pool of talent can be brought to bear on any task, including testing tasks. In a traditional model not only is the talent pool fixed, by definition there are only so many top engineering talents to go around. In a virtual workforce, resources perform only the tasks they opt to do; and with competition there is no need to know a priori which resource is elite – by winning the competition a person proves the quality of their deliverable. By using metrics like reliability (how often does a person complete a task of this type after registering) and rating (how good are they at this task) - one can project the likelihood of any single resource completing the task. However competition adds another aspect of assurance; multiple resources can be brought to bear on an identical task, while incurring only a marginal cost. This does not mean any task can be completed at any price point; in fact one thing learned early on is that if the scope and compensation are not in balance - a competition will almost always fail. While competition and a global virtual workforce stabilize prices by creating a downward pressure, they don’t eliminate the upward pressures.10

Cost can also be distributed across a wider set of customers. In practice TopCoder has seen generic enterprise components leveraged for up to 50 percent or more of a new application’s code base. In this case these existing components are similar to open source software, with a variety of customer users ‘hardening’ the product and a developer community supporting the maintenance, and the common process and standard deliverables associated with each component.6

---

5 TopCoder is currently rolling out a new “crowdsourced” testing offering called “Bug Hunts” which let a client make their application available to the community and have members perform ad-hoc testing to identify outstanding issues.

---

6 Typically a resource would be a single developer or tester, but also it could be an outsource partner or other entity to which tasks are assigned. For tasks completed by TopCoder individual members are the competing resources.

7 See the member profile linked above

8 See the member profile linked above

9 The actual cost of redundancy is somewhat higher – TopCoder typically pays a consolation prize for the second place winner and also pays reviewers. Beyond that TopCoder uses monthly Digital Run prizes to reward frequent competitors that score well on competitions, even when they may have placed lower. Even so the cost of any discreet task tends to be lower than the cost of managed labor.

10 When deciding how much effort to put into testing – the intended use of the application has to be considered. Testing to ensure “five 9” up time in a demo product to raise venture funding wouldn’t make sense, but for an application running major pieces of the internet it does.
Sidebar 3 - About TopCoder Competitions

The general flow of each competition is standard across all types of competitions. Each competition requires a specification (along with any supporting documents) to be posted to the community. The community is then engaged to review that posting and validate the scope, price, and timeline. If the specification review fails, it will be updated based on the community’s feedback.

The competition portion involves registration of members as submitters and reviewers, submission of solutions by one or more competitors, and review of the submissions. Each review follows a standardized scorecard. Submitters are given an opportunity to appeal the reviewers’ decisions on each scorecard question. Once appeals are finished a winner is determined by the highest aggregate score.

In the event none of the submitters passed review (i.e. achieved a minimum aggregate score) then the competition will have “failed.” Failure rate is fairly low (less than 10%), but when it does happen the competition will be rerun, usually with reduced scope or increased prize. Otherwise the winner is obligated to resolve any outstanding issues from review, and one of the reviewers will perform final review to ensure these changes are completed. This cycle is known as “Final Fixes” in the TopCoder SDLC.

**Studio Ideation** – Brainstorming for new product ideas, typically contestants can submit multiple ideas against a general guideline provided by customers. Customers can select multiple ideas to purchase and develop.

**Studio Storyboard** – Develops the look and feel elements for a GUI. This is used most frequently to “brand” a web applications HTML pages, coming up with the color schemes, styles, and graphical elements for a website. It can also be used to develop traditional print graphics (all the print artwork hanging at TopCoder Headquarters was produced this way, as well as the T-Shirts given out at the TopCoder Open.)

**Studio Wireframes** – Wireframes are used to establish the page flow, user inputs, and content of each page of an application, without regard to the specific styles of the application or a given page. The wireframe and storyboard compliment each other in helping define the user experience quickly before a prototype is built.

**Algorithm** – This is the first TopCoder competition and most popular, although it isn’t used for producing production software, it’s largely responsible for attracting new members to the TopCoder community.

**Marathon Match** – A long running algorithm match which can be used for R&D purposes as well as to improve performance of existing algorithms. Recently the algorithm in an open source image compression routine was improved 300% using this competition.

**Conceptualization** – This competition takes an idea and refines it into a set of business requirements defined at the use case level. Customers complete a questionnaire about the product, and answer questions from competitors in the forum. This competition can produce in a single week what normally takes a month or more.

**Architecture** – This contest compliments the conceptualization by defining the multiple views that will constrain how the application will be built.

**Module Specification and Design** – These contests take the work of a single vertical iteration and capture the detail requirements for that module, then break the module into software components and assemblies.
Assembly – This is a versatile contest that can be used for as-hoc software development tasks, typically prototype conversion (i.e. turning a storyboard and wireframe into a functioning front end) and integration of software components and system interfaces.

Component Design – Components make up the majority of the LOC (averaging 80%) in applications built by TopCoder. Component Design produce the written and UML design specifications that ensure the component follows good object oriented design practices, correctly leverage technologies, and will be reusable.

Component Development – The contests pick up where Component Design leaves off. The components actual production code and unit test code is produced (usually resulting in 3:1 test code to production code ratio.)

Test Plan – This is similar to the component design, in that it identifies all the tests that must be performed to ensure system quality. Typically a range of test scenarios will be completed; scoped at either application or module level. Each test scenario is made up of information like data setup, execution steps, input set(s), and expected results.

Test Automation – For a given Test Plan this competition will automate all the designated test scenarios. In some cases a test scenario may be identified in which case it’s not automated.

Bug Race – This is a quick strike capability for dealing with urgent or one off issues with smaller scopes. The first member to submit a passing submission wins, so speed is a critical aspect.
Firstly, the process itself promises a higher degree of confidence can be placed in the initial quality of a produced component. The same process is used and refined, like a factory that produces a product within a standard deviation and this promotes initial quality. The net result is the community of developers, i.e. the virtual workforce, is fungible between the components in the catalog, even when there may have been little or no exposure to a given component previously.

Finally, the stringent review process required for running a competitive methodology also means each and every task undergoes immediate scrutiny. For the unit tests of code provided by development competitors, reviewers immediately confirm if sufficient level of test is performed. This not only includes a quantitative metric driven analysis - like a minimum code coverage percentage, but also a qualitative analysis: Are the tests written appropriately? Do they accurately verify boundary cases? etc. Another aspect of the review is that it ferrets out requirement ambiguities. When reviewers construct tests for a software component they implicitly codify their assumptions and understanding of the written requirements. The same is true of a developer with his/her production code. When reviewers then run their tests, divergent understanding of requirements is brought to the forefront. This creates a positive feedback loop: as conflicts are identified and escalated to the customer\textsuperscript{11} – they are resolved, allowing the reviewer and developer to sync their work products with the customer’s intention and to alert the customer of problems with their requirements. These tests then become a regression base for future versions of the component (whether in the current iteration or a future one.)

Conclusion

In summary, there is no short cut to quality, and this means significant effort during the SDLC will be spent on testing activities. Regardless of the functionality desired, if it’s not encapsulated in a quality product, it’s not valuable. Although the problems of software engineering using a traditional approach can be mitigated – they cannot be totally eliminated. As long as resources are constrained the lower priority tasks will be shortchanged once problems arise. It is possible to have a mature development team that insists on completing testing activities and doesn’t cave in to the natural pressure to gain more functionality at the expense of quality. It’s also possible to have sophisticated project management that prevents an immature development team from short changing the testing activities in order to meet a release date. In either case the problem is simply being worked around and not resolved at a fundamental level. The platform model changes the calculus of what can be done by decoupling the question of who is available for work from what work is available to do. Certain essential engineering or non repeatable engineering is obviously going to require specialization in a given domain, but major parts of systems as high as 80% are non-domain specific. In TopCoder’s experience 50 percent of a system can be downloaded from the generic catalog. To achieve better quality, the testing activities must be completed as rigorously as the development activities; leveraging the platform is a great enabler of this goal.

