Software Engineering Institute

Using Software Development Tools and Practices in Acquisition

Harry L. Levinson
Richard M. Librizzi

December 2013

TECHNICAL NOTE
CMU/SEI-2013-TN-017

Software Solutions Division
http://www.sei.cmu.edu

Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the
SEI Administrative Agent
AFLCMC/PZM
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

DM-0000791.

Table of Contents

1 List of Tables

Acknowledgments

We would like to thank the following people from the Software Engineering Institute for asking questions, posing challenges, suggesting solutions, and reviewing this document:

Grady Campbell
Mike Philips
John Foreman
Ted Marz
Bob Ferguson
Pat Place
John Robert
Fred Schenker
Robert Seacord
Jeff Thieret
David Zubrow
Lorraine Adams
Gerald Miller
Eileen Wrubel

This technical note is dedicated to Richard Librizzi, who passed away prior to publication.

Abstract

Acquiring software-reliant systems from an external resource can be time consuming, costly, and often unreliable. During independent technical assessments and customer engagements, the Software Engineering Institute observed many contractors who are not utilizing mature, readily available software development tools when creating code for government programs. These tools include static analysis, test automation, and peer review techniques. In addition to the aforementioned tools being important for software developers, they offer significant insight and confidence to customers who acquire software products.

There are many benefits from making these tools and practices integral to software development and acquisition processes, including:

- reduced risk: programs deliver more predictably
- improved customer satisfaction: products developed experience fewer field defects
- lower cost of ownership: programs experience lower life-cycle maintenance costs when deployed operationally

This technical note provides an introduction to key automation and analysis techniques, the use of which the authors contend will benefit motivated acquirers and developers.

1 State of Software Acquisition and Development in Large Systems

Increasing demands for more functionality alongside improving important system qualities like reliability, availability, and security are driving up the complexity of developing software-reliant systems. Continually striving to reduce cost and schedule is a key requirement for surviving in a competitive market; often, the result is that low-quality code is rushed to production. For large mission-critical systems such as military and banking applications, significant delays may occur and not be fully realized until late in the program development cycle. Although it is not usually obvious, one of the more effective actions to improve cost and schedule is to improve efficiency of code development. By doing this, more time can be spent on design and test efforts that further support the overall quality of the delivered system.

A considerable number of robust and reliable software development tools and practices have been developed over the years and have been accepted as state of the practice, enabling developers to deliver quality products on a predictable schedule. The open-source community has created inexpensive and effective tools. In addition, many are not complicated and with minimal training can be used across an organization's development process. These tools and practices also become critical to the success of the use of new methodologies such as agile software development. Therefore, it is not clear to the authors why many acquisition and software development programs we have encountered do not incorporate the use of these tools into their software development processes.

This section explores the current state of software acquisition and development with a focus on challenges related to large system development. Section 2 describes recommended tools for every software development project. The Section 3 looks at automation, peer reviews, and some selected management processes that have tools and practices applicable to most phases of software development. Section 4 concludes with recommendations as to how acquisition organizations should apply these development tools and processes.

1.1 Increasing Quality and Development Efficiency

Creating software is a complex process that requires large numbers of people with a diverse range of technical skills. Making improvements to the software development process has been a challenge for many years, but throughout the short history of software development, a wide range of improvements have allowed the industry to increase the functionality, quality, and development efficiency of delivered systems. This list includes highly structured approaches for process improvement like CMMI® and software development processes like Rational Unified Process (RUP). Somewhat orthogonal to these structured methods are software development models such as waterfall, spiral, and agile. One can also look at the various design modeling approaches such as structured decomposition, object oriented, aspect oriented, and others. Additionally, over the past 25 years or so, a focus has been on software architecture as an impactful way to identify important quality attributes (QAs) early in the design process. The architecture area also included a focus on architectural and more detailed design patterns as a way of addressing common issues

found during software development. These and other ideas were directed toward addressing the challenge of how to best deliver quality code on a predictable schedule and cost.

Increased quality earlier in the software development process provides increased productivity, and software development life cycles often become more predictable and thus more manageable, resulting in lower costs. In both commercial and government acquisitions, the quality of the developed software is a key factor to the overall cost and schedule of the delivered software system and software sustainability throughout the lifecycle. Effective software development processes, utilizing accepted software engineering practices and tools, make for more predictable software development.