About the Author

Brendan Wright is a Director of Development at TopCoder (www.topcoder.com), a competitive software development community with over 200,000 developers representing more than 200 countries. His experience spans a wide range of software projects, influenced by a decade in various roles (software engineer, architect, consultant, project manager and more), with special focus on software development and related processes in the enterprise and security software marketplaces.

Author Contact Information

Email: bwright@topcoder.com.

\textsuperscript{11} In this context customer might mean the actual end user of the product, but it just as might mean some proxy, usually TopCoder has either a Platform Manager (a staffed consultant) helping clients interact with members, or sometimes a Copilot (a member fulfilling the same role), but sometimes customers handle it directly
The DACS Gold Practice Initiative:

- Promotes effective selection/use of software acquisition & development practices
- Defines essential activities/benefits of each practice
- Considers the environment in which each practice is used
- Addresses the timeliness of practice benefits
- Recognizes interrelationships between practices that influence success or failure
- Contains quantitative and qualitative information
- Includes discussion forums, case studies and survey opportunities
- A continually evolving resource for the DoD, Government, Industry and Academia
- Free to use/free to join

Current Gold Practices:

- Acquisition Process Improvement
- Architecture-First Approach
- Assess Reuse Risks and Costs
- Binary Quality Gates at the Inch-Pebble Level
- Commercial Specifications and Standards/Open Systems
- Ensure Interoperability
- Formal Inspections
- Formal Risk Management
- Goal-Question-Metric Approach
- Integrated Product and Process Development
- Model-Based Testing
- Plan for Technology Insertion
- Requirements Management
- Requirements Trade-Off/Negotiations
- Statistical Process Control
- Track Earned Value

Learn More About the DACS
Gold Practice Initiative:
http://www.goldpractices.com

Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

http://thedacs.com
Automated Software Testing Distilled – Standards and other Considerations as Prerequisites to Success

AST EFFORTS CAN FAIL WHEN SOFTWARE DEVELOPMENT DOESN’T TAKE INTO ACCOUNT THE AUTOMATED TESTING TECHNOLOGIES OR FRAMEWORK IN PLACE.

by Elfriede Dustin, Bernie Gauf, and Thom Garrett, Innovative Defense Technologies (IDT)

The primary challenge today for testing software is that customers want more software functionality to be delivered faster and cheaper, while at the same time they expect the quality of the software to at least meet if not exceed their expectations. Add to this challenge that software is becoming increasingly complex and given the current economic times and having to do more with less, the implications for the software test team are enormous. Increased complexity and functionality for example means each software baseline to be tested will be more time consuming, yet often more and increasingly complex test cases must be run for each delivery using the same resources. In other programs, more deliveries will be made each year, perhaps three or four product releases instead of one or two. More deliveries also mean more regression testing cycles. Simply stated, there is more software to test, with increasing complexity, more often, with fewer people. The current manual testing methods cannot keep pace. Automated Software Testing (AST) is a key technique that addresses some of the challenges software testers face today. At IDT we define automated software testing (AST) as encompassing all types of testing phases, including cross-platform compatibility and it is considered process-independent. Generally, all tests that are currently run as part of a manual testing program—functional, performance, concurrency, stress, security, and more—can be automated. Yet, just because something is automatable doesn’t mean it should be. In our book “Implementing Automated Software Testing”1 we provide a checklist that will help the reader determine whether a test should be automated. Our experience has shown that if implemented correctly, AST:

1. Can reduce the time and cost of software testing
2. Can improve software quality
3. Enhances manual testing efforts via increased testing coverage and replaces the manually mundane and labor intensive tasks
4. Does what manual testing can hardly accomplish, such as memory leak detection under specific conditions; concurrency testing, stress and performance testing, and more.

Also, see our article titled “The Business Case for Automated Testing”, published by DACS in the November 2007 issue of Software Tech News, for more detail on the benefits of AST.

Cost, schedule, and quality are parameters that are critical to the success of any software development product. Additionally, we have determined various areas that can contribute to the success of AST, such as:

• Software development considerations
• Automation standards across tool vendors
• Implementing the next generation of AST

Software Development Considerations for Automated Software Testing (AST)

AST efforts can fail when software development doesn’t take into account the automated testing technologies or framework in place. Developers who use the test driven development (TDD) approach consider their unit testing before they start developing, i.e. they develop a test before developing code. This same concept needs to be applied to automated software testing typically deployed during system and integration testing. Software developers can contribute to the success of automated testing efforts if they consider the impacts on automated testing efforts when making code or technology changes. Additionally, if developers consider some of the selected best practices described here, AST efforts can reap the benefits. The selected best practices include the following:

1. **Build testability into the application.**
2. **Adhere to Open Architecture standards.**
3. **Adhere to documentation standards** to include standard ways of documenting test cases and using

---

1 Book “Implementing Automated Software Testing,” Addison Wesley, March 2009, Dustin, Garrett, Gauf
the Object Management Group\textsuperscript{1} (OMG) IDL, for example, which would allow for automated test case code generation.

4. **Adhere to Coding Standards**: Facilitate automation tool recognition of objects: Uniquely name all objects, for example. Considering various platforms, i.e., client/server, Web, etc., and GUI/interface testing considerations, such as in the case of Windows development, for example, within the Windows architecture, uniquely name all objects. Additionally, don’t change the object names without AST considerations. Follow standard development practices; for example, maintain a consistent tab sequence.

Each of these recommendations is discussed in more detail in the next sections.

**Practice 1 - Build Testability into the Application**

Software developers can support the automated testing effort by building testability into their applications, which can be supported in various ways. One of the most common ways to increase the testability of an application is to provide a logging, or tracing, mechanism that provides information about what components are doing, including the data they are operating on, and any information about application state or errors that are encountered while the application is running. Test engineers can use this information to determine where errors are occurring in the system, or to track the processing flow during the execution of a test procedure.

As the application is executing, all components will write log entries detailing what methods, also known as functions, they are currently executing and the major objects they are dealing with. The entries are written typically to a disk file or database, properly formatted for analysis or debugging, which will occur at some point in the future, after the execution of one or more test procedures. In a complex client/server or Web system, log files may be written on several machines, so it is important that the log include enough information to determine the path of execution between machines.

It is important to place enough information into the log that it will be useful for analysis and debugging, but not so much information that the overwhelming volume will make it difficult to isolate important entries. A log entry is simply a formatted message that contains key information that can be used during analysis. A well-formed log entry includes the following example pieces of information:

- **Class name and method name**. This can also simply be a function name if the function is not a member of any class. This is important for determining a path of execution through several components.

- **Host name and process ID**. This will allow log entries to be compared and tracked if they happen on different machines or in different processes on the same machine.

- **Timestamp of the entry (to the millisecond at least)**. An accurate timestamp on all entries will allow the events to be lined up if they occur in parallel or on different machines.

- **Messages**. One of the most important pieces of the entry is the message. It is a description, written by the developer, of what is currently happening in the application. A message can also be an error encountered during execution, or a result code from an operation. Gray-box testing will greatly benefit from the logging of persistent entity IDs or keys of major domain objects. This will allow objects to be tracked through the system during execution of a test procedure.

Having these items written to the log file by every method, or function, of every component in the system can realize the following benefits:

- The execution of a test procedure can be traced through the system and lined up with the data in the database that it is operating on.

- In the case of a serious failure, the log records will indicate the responsible component.

- In the case of a computational error, the log file will contain all of the components that participated in the execution of the test procedure and the IDs or keys of all entities used.

- Along with the entity data from the database, this should be enough information for the test team to pass on to the development personnel who can isolate the error in the source code.

Following is an example of a log file from an application that is retrieving a customer object from a database\textsuperscript{ii}:

<table>
<thead>
<tr>
<th>Function: main (main.cpp, line 100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine: testsrvr (PID=2201)</td>
</tr>
<tr>
<td>Timestamp: 5/10/2009 20:26:54</td>
</tr>
</tbody>
</table>
| Message: connecting to database [dbserver1,
customer_db]
Function: main (main.cpp, line 125)
Machine: testsrvr (PID=2201)
Timestamp: 5/10/2009 20:26:56
Message: successfully connected to database [dbserver1, customer_db]

Function: retrieveCustomer (customer.cpp line 20)
Machine: testsrvr (PID=2201)
Timestamp: 5/10/2009 20:26:56
Message: attempting to retrieve customer record for customer ID [A1000723]

Function: retrieveCustomer (customer.cpp line 25)
Machine: testsrvr (PID=2201)
Timestamp: 5/10/2009 20:26:57
Message: ERROR: failed to retrieve customer record, message [customer record for ID A1000723 not found]

This log file excerpt demonstrates a few of the major points of application logging that can be used for effective testing.

• In each entry, the function name is indicated, along with the filename and line number in the code where the entry was written. The host and process ID are also recorded, as well as the time that the entry was written.

• Each message contains some useful information about the activity being performed; for example, the database server is dbserver1, the database is customer_db, and the customer ID is A1000723.

• From this log, it is evident that the application was not able to successfully retrieve the specified customer record.

In this situation, a tester could examine the database on dbserver1, using SQL tools, and query the customer_db database for the customer record with ID A1000723 to verify its presence. This information adds a substantial amount of defect diagnosis capability to the testing effort, since the tester can now pass this information along to the development staff as part of the defect information. The tester is now not only reporting a “symptom,” but along with the symptom can document the internal application behavior that pinpoints the cause of the problem.

Practice 2 - Adhere to Open Architecture Standards

Open Architecture (OA) principles, described in the Open Architecture Computing Environment Design Guidance and in the Open Architecture Computing Environment Technologies and Standards documents, were developed by the U.S. Navy and emphasize the use of widely adopted industry standards and component-based technologies. The open standards approach has been demonstrated to reduce cost and improve rapid insertion of new software capability into an existing system.

By implementing and following the OA standards, developers can expect various benefits including assured technical performance, reduced lifecycle cost, affordable technology refresh, and reduced upgrade cycle time. Additional expected benefits include:

• Scalable, load-invariant performance
• Enhanced information access and interoperability
• Enhanced system flexibility for accomplishment of mission and operational objectives
• Enhanced survivability and availability
• Reduced lifecycle cost and affordable technology refresh
• Reduced cycle time for changes and upgrades

The Defense Advanced Research Projects Agency (DARPA), academia, and industry’s R&D efforts have focused on certain architectural concepts intended to foster lifecycle cost benefits, as well as technical performance benefits. Developing software using OA will result in additional benefits, such as:

• Open architectures
• Distributed processing
• Portability
• Scalability
• Modularity
• Fault tolerance
• Shared resource management
• Self-instrumentation

For additional details and best development practices specific to automated testing tool development, see Table 8.5, “Test Development Guidelines” in the book Automated Software Testing.
Practice 3 - Adhere to Standard Documentation Format

Software testing efforts often include sifting through documentation to verify that all information is provided. Documentation assessment efforts can also be automated, but we would like to offer the following recommendation for software developers or documentation teams in order to support their successful automation. Currently almost all software providers and vendors use various documentation formats to produce documentation deliverables; no one specific format is being followed. We recommend that a standard documentation format be used, i.e., templates that offer multiple-choice offerings, standard notations, and naming conventions.

Adherence to standard templates, using a finite set of allowable keywords, will make the automation of documentation assessment a straightforward exercise. It is recommended to develop documentation templates that follow OMG documentation standards, for example, or any other type of standard that the customer would like its developers to adhere to (ISO, IEEE, and so forth).

Document Test Cases in a Standard Way

Much time is spent on test case documentation. While some sort of test case documentation is always desired, this process can be automated partially, if automated test case generation from use cases or models, for example, is allowed. Much research has gone into various technologies that allow test case generation from models, etc., and a standard way of documenting test cases is the goal. Various efforts are under way to develop standards, such as the MOF to Text standard (the Web site is www.omg.org/cgi-bin/doc?formal/08-01-16.pdf) and IBM/Telelogic’s Rhapsody, which provides an automated test case generator (called ATG)) to produce unit test cases.

ATG is developed using a formal methodology to decompose requirements written in a natural language in order to produce a set of unambiguous rules, object relationships, states, and so on that define the rulesbehavior described by the requirements document(s). The rules and relationships are captured using a formal language.

The “formal” language description then becomes the blueprint to generate and identify rule dependencies (actions/reactions), which form “threads” of potential sequences.

These dependency threads are the basis for test case development (to develop required data sets, system configuration, and event triggers/and system stimulation). ATG has a test driver component that is used to stimulate system sensor and communication interfaces in a controlled or ad hoc environment and monitor responses.

The concept of ATG can be applied to virtually any rule set. At IDT, our work has focused much on developing test cases and test capability to assess tactical data link standards, as these standards use a lot of automated processes that can be readily verified using automated test case documentation and test case generation.

We have developed a to-be-patented technology that allows for test case generation using GUI stimulation. This is another way to automate the development of standard test case documentation.

Practice 4 - Adhere to Coding Standards

Software generally needs to be cross-platform-compatible and developed in a standardized fashion, in order to allow for maintainability and portability and to be most efficient. For best AST support, it is important that customer developers adhere to coding standards such as those defined in the OMG C++ Language Mapping Specification, version 1.1, June 2003, or the OMG C Language Mapping Specification, June 1999 and other standards, such as ISO/ANSI C++, and so forth.

Use of OMG’s Interface Description Language (IDL)

Another example of using standardized documentation is to use the OMG IDL to help define the interfaces. The OMG IDL is also an ISO International Standard, number 14750. The use of IDL allows for automatic code generation, turning a time-consuming and error-prone manual task into an efficient automated task, saving valuable time. This is another concept as part of IDT’s recommended “Automating the Automation (AA)” practice.

Graphical User Interface (GUI) Testing Recommendations

Numerous user interface standards and guidelines exist for developers to adhere to when developing their Applications

Continued on pg 30
The RIAC/DACS System Reliability Toolkit provides technical guidance in all aspects of system reliability, allowing the user to understand and implement techniques to ensure that system and product designs exhibit satisfactory hardware, software and human reliability, and to minimize the inherent risks associated with deficiencies in system reliability.

To purchase, please contact:
The Reliability Information Analysis Center
6000 Flanagan Road
Suite 3
Utica, NY 13502-1348

1-877-363-RIAC http://theriac.org

This first edition of the DACS Software Reliability Sourcebook provides a concise resource for information about the practical application of software reliability technology and techniques. The Sourcebook is divided into nine major sections, plus seven supporting appendices.

To purchase, please contact:
The Data & Analysis Center for Software
775 Daedalian Drive
Rome, NY 13441-4909

1-800-214-7921 http://thedacs.com
Under Test’s (AUT) GUI. However, their usability is limited because they are numerous, i.e. different guidelines depending on technology, and or/and conflicts within and among standards documents, redundancy between documents, and a lack of specific guidance. There are also many GUI Builders available, such as Netbeans, Eclipse GUI builder, and so forth, each allowing the developers various opportunities to create a GUI, none really providing specific standards to allow for ease of AST.

A great idea for effective GUI generation that can support AST efforts is provided by IBM’s Reflexive User Interface Builder (RIB), which builds Java GUIs simply and quickly with new technology from alphaWorks and it provides the following features:

- RIB specifies a flexible and easy-to-use XML markup language for describing Java GUIs and provides an engine for creating them. You can define color scheme, font, icon usage, menu/functionality placement, etc.
- You can use RIB to test and evaluate basic GUI layout and functionality, or to create and render GUIs for an application. This concept for AST of a GUI application is ideal.

For effective GUI AST a tool exists that allows for GUIs to be generated in a standard way as described in the RIB concept.