During the development process there are many opportunities for the injection of defects into the resulting code. A typical software development process usually consists of these general activities:

- requirements engineering
- architecture and detailed design
- code and unit test
- integration
- system test

Many of today's development methodologies encourage the above processes to occur simultaneously, often overlapped in time to some extent, and continuously throughout the life cycle of the product.

A basic tenet of creating quality software is to detect and correct defects close to when they are introduced. Boehm and Papaccio reported in 1988 that requirements defects found after a system is fielded cost 50 to 200 times more to rectify than if they had been found and corrected near the beginning of the development [Boehm 1988].

The current situation is not much different today as industry data still supports the notion that errors in requirements specification are expensive to fix: Wiegers asserts that 50 percent of defects originate in the requirements phase, and 80 percent of the rework cost on a project can be traced to defects arising during the requirements specification phase [Wiegers 2001]. Granted that preventing defects in the early phases like requirements and design give the greatest payback, this paper examines tools and processes that help to find defects injected during the implementation phase. Quite often these defects are not found until the integration and system test phases. When the tools and processes recommended in this paper are used throughout the implementation phase, they can identify many defects that are relatively easy to find and fix.

1.2 Acquisition and Development Challenges

Many large development projects today involve an acquisition team (acquirer) working with a development team (developer) to create and deliver software-intensive systems. While the acquisition team usually focuses on the financial, business, and political aspects, they also need visibility into the technical development status and progress, and the quality of the developing system.

In the United States Department of Defense (DoD), acquisition of software-intensive solutions is primarily performed through the use of program offices contracting with defense contractors to define, develop, and deliver software-intensive systems. In the commercial world, acquisition is usually done through business and marketing teams working closely with internal or external developers to deliver the desired products. In both realms, the software development processes are similar so these tools and practices are applicable in both domains.

Government and commercial acquisitions have historically encouraged the use of process models and frameworks like CMM/CMMI, ISO9001, and Information Technology Infrastructure Library (ITIL) for many years in an attempt to gain confidence that the development team can consistently deliver as promised. Likewise, process methodologies like Total Quality Management (TQM), Six Sigma, and Lean have been part of the software engineering culture for many years. These models and methodologies are accepted as good practices, yet they have been slow to be adopted. While the authors firmly advocate the use of the above models and methodologies, this paper argues that acquirers can make some basic improvements to their oversight solely by using state-of-the-art software engineering tools and practices.

The CMMI-ACQ model (Capability Maturity Model-Integrated for Acquisition) discusses objectives that an acquirer should actively manage. The introduction to the CMMI-ACQ states: "The acquirer owns the project, executes overall project management, and is accountable for delivering the product or service to the end users. Too often acquirers disengage from the project once the supplier is hired. Too late they discover that the project is not on schedule, deadlines will not be met, the technology selected is not viable, and the project has failed" [CMMI Product Team 2010].

In addition, the acquisition technical management process area from CMMI-ACQ focus the acquirer on the following activities [CMMI Product Team 2010]:

- conducting technical reviews of the supplier's technical solution
- analyzing the development and implementation of the supplier's technical solution to confirm technical progress criteria or contractual requirements are satisfied
- managing selected interfaces

There are varied strategies, processes, and methods that the acquirer may leverage to develop an accurate understanding of developer progress, technical decisions, and the quality of the delivered product. They range from using formal, documentation-driven processes to including the acquirer as an active participant in the decision-making process. The more confidence that the acquirer has in the contractor's process, the less friction there will be between the two parties.

1.3 Acquisition and Development Process Improvement

Typically, acquisition organizations focus on schedule and cost and, as Watts Humphrey observed,

> If the organization is not cost competitive, or if it produces lower quality or less attractive products, a focus on current performance will not improve the situation. The immediate problems may be fixed and the burning issues resolved, but the organization will continue working pretty much as it always has. It will thus continue producing essentially the same

> results and generating essentially the same problems and issues. This brings us to the definition of insanity: doing the same thing over and over and expecting a different result [Humphrey 2009].

A key reason to improve productivity through the incorporation of software development tools and practices is to free up engineering talent from non-value-added activities. It is also a way to improve schedule and cost predictability. The large body of work in process improvement and change management demonstrates that successful organizations continually reflect on the current process and identifying opportunities for improvement. In almost any process improvement methodology, a key focus is on avoiding rework. Since humans, and thus software engineers, are not perfect, finding ways to detect and remove defects early is an important part of all software development. As discussed in section 1.1, it is generally less costly to detect and correct an error closer to the time when it was introduced.