Some tools use the Virtual Network Computing (VNC) technology for AST. Capture/playback tools record the test engineer’s keystrokes in some type of scripting language and allow for script playback for baseline verification. Often there is a requirement specifying that the capture/playback tool cannot reside on the AUT, and remote technology is required to access the AUT; in this case VNC technology is applied. In the case of the automated tool using VNC technology, the following recommendations should be considered, because these types of capture/playback tools are sensitive to these changes:

- Control “font smoothing” and other text characteristics should not be changed
- Don’t change the color depth of the AUT
- Display settings need to stay the same (resolution, etc.)
- If possible, keep the default settings in the OS; use standard visual settings
- Many more practices can be applied to support standard GUI generation and to make GUI testing more effective, such as when the tester clicks on a GUI control, customized code modules get generated, that allow for quick and consistent code generation, removing the manual code entry/development (this is another concept as part of our “Automating the Automation (AA)” practice.

Additionally, GUI Object Naming Standards are recommended, and are discussed next.

### GUI Object Naming Standards

Many automation tools key on the names of objects. Not only does this facilitate development of automation test programs, but it also encourages good software development practices. Microsoft, for example, promotes naming standards at [http://support.microsoft.com/](http://support.microsoft.com) kb/110264.

Failure of application developers to name objects consistently is certain to delay test automation programming.

### Automation Standards across Tool Vendors

Numerous vendors provide AST tools, and various open-source testing tools are available, but a lack of automation standards persists.

Many different types of standards have the potential to affect AST. Improved ROI from automated testing can be realized through standards applied to the component(s) under test, the test tools and harness, and other aspects of the test environment. Key considerations in determining the types of standards of greatest interest include the degree to which standards of that type support the following characteristics:

- Ease of automation—reduction of the time and complexity needed for automation, thereby reducing the initial investment or increasing the degree of automation that can be achieved
- Plug and play—increased reuse of automated test patterns across products, allowing for reusability of various automation components given the same test scenario
- Product availability—increased selection of products supporting automated testing, including test tools as well as other related capabilities such as products to support monitoring and control of the application during testing
- Product interchangeability—reduction of vendor lock-in, enabling developers to choose different automation tools for different parts of the testing process or for different baselines, while leveraging prior automation efforts
• Product interoperability—ability to use multiple products within a single test set, enabling developers to leverage the capabilities provided by multiple products resulting in a more robust and higher-quality test

• Cross-platform compatibility—ability of one or more tools to be cross-platform-compatible across various OSs and technologies

• Testing capability—improved robustness and thoroughness of automated testing, resulting in higher-quality tests

Sample Automated Test Tool Standards

Automated test tools provide many opportunities for standardization. Currently, most automated test standards address hardware testing. However, significant benefits could be gained by standardizing various aspects of software test tools.

• Scripting language. Automated test tools use a scripting language to control the events of the tests. Each test tool comes with its own scripting language. Although some of these languages conform to open standards, others are proprietary. A standard for a common scripting language would improve product interchangeability. A set of scripts that works with one automated testing tool could potentially be used with others. For example, currently, even if tool A uses the same scripting language as tool B, the scripts are not interchangeable, because tool A and tool B have different recording mechanisms.

• Capture feature. Many automated testing tools come with a capture feature, where the tester’s keystrokes are recorded as s/he is executing a test. All automated testing tools have different and unique recording features. None of the features are standardized, thus producing different script outputs for the same testing steps.

• Test data. Many automated testing tools provide a way to generate test data. Some provide a test data database, others provide a flat file, but rarely do they provide reuse across tools. A standardized way of generating test data would be useful.

• Modularity. Currently the various automated testing tools provide different features to allow for test script componentizing and modularity; i.e., a subscript can be called from a superscript. Providing a modularity feature that’s standardized across tools would allow for reusability of modular and componentized scripts.

• APIs. Many automated tests require a test harness; i.e., the test harness receives inputs from the test tool via an application program interface (API) that is specific to the tool. It converts the inputs into API calls that are specific to the application being tested, effectively serving as a bridge between the test tool and the target application. The harness is also responsible for collecting the results and providing them back to the test tool. The API between the test tool and the test harness is not currently standardized. Standardization of this API would be another significant step toward enabling interchangeability of automated test tools. The consistency provided by common scripting languages and APIs would also provide greater ease of use and would reduce the learning curve associated with each new test tool.

• Test tool output and reporting. Many automated testing tools currently use various methods to produce test results output and reporting. No standard way of test-run output and reporting exists.

• Integration with other testing support tools. Many automated testing tools provide APIs that allow imports and/or exports to other test support tools, such as requirements management, configuration management, and defect tracking tools. However, not all tools provide this standard capability.

In addition to the standards described here, other areas can benefit from standardization to support the automation cause, such as systems engineering standards to be applied during software development and so forth in support of AST. Currently an effort is under way at OMG, which IDT helped initiate, to standardize some of the AST efforts, called Automated Test and Re-test (ATRT).

Implementing the next generation of AST

Although AST is most often an improvement over manual software testing efforts, software development is required to test developed software. Testers rely on effective software development that considers AST and they develop software to test software because currently no better alternative exists. Efforts are under way, such as IBM’s autonomic computing initiative, that could possibly provide for self-testable components in the future, but while some progress has been made, at the writing of this book only a few companies have implemented and applied the autonomic computing concepts.
Automated Software Testing Distilled - Standards and Other Considerations as Prerequisites to Success (Cont.)

Ultimately, the AST goal is to develop a mechanism that will allow the user to not only determine failure information or detect software troubles and defects, but additionally provide the user with a list of recommended actions to solve the failure. In a software failure situation, AST efforts then not only detect a defect but is then able to programmatical support trouble-shooting/pointing the issue; code is setup to be able to “self-heal” or “self-repair” by allowing the user to programmatically select from a list of updated modules or list of actions that are recommended to fix the issue. For example, if a module fails – immediately the AST error detection mechanism/error message would point to the module that failed and why it failed, i.e. AST would have knowledge and be able to determine the various ways this module could fail, knowing the intricacies of the various module dependencies, data flows, path flows and so forth. Based on the various possible failures AST would provide self-healing/self-fixing modules and recommendations to solve the issue. This is IDT’s plan for the next generation of AST.

Given the current AST state and the need for developing software to test software, effective automated testing includes reducing human involvement and automating as much of what is involved in the effort as possible. The effort of developing software to test software needs to be efficiently streamlined. Using automation to implement software test automation is an effective way of removing required and error-prone manual human interaction. The goal as part of the automated testing streamlining is to automate the test automation.

For example, companies often spend time developing a test framework from scratch, creating functions to allow for batch testing, distributed testing or e-mail notification; although numerous frameworks already exist that provide such features. Taking advantage of freely available open-source components or free/shareware that provides the required features allows for reuse and ease of additional feature integration, saving time and money. Instead of developing new testing framework features we find that often much of what we need has already been developed by the open-source community and is ready to be downloaded. We recommend that any automation test effort consider open-source components for designing and developing test automation frameworks. When considering open-source tools, keep the validity of the open-source license in mind. Some approved open-source licenses can be found and verified at http://opensource.org/licenses.

Test automators know that automated testing requires its own development lifecycle, so let’s borrow from what the software development community has already figured out: using tools that help automate all phases of the lifecycle, from requirements to design, to implementation, testing, and defect tracking. As part of your test framework, for test procedure development you could follow a model-driven design. Telelogic/IBM has a promising vendor-provided solution, with its Automated Test Generator (ATG) and Test Conductor (TC). Together ATG and TC generate a “test harness/platform” that can be used to generate test cases and the test code needed to completely test your model on the development platform as well as the target. An alternative to a vendor-provided solution would be the open-source AndroMDA; “you can now create and use your own Eclipse-Modeling-Framework (EMF)-based metamodels and write model-to-model transformations to transform from a PIM [platform-independent model] (at the highest level of abstraction) down through a stack of intermediate PSMs [platform-specific models] (using easily understandable transformation steps) until you finally reach the lowest level PSM from which it is easy to generate code using simple model-to-text transformations.”

Testers are often faced with the challenge of verifying a huge set of test data combinations and permutations, and it is tedious and error-prone to manually develop such a data set using an Excel spreadsheet. But a manual test data creation process is still common in many testing shops. Why not use one of the many test data generators that provide us with automated combinatorial-based data output? For example, the National Institute for Standards and Technology (NIST) has developed a useful tool called FireEye (or ACTS), which can be downloaded at http://csrc.nist.gov/groups/SNS/acts/download/ (contact NIST for a login ID and password), which allows for such automated test data generation. NIST compares its test data generator tool FireEye to other available tools at http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html.

There are many more ways to automate test automation: using existing Java libraries to test a user interface, such as Jemmy, or using an interface design language (IDL) to allow for code generation, or using built-in probes (see www.ocsystems.com), or developing self-testable components. As long as we are required to use/develop software to test software, we may as well automate that process as much as possible. Some additional examples of ways to streamline test automation are presented on the following page:
• Automated test case code generation

<table>
<thead>
<tr>
<th>Software interface test code</th>
<th>Test case procedure code</th>
</tr>
</thead>
<tbody>
<tr>
<td>• A software interface could be any type of middleware, such as CORBA, DDS, or a simple Multicast layout.</td>
<td>• Each step in the test case procedure has associated code generated to execute its behavior.</td>
</tr>
<tr>
<td>• You can then generate software interface code from various input formats, such as IDLs, C-style headers, and so forth.</td>
<td>• A standardized test case format facilitates automated step procedure extraction</td>
</tr>
<tr>
<td>• You would then generate code that glues the interface code to the test framework.</td>
<td>• XML could be used to define the extracted test step information as input into the autogeneration process.</td>
</tr>
</tbody>
</table>

• Tools to be used for this “Automating the Automation” example

| • ANTLR | • (www.antlr.org/) |
| • | • Works off of grammar files. Several common languages’ grammar files are available for download (such as IDL and C). |
| • | • Generates a lexical analyzer and parser functionality in Java. (Other languages are supported as well.) |
| • StringTemplate | • (www.stringtemplate.org/) |
| • | • Uses ANTLR. Developed by the same team. |
| • | • Provides pluggable templates. Code can be generated in multiple languages without having to modify the autogen code. |
| • JAXB | • (https://jaxb.dev.java.net/) |
| • | • XML to Java object binding based on XML schema. |
| • | • Generates Java classes used to interpret XML data. |

See our book “Implementing Automated Software Testing” Appendix D for a case study discussion of how we used some of these automated testing technologies to implement our own framework, using mostly open-source tools. Also, see our presentation at the Google Test Automation Conference (GTAC 2008) for more details on how to implement the automated code generation.

Additionally, innovative developers are turning to composite applications for the agility needed to get ahead of rapidly changing business conditions. The concept of checking out software components within a library setting in order to support software component reuse is notable. Being able to capitalize on existing components and assemble these into custom-built components allows for a new look and feel, and new user interfaces and transactions, and behind-the-scenes composition of resources into new patterns of applications, allow for quicker turnaround of a new release. We support this “library” and “software reuse” concept, because existing automated software tests can be reused on the various components. For example, an AST process will allow for reuse of those automated testing efforts for each component in a shortened time frame, supporting the required efficiencies of the “library” software component checkout and reuse, by being able to reuse the equivalent or applicable “automation component.” This in turn will help streamline automated testing efforts.

**In summary,** much consideration needs to be taken into account to support AST success. Development needs to keep AST in the forefront during their software development activities and it can’t be an afterthought. Various standards can make it more successful and the next generation of AST, while R&D is taking place, still needs to be developed and advanced. While effective AST improves software testing efforts, much still can be done to make it even more effective. If the listed areas, for example, are taken into consideration chances for AST success will be enhanced. IDT continues working on this Next Generation of Automated Software Testing (AST NG).

---

**Endnotes**

i www.omg.com/.


iv Dustin et al., *Automated Software Testing.*
Recent Technical Reports from DACS

- **Software Project Management for Software Assurance: A DACS State of the Art Report**
  Released: 9/30/2007

- **Enhancing the Development Life Cycle to Produce Secure Software**
  Released: Version 2.0, October 2008
  https://www.thedacs.com/techs/enhanced_life_cycles/

  Released: August 2007

  Released: 9/30/2007

PDF versions of these reports can be downloaded free of charge by registered DACS visitors who are logged in.

Hard copies can be ordered (for fee) from the DACS store online at [https://store.thedacs.com/](https://store.thedacs.com/)

---

**About the Authors**

Elfriede Dustin, Thom Garrett, and Bernie Gauf work together at Innovative Defense Technologies (www.idtus.com), which specializes in the design, development, and implementation of automated software testing solutions.

**Elfriede Dustin** has authored multiple software testing books and articles based on her many years of actual hands-on automated software testing experience. Elfriede leads IDT’s efforts in automated software testing research programs. She can be contacted at edustin[AT]idtus.com

**Thom Garrett** has experience in planning, testing, and deployment of complex systems for DoD and commercial applications for companies such as Public Broadcasting Service (PBS), Digital System Resources (DSR), Inc., and America Online (AOL). Thom received a master’s degree from the University of San Francisco. He can be contacted at tgarrett[AT]idtus.com

**Bernie Gauf** is the president of IDT. Bernie has been invited to participate in numerous DoD panels associated with the use of new technology, testing conferences, and as a guest speaker to share his insights on automated software testing. He can be contacted at bgauf[AT]idtus.com
Are you getting All you can from your software investment?

The DACS ROI Dashboard

Impact on Quality (% defect reduction)

75th Percentile: 90

Oklahoma City Air Logistics Center, Test Program Set and Industrial Automation line observed the defect density decrease 91%, from 3.3 to 0.3 Defects per KSLOC, while achieving CMM Level 4.

Access the DACS ROI Dashboard!
http://www.thedacs.com/databases/roi/

Technologies Covered:
- SEI CMM/CMMI
- SEI Team Software Process (TSP)
- SEI Personal Software Process (PSP)
- Inspections
- Reuse
- Cleanroom
And Many More!

Graphs Showing Impact Of Software Technologies On:
- ROI
- Productivity
- Quality

Summarizes Facts From Open Literature

Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

http://thedacs.com
Predicting Test Cases and Test Team Size

BECAUSE MANY INDIVIDUAL TEST STAGES SUCH AS UNIT TEST ARE SO LOW IN EFFICIENCY, IT CAN BE SEEN WHY SEVERAL DIFFERENT KINDS OF TESTING ARE NEEDED TO ACHIEVE USEFUL LEVELS OF SOFTWARE DEFECT REMOVAL EFFICIENCY.

by Capers Jones, President, Capers Jones & Associates LLC

Author’s Note: This article is an excerpt from chapter 5 of “Software Engineering Best Practices” to be published by McGraw Hill later in 2009.

There is no standard way of testing software applications in 2009. Not only is there no standard way of testing, but there are no standard measures of test coverage or defect removal efficiency, although both are technically straightforward measurements.

The most widely used form of test measurement is that of test coverage, which shows the amount of code actually executed by test cases. Test coverage measures are fully automated and therefore easy to do. This is a useful metric, but much more useful would be to measure defect removal efficiency as well.

Defect removal efficiency is more complicated and not fully automated. To measure the defect removal efficiency of a specific test stage such as “unit test”, all defects found by the test are recorded. After unit test is finished, all other defects found by all other tests are recorded, as are defects found by customers in the first 90 days. When all defects have been totaled, then removal efficiency can be calculated.

Defect removal efficiency originated in IBM in the early 1970’s as a method for evaluating testing effectiveness. Because this metric provided information that could be used for inspections and static analysis, as well as testing, it has become a general way of measuring the relative effectiveness of all forms of defect removal activity. Recent findings by companies such as IBM, Coverity, and Dovél Technologies show removal efficiency levels that often top 96%.

A software “defect” is defined as a bug or error that causes software to either stop operating or to produce invalid or unacceptable results. Using IBM’s severity scale, defects have four levels of severity:

Severity 1 means that the software application does not work at all.
Severity 2 means that major functions are disabled or produce incorrect results.
Severity 3 means that there are minor issues or minor functions are not working.
Severity 4 means a cosmetic problem that does not affect operation.