The processes used by the developer to create the software directly affects the quality of the delivered software product. Mature software development tools and practices can give the acquirer important indicators and predictors of quality. A key challenge from the acquirer's point of view is gaining insight into the progress on and quality of the software being developed. Development of this insight can take many forms including

- observation of progress measures at key milestone events
- review of process artifacts to gain insight as to how closely the developer followed the defined processes
- review of intermediate work products and metrics
- participation in the actual development processes through active review of code and artifacts
- side-by-side participation of acquisition personnel supplying real-time requirements, peer review, and verification

The objective of process improvement efforts is to increase the effectiveness of system and software development talent. The end game is about producing software intensive systems that are high quality, within budget and schedule. The authors advocate the redirection of engineering efforts to be focused on higher value activities such as

- focus on deeper understanding of the written and implied requirements
- create and tune the architecture and detailed design of the system to improve the significant quality attributes including security, usability, flexibility, sustainability, and so forth
- continue process improvements by identifying more places to use tools and add automation

The acquirer should encourage and actively participate in improvements that enable software engineers to focus on requirements, design, and analysis activities that improve the quality of software-intensive systems. The next section explores tools recommended for use by both the acquisition and development teams that can help focus engineering talent to create quality software.

2 Software Development Tools

The software engineering community has an ever-increasing set of tools and practices aimed at improving the quality of developed software. As the development team includes these tools and processes into its software development methodology (coding, unit tests, integration, etc.), it helps to improve the development team's productivity, while at the same time it gives insight and confidence to the acquisition team. This section gives an overview of tools and practices available with specific details on the tools and practices most effective during the implementation phase.

2.1 Requirements, Architecture, and Design Overview

It is beyond the scope of this report to discuss at length all areas of requirements, architecture (high-level design), and detailed design, but it is important to point out that there is a vast array of tools and practices that can improve the effectiveness of the development team during these activities. Many of these methodologies take discipline to implement and execute, but as described in the first section, the goal is to get it right early in the project. Included in these are

- Requirements: Management databases to aid in tracking and tracing of requirements throughout the complete development life cycle
- Software Architecture: Modeling and simulation at a high level with focus on the quality attributes of the system. These visualization and simulation techniques enable software designers to consider the various software architectural tradeoff decisions plus can aid in design decisions such as implementation language, SEI COTS Usage Risk Evaluation methodology, reuse of code, and related Integrated Development Environment (IDE) environments.

The above activities are therefore important areas in which the acquisition team should participate and have insight into design and process decisions. Many formal methodologies (such as the SEI's Quality Attribute Workshop and Architecture Tradeoff Analysis Method) are available to help teams review and understand the architectural and design decisions that are made.

2.2 Implementation

As pointed out in many software development models and textbooks, writing code is usually a small part of the total effort needed to deliver a software product successfully. Yet, writing code presents the easiest opportunity for defects to be introduced into the product. Many automated tools exist to instrument code in order to catch and remove defects before allowing the code to move into integration and test. An important tool area is called static analysis, which often includes code beautifiers, interface extraction (i.e., reengineering), and documentation extraction. This type of tool supports the use of coding standards and peer-review processes (to be covered in the next section).

2.2.1 Coding Standards

The CERT C Secure Coding Standard contains a good description of a coding standard:

> Coding standards encourage programmers to follow a uniform set of rules and guidelines determined by the requirements of the project and organization, rather than by the programmer's familiarity or preference. Once established, these standards can be used as a metric to evaluate source code (using manual or automated processes) [CERT Secure Coding Standards 2011].

The objectives of coding standards are to increase the maintainability, reliability, and security of the code. A coding standard discourages software developers from using practices and behaviors that may create defective code.