There is some subjectivity with these defect severity levels because they are assigned by human beings. Under the IBM model the initial severity level is assigned when the bug is first reported based on symptoms described by the customer or user who reported the defect. However a final severity level is assigned by the change team when the defect is repaired.

This definition of quality is one favored by the author for several reasons. First, defects can be predicted before they occur and measured when they do occur. Second, customer satisfaction surveys for many software applications appear to correlate more closely to delivered defect levels than to any other factor.

Assume unit test found 100 defects, function test and later test stages found 200 defects, and customers reported 100 defects in the first 90 days of use. The total number of defects found was 400. Since unit test found 100 out of 400 defects, in this example its efficiency is 25%, which is actually not far from the 30% average value of defect removal efficiency for unit test.

It is an unfortunate fact that most forms of testing are not very efficient, and find only about 25% to 40% of the bugs that are actually present, although the range is from less than 20% to more than 70%.

Because many individual test stages such as unit test are so low in efficiency, it can be seen why several different kinds of testing are needed to achieve useful levels of software defect removal efficiency. The term “cumulative defect removal efficiency” refers to the overall efficiency of an entire sequence of tests or defect removal operations.

Companies that depend purely upon testing for defect removal almost never top 90% in cumulative defect removal, and often are below 75%. These observations are derived from quality benchmark studies carried out by the author.
and his colleagues between 1984 and 2009 in several hundred companies in the United States, Europe, and Japan.

The newer forms of testing, such as “test-driven development” (TDD), use test cases as a form of specification and create the test cases first, before the code itself is created. As a result, the defect removal efficiency of TDD is higher than many forms of testing and can top 85%. However, even with TDD a phenomenon called “bad-fix injection” needs to be factored in to the equation. About 7% of attempts to fix bugs accidentally include new bugs in the fixes themselves.

If TDD is combined with other approaches such as formal inspection of the test cases and static analysis of the code then defect removal efficiency can top 95%.

The poor defect removal efficiency of normal testing brings up an important question: “If testing is not very effective in finding and removing bugs, what is effective?” The answer to the question is that a combination of defect prevention and multiple forms of defect removal are needed for optimum effectiveness.

The phrase “defect prevention” refers to methods and techniques that can lower defect potentials from U.S. averages of about 5.0 per function point. Examples of methods that have demonstrated effectiveness in terms of defect prevention include the higher levels of the capability maturity model (CMMI), joint application design (JAD), quality function deployment (QFD), root-cause analysis, six-sigma for software, the team software process (TSP) and also the personal software process (PSP).

In terms of defect removal, pre-test inspections are valuable. Using automated static analysis is also valuable, with the caveat that only about 15 programming languages out of more than 700 are currently supported by static analysis tools.

As it happens, formal inspections of requirements, design, and code serve double duty, and are very effective in terms of defect prevention as well as being very effective in terms of defect removal. This is because participants in formal inspections spontaneously avoid making the same mistakes that are found during the inspections.

In a private communication with the author, Watts Humphrey noted that after a year of inspections at IBM there was some executive pressure to stop doing inspections on the grounds that so few defects were being found that inspections were no longer cost effective. However the IBM labs that did stop doing inspections soon found that defect potentials were rising again.

The combination of methods that has been demonstrated to raise defect removal efficiency levels include formal inspections of requirements, design, code, and test materials; static analysis of code prior to testing, and then a test sequence that includes at least eight forms of testing: 1) unit test, 2) new function test, 3) regression test, 4) performance test, 5) security test, 6) usability test, 7) system test, and 8) some form of external test with customers or clients, such as Beta test or acceptance test.

Such a combination of pre-test inspections, static analysis, and at least eight discrete test stages will usually approach 99% in terms of cumulative defect removal efficiency levels. Not only does this combination raise defect removal efficiency levels, but it is also very cost effective.

Projects that top 95% in defect removal efficiency levels usually have shorter development schedules and lower costs that projects that skimp on quality. And, of course, they have much lower maintenance and customer support costs too.

Observations and depositions from litigation involving software projects that either failed or greatly exceeded their planned schedules and costs revealed that the projects had no overt symptoms of trouble until testing began. When testing did being, unexpectedly large numbers of bugs stretched out the testing cycle and kept the projects from going forward as planned.

Testing is a teachable skill, and there are a number of for-profit and non-profit organizations that offers seminars, classes, and several flavors of certification for test personnel. While there is some evidence that certified test personnel do end up with higher levels of defect removal efficiency than uncertified test personnel, the poor measurement and benchmark practices of the software industry make that claim somewhat anecdotal. It would be helpful if test certification included a learning segment on how to measure defect removal efficiency.

At this point it is useful to address three topics that are not well covered in the testing literature:

1. How many testers are needed for various kinds of testing?
2. How many test cases are needed for various kinds of testing?
3. What is the defect removal efficiency of various kinds of testing?

Table 1.0 shows the approximate staffing levels for 17 forms of testing that are common in the United States. Note that this
information is only approximate, and there are wide ranges for each form of testing.

Because testing executes source code, the information in table 1.0 is based on source code counts rather than on function points. With more than 700 programming languages ranging from assembly through modern languages such as Ruby and E, the same application illustrated in table 1.0 might vary by more than 500% in terms of source code size. Java is the language used in table 1.0 because it is one of the most common languages in 2009.

The column labeled “assignment scope” illustrates the amount of source code that one tester will probably be responsible for testing. Note that there are very wide ranges in assignment scopes based on the experience levels of test personnel, on the cyclomatic complexity of the code, and to a certain extent on the specific language or combination of languages in the application being tested.

Additional data on assignment scope is available from a number of companies that perform benchmarks. The non-profit International Software Benchmark Standards Group (ISBSG) is one source. The Japanese government also collects such benchmark data as do for-profit consulting organizations such as Software Productivity Research (SPR), the David Consulting Group, and Galorath Associates.

Because the testing shown in table 1.0 involves a number of different people with different skills who probably would be from different departments, the staffing breakdown for all 17 tests would include 5 developers through unit test; 2 test specialists for integration and system test; 3 specialists for security, nationalization, and usability test; 1 SQA specialist; 7 outside specialists from other companies; and 2 customers: 20 different people in all.

Of course it is unlikely that any small application of 1000 function points or 50 KLOC would use (or need) all 17 of these forms of testing. The most probable sequence for a 50 KLOC Java application would be 6 kinds of testing performed by 5 developers, 2 test specialists; and 2 users; for a total of 9 test personnel in all.

In table 2.0 (following page), the purpose is to show the approximate numbers of test cases produced for each test stage, and then the total number of test cases for the entire application. Here too there are major variations so the data is only approximate.

The code defect potential for the 50 KLOC code sample of the Java application would be about 1,500 total bugs which is equal to 1.5 code bugs per function point or 30 bugs per KLOC. (Note that earlier bugs in requirements and design are excluded and assumed to have been removed before testing begins.)

If all 17 of the test stages were used, they would probably detect about 95% of the total bugs present, or 1,425 in all. That would leave 75 bugs latent when the application is delivered. Assuming both the numbers for potential defects and the numbers for test cases are reasonably accurate (a questionable assumption) then it takes an average of 1.98 test cases to find 1 bug.

Of course since only about 6 out of the 17 test stages are

<table>
<thead>
<tr>
<th>Table 1.0: Test Staffing for Selected Test Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application language</td>
</tr>
<tr>
<td>Application code size</td>
</tr>
<tr>
<td>Function points</td>
</tr>
<tr>
<td>General Testing</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>Special Testing</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>External Testing</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
</tbody>
</table>
usually performed, the removal efficiency would probably be closer to 75%, which is why additional non-test methods such as inspections and static analysis are needed to achieve really high levels of defect removal efficiency.

If even this small 50 KLOC example uses more than 2,800 test cases, it is obvious that corporations with hundreds of software applications will eventually end up with millions of test cases. Once created, test cases have residual value for regression test purposes. Fortunately there are a number of automated tools that can be used to store and manage test case libraries.

The existence of such large test libraries is a necessary overhead of software development and maintenance. However this topic needs additional study. Creating reusable test cases would seem to be of value. Also, there are often errors in test cases themselves, which is why inspections of test plans and test cases are useful.

With hundreds of different people creating test cases in large companies and government agencies, there is a good chance that duplicate tests will accidentally be created. In fact this does occur, and a study at IBM noted about 30% redundancy or duplicates in one software lab’s test library.

The final table 3.0, (following page), in this article shows defect removal efficiency levels against six sources of error: requirements defects, design defects, coding defects, security defects, defects in test cases, and performance defects.

Table 3.0 is complicated by the fact that not every defect removal method is equally effective against each type of defect. In fact many forms of defect removal have 0% efficiency against security flaws. Coding defects are the easiest type of defect to remove; requirements defects, security defects, and defects in test materials are the most difficult to eliminate.

Historically formal inspections have the highest levels of defect removal efficiency against the broadest range of defects. The more recent method of static analysis has a commendably high level of defect removal efficiency against coding defects, but currently operates only on about 15 to perhaps 30 programming languages out of more than 700. Also, static analysis does not detect requirements errors.

The data in table 3.0 has a high margin of error, but shows the kind of data that needs to be collected in much greater volume to improve software quality and raise overall levels of defect removal efficiency across the software industry. In fact every software application larger than 1000 function points...
## Table 3.0: Defect Removal Efficiency by Defect Type

<table>
<thead>
<tr>
<th>Defect types →</th>
<th>Requirements</th>
<th>Design</th>
<th>Code</th>
<th>Security</th>
<th>Test</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Test Removal</td>
<td><strong>Inspections:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Requirements</td>
<td>85.00%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Design</td>
<td>85.00%</td>
<td></td>
<td>25.00%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 Code</td>
<td>85.00%</td>
<td>40.00%</td>
<td></td>
<td>15.00%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 Test plans</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>85.00%</td>
</tr>
<tr>
<td>5 Test cases</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>85.00%</td>
</tr>
<tr>
<td>6 Static analysis</td>
<td></td>
<td>30.00%</td>
<td>87.00%</td>
<td>25.00%</td>
<td></td>
<td>20.00%</td>
</tr>
<tr>
<td>General Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 Subroutine test</td>
<td></td>
<td></td>
<td>35.00%</td>
<td></td>
<td></td>
<td>10.00%</td>
</tr>
<tr>
<td>8 Unit test</td>
<td></td>
<td></td>
<td>30.00%</td>
<td></td>
<td></td>
<td>10.00%</td>
</tr>
<tr>
<td>9 New function test</td>
<td></td>
<td>15.00%</td>
<td>35.00%</td>
<td></td>
<td></td>
<td>10.00%</td>
</tr>
<tr>
<td>10 Regression test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15.00%</td>
</tr>
<tr>
<td>11 System test</td>
<td>10.00%</td>
<td>20.00%</td>
<td>25.00%</td>
<td>7.00%</td>
<td></td>
<td>25.00%</td>
</tr>
<tr>
<td>Special Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 Performance testing</td>
<td></td>
<td>5.00%</td>
<td>10.00%</td>
<td></td>
<td></td>
<td>70.00%</td>
</tr>
<tr>
<td>13 Security testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>65.00%</td>
</tr>
<tr>
<td>14 Usability testing</td>
<td></td>
<td>10.00%</td>
<td>10.00%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 Component testing</td>
<td></td>
<td>10.00%</td>
<td>25.00%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 Integration testing</td>
<td></td>
<td>10.00%</td>
<td>30.00%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 Nationalization testing</td>
<td></td>
<td>3.00%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18 Platform testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10.00%</td>
</tr>
<tr>
<td>19 SQA validation testing</td>
<td></td>
<td>5.00%</td>
<td>5.00%</td>
<td>15.00%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 Lab testing</td>
<td>10.00%</td>
<td>10.00%</td>
<td>10.00%</td>
<td></td>
<td></td>
<td>20.00%</td>
</tr>
<tr>
<td>External Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21 Independent testing</td>
<td></td>
<td>5.00%</td>
<td>30.00%</td>
<td>5.00%</td>
<td>5.00%</td>
<td>10.00%</td>
</tr>
<tr>
<td>22 Beta testing</td>
<td></td>
<td>30.00%</td>
<td>25.00%</td>
<td>10.00%</td>
<td></td>
<td>15.00%</td>
</tr>
<tr>
<td>23 Acceptance testing</td>
<td></td>
<td>30.00%</td>
<td>20.00%</td>
<td>5.00%</td>
<td></td>
<td>15.00%</td>
</tr>
<tr>
<td>Special Activities</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24 Audits</td>
<td></td>
<td>15.00%</td>
<td>10.00%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25 Independent Verification and Validation (IV&amp;V)</td>
<td></td>
<td>10.00%</td>
<td>10.00%</td>
<td></td>
<td></td>
<td>10.00%</td>
</tr>
<tr>
<td>26 Ethical hacking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>85.00%</td>
</tr>
</tbody>
</table>
in size should collect this kind of data.

Because of wide variations in removal efficiency, readers are urged to perform studies that utilize their own data, rather than the data shown in table 3.

One important source of defects is not shown in table 3 and that is “bad fix injection.” About 7% of bug repairs contain a fresh bug in the repair itself. Assume that unit testing found and removed 100 bugs in an application. But there is a high probability that 7 new bugs would be accidentally injected into the application due to errors in the fixes themselves. (Bad-fix injections > 25% may occur with error-prone modules.)

Bad-fix injection is a very common source of defects in software, but it is not well covered in either the literature on testing or the literature on software quality assurance. Bad fix data has been collected since the early 1970’s and the results have stayed fairly constant, indicating that this is a chronic problem.

Summary and Conclusions

In spite of the long history of testing and the large number of test personnel employed by the software industry, a great deal more research is needed. Some of the topics that need research are automatic generation of test cases from specifications; developing reusable test cases; better predictions of test case numbers and removal efficiency; and much better measurement of test results in terms of defect removal efficiency levels.

The literature on testing is extensive but almost totally devoid of quantitative data that deals with defect removal efficiency, with testing costs, with test staffing, with test specialization, with return on investment (ROI), or with the productivity of test personnel. However there are dozens of books and hundreds of web sites with information on testing.

Given the central role of testing over the past 50 years of software engineering, the gaps in the test literature are surprising and dismaying. A technical occupation that has no clue about the most efficient and cost-effective methods for preventing or removing serious errors is not qualified to be called “engineering.”

But to achieve really high levels of quality in a cost-effective manner, testing alone has always been insufficient and remains insufficient in 2009. A synergistic combination of defect prevention and a multi-phase suite of defect removal activities that combine inspections, static analysis, automated testing, and manual testing provide the best overall results, with the potential to raise average defect removal efficiency levels from the approximate today’s average of about 85% in 2009 up to about 97%. Even levels that approach 99.9% could be achieved many cases.

Effective combinations of defect prevention and defect removal activities are available in 2009 but seldom used except by a few very sophisticated organizations. What is lacking is not so much the technologies that improve quality, but awareness of how effective the best combinations really are. Also lacking is awareness of how ineffective testing alone can be. It is lack of widespread quality measurements and lack of quality benchmarks that are delaying improvements in software quality.

The final conclusion is that until the software industry can routinely top 95% in average defect removal efficiency levels, and hit 99% for critical software applications, it should not even pretend to be a true engineering discipline. The phrase “software engineering” without effective quality control is a hoax.

Books and Readings on Software Testing and Quality Control


Crosby, Philip B.; *Quality is Free;* New American Library, Mentor Books, New York, NY; 1979; 270 pages.


Gack, Gary; Applying Six Sigma to Software Implementation Projects; http://software.isixsigma.com/library/content/c040915b.asp.

Hallowell, David L.; *Six Sigma Software Metrics, Part 1*; http://software.isixsigma.com/library/content/03910a.asp.


---

**About the Author**

Capers Jones is a renowned author, consultant, and speaker. He was the founder and chairman of Software Productivity Research (SPR), where he retains the title of Chief Scientist Emeritus. Jones is the author of *Estimating Software Costs, Second Edition*, and *Applied Software Measurement*, 3rd Edition, 2008 as well as many other books and articles. As noted in the article introduction, this article is an excerpt from chapter 5 of Jones’ new book titled “Software Engineering Best Practices” to be published by McGraw Hill later in 2009.

**Author Contact Information**

Email: Capers.Jones3@Gmail.com
Technical Training On Demand

Cost effective way for organizations to provide continuous learning opportunities for their employees

- Pay one annual subscription fee ($475)
- Take as many classes as you want

Classes available 24/7/52

From:
- Home
- Office
- Travel

AFFORDABLE

DACS On-Line Learning Center

ACCESSIBLE

450+ classes
12,000 topics
Programming + Web Development
Course Catalog

Latest Technologies
Java  XML
Oracle  Server Technologies
.NET Framework

COMPREHENSIVE

To view the catalog visit
www.thedacs.com/training
For details call 1.800.214.7921

FLEXIBLE

- Fit course work into your schedule
- Self paced
- Refresh knowledge
- Study during lunch or after work

ACCREDITATION CERTIFICATION

ORACLE

- CEUs granted
- LACET
Accreditation Supported

MICROSOFT
Certification Suite
- MCAD
- MCDRA
- MCITP
- MCPD
- MCSD
- MCSE
- MCTS
- MCAS
The STN is a theme-based quarterly journal. In the past DACS has typically solicited specific authors to participate in developing each theme, but we recognize that it is not possible for us to know about all the experts, programs, and work being done and we may be missing some important contributions. Therefore, beginning in 2009 DACS is adopting a policy of accepting articles submitted by the software professional community for consideration.

DACS will review articles and assist candidate authors in creating the final draft if the article is selected for publication. Note that DACS does not pay for articles published. Note also that submittal of an article constitutes a transfer of ownership from the author to DACS with DACS holding the copyright.

Although the STN is theme-based, we do not limit the content of the issue strictly to that theme. If you submit an article that DACS deems to be worthy of sharing with the community, DACS will find a way to get it published. However, we cannot guarantee publication within a fixed time frame in that situation. Consult the theme selection page and the Author Guidelines located on the STN web site (see https://www.softwaretechnews.com/) for further details.

To submit material (or ask questions) contact news-editor@thedacs.com

Themes for 2009 are:
- Earned Value
- Software Testing
- Project Management
- Model Driven Development
The first 50 people to send in a completed survey will receive a FREE DoD/IT Acronym CD from the DACS.

This valuable CD-ROM contains over 9,000 Department of Defense and Information Technology acronyms. There are hundreds of acronym lists available but none are as well done as this CD AND specifically targeted towards DoD and Information Technology. This unique-shaped CD-ROM plays in your computer's regular, hub-mounted, CD drive. You'll use this great resource over and over again. It's FREE, just for filling out our brief survey on the next page!

Data & Analysis Center for Software (DACS)

http://thedacs.com

Data & Analysis Center for Software (DACS)
SOFTWARE TECH NEWS FEEDBACK

1. Which volume of the Software Tech News did you receive?  

STN Vol. 12, No. 2

2. When did you receive the newsletter? (month/year)  

3. How satisfied were you with the CONTENT of the newsletter? (Article Quality)  

- Very Satisfied  - Satisfied  - Neither Satisfied nor Dissatisfied  - Dissatisfied  - Very Dissatisfied

4. How satisfied were you with the APPEARANCE of the newsletter?  

- Very Satisfied  - Satisfied  - Neither Satisfied nor Dissatisfied  - Dissatisfied  - Very Dissatisfied

5. How satisfied were you with the OVERALL QUALITY of the newsletter?  

- Very Satisfied  - Satisfied  - Neither Satisfied nor Dissatisfied  - Dissatisfied  - Very Dissatisfied

6. How satisfied were you with the ACCURACY of the address on the newsletter?  

- Very Satisfied  - Satisfied  - Neither Satisfied nor Dissatisfied  - Dissatisfied  - Very Dissatisfied

7. Approximately how much of the newsletter do you read?  

- The entire issue  - Most of the content  - About half the content  - Briefly Skimmed  - Didn’t Read

8. Would you read this newsletter in an E-mail newsletter format?  

- Definitely  - Probably  - Not Sure  - Probably Not  - Definitely Not

9. How did you request the product or service?  

- Phone Call  - E-mail  - DACS Website  - Subscription Form  - Other ________________

10. Would you recommend the Software Tech News to a colleague?  

- Definitely  - Probably  - Not Sure  - Probably Not  - Definitely Not

11. What topics would you like to see this newsletter devoted to?  

_____________________________________________________________________________________

_____________________________________________________________________________________

Comments (optional)  

REGISTER FOR THE FIRST TIME | UPDATE CURRENT SUBSCRIPTION

Name: _______________________________  Position/Title: _______________________________

Organization: ___________________________  Office Symbol: ___________________________

Address: _______________________________  City: _______________________________

State: ___________________  Zip: ____________  Country: ___________________________

Telephone: _____ - _____ - _____  Fax: _____ - _____ - _____  Email: _______________________

Functional Role: ___________________________

Organization Type:  

- Air Force  - Army  - Navy  - Other DoD 

- Commercial  - Non-Profit  - Non-US  - US Government  - FFR&D  - Other _______________

*Note: You must give us your address to receive the CD.
ABOUT THE SOFTWARE TECH NEWS

STN EDITORIAL BOARD

Ellen Walker
Managing Editor
ITT Corporation, DACS

Dan Ferens
Co-Editor
ITT Corporation, DACS

Philip King
Production Editor
ITT Corporation, DACS

Wendy Butcher
Graphic Designer
ITT Corporation, DACS

Paul Engelhart
DACS COTR
Air Force Research Lab

Morton A. Hirschberg
Editorial Board Chairman
Army Research Lab (retired)

Dr. Kenneth E. Nidiffer
Software Engineering Institute

Dennis Goldenson
Software Engineering Institute

John Scott
Mercury Federal Systems

ADVERTISEMENTS

The Software Tech News is now accepting advertisements for future newsletters. In addition to being seen by the thousands of people who subscribe to a paper copy, an electronic version is available at the DACS website, exposing your product, organization, or service to hundreds of thousands of additional eyes every month.

For rates and layout information contact: news-editor@thedacs.com

COVER DESIGN

Wendy Butcher
Graphic Designer
ITT Corporation, DACS

ARTICLE REPRODUCTION

Images and information presented in these articles may be reproduced as long as the following message is noted:

“This article was originally published in the Software Tech News, Vol.12, No.2 July 2009. Requests for copies of the referenced newsletter may be submitted to the following address:

Philip King, Production Editor
Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

Phone: 800-214-7921
Fax: 315-838-7130
E-mail: news-editor@thedacs.com


In addition to this print message, we ask that you notify DACS regarding any document that references any article appearing in the Software Tech News.

ABOUT THIS PUBLICATION

The Software Tech News is published quarterly by the Data & Analysis Center for Software (DACS). The DACS is a DoD sponsored Information Analysis Center (IAC), administratively managed by the Defense Technical Information Center (DTIC). The DACS is technically managed by Air Force Research Laboratory, Rome, NY and operated by ITT, Advanced Engineering and Sciences Division.
IN THIS ISSUE

Tech Views
by Ellen Walker, STN Managing Editor ................................................................. 3-4

Network Centric Warfare, Total Systems Design & Testing
by John Scott, Defense Consultant ........................................................................... 5-14

Competitive Testing: Eyes on the Quality Prize
by Brendan Wright, TopCoder, Inc. ........................................................................ 16-23

Automated Software Testing Distilled - Standards and other Considerations as Prerequisites to Success
by Elfriede Dustin, Bernie Gauf, and Thom Garrett, Innovative Defense Technologies (IDT) ................. 25-34

Predicting Test Cases and Test Team Size
by Caper Jones, President, Capers Jones & Associates LLC ........................................ 36-42