A coding standard usually covers these four areas:

1. Common visualization: includes ways to have common layout, code indentation, and the usage of spaces
2. Naming standards: requires developers to give variables and functions meaningful names using a common naming convention (e.g. CamelCase, _under_score, and ALLCAPS)
3. Comment standards: give guidance as to how many and what types of comments are expected
4. Coding rules: help guide the programmer as to the proper use of common coding patterns such as if, switch, do, while, variable creation, initialization, and assignment

There are many specific features and automations in this area; some of them include

- code “beautifiers” and “pretty printers” to enforce layout features like indentation, alignment of structures, spacing, matching parentheses, and brackets
- comment extractors to create stand-alone documentation from pre-formatted comments within the code
- static analysis enforcing the use of code constructs and rules from the coding standard including naming standard and comment rules in addition to discourage poor coding practices

The security community has developed coding standards such as Common Weakness Enumeration (CWE) and the aforementioned CERT C Secure Coding Standard. Other communities have created similar coding standards to deal with safety, performance, and maintenance. These are published with the goal of making the set of coding guidelines and best practices easily available to all developers so they can incorporate them into their coding standards. The rules and recommendations in these standards are usually based upon the coding language and how it is used. They define common, yet poor coding patterns that developers tend to use that affect these software attributes.

2.2.2 Static Analysis

Static code analysis is the examination of aspects of the software system implementation that are available prior to execution. The concepts of the implementation language are brought together with the coding standard as the metric against which the code is automatically inspected. Static analysis tools analyze the components and resources of an application without having to run the application. This contrasts with dynamic analysis, which requires one to run the application. Be-

cause of this capability, it can be used at an earlier phase of the development cycle, therefore exposing and correcting problems before entering the integration and test phase.

Static analysis can be viewed as an automation of the coding standard and the peer-review process. The objective of these tools is to find defects at machine speeds by removing the burden of manual syntax checking. Static analysis reviews compare the source code of an application against a set of coding rules to

- ensure that the source code complies with the selected standards
- find unwanted dependencies
- ensure that the intended structural design of the code is maintained

Static analysis is a mature tool type that, based on SEI customer engagements, is underutilized in many of today's software engineering practices. As developers find ways to incorporate the use of static analysis into their daily development processes, it brings many benefits:

- common coding patterns that lead to defects are found almost instantaneously
- gives developers, their management, and the acquirer confidence that the "simple" defects are found early
- frees up valuable software engineering time to focus on the tougher challenges

A common comparison in the industry is that static analysis tools should be treated like spell checkers; human intervention is required to find the important problems. Consequently, static analysis tools will *not* answer questions such as:

- Is the code organized so that another developer can read and maintain it?
- Is it a robust and efficient implementation of the appropriate algorithm?
- Does the code work as required and documented?
- Are the unit tests sufficient and correct?

When static analysis is used as part of a software development process, it gives the development engineers and thus the acquirers confidence that the basic coding, design, and style issues have been resolved so that they can focus on these more difficult questions.

There are many open source tools along with commercial static analysis tools available in today's market. Each has various features, so selecting one for use is a project unto itself. Table 2 below contains a partial list with brief descriptions of defect detection that can be accomplished by today's static analysis tools:

Table 2: Some Types of Defect Detection Possible With Static Analysis

Defect	Description	Resolution
Potential null dereference	Code attempts to call a method or access a field of an object that has been set to null	Check for failure after memory allocation
Unused variable	Variable is never written to or read from	Remove from code base
Overly broad throw of exception	Catching any exception that is de-	Understand all exception handling

Defect	Description	Resolution
	rived from the top-level Exception class but treating it like a typical non-runtime exception masks runtime errors	and correct using best practice
Logging warnings/errors inconsistently	Logging is not consistent throughout	Pick a single logging mechanism and use it exclusively
Unchecked method return values	Returned values of method calls are ignored	Check all return values
Bypassing of architectural or security-based API	APIs that exist to enforce consistent resource allocation or input validation are not used in all situations	Enforce use of API to maintain architectural and security integrity
SQL injection, path manipulation, command injection, log forging	When querying or commanding a database or operating system, user input is used directly to form the executed instruction	Use parameterized SQL statements that bind parameters; validate and whitelist all input used in dynamically created instructions
Unreleased resource	Calls to resource allocations (memory, threads, etc.) are not matched up with deallocation calls	Ensure for every resource used that the appropriate management methods are used
Use of dangerous functions related to string management	Some common functions exist that cannot be used safely or are difficult to use safely	Use approved functions in the approved manner
Debug code in released version	Debug code can affect security, performance, and reliability and should be removed before release to the user	Remove all unit test and diagnostic code that is not intended for ongoing product support
Empty exception catch block	Exceptions not handled will lead to unexplained software failures	Deal with all expected and unexpected error situations making sure the information is available to diagnose failures

Quite often today, static analysis tools also include features that help developers visualize the organization and interfaces of the major components of the software. The ability to analyze and visualize the structural and architectural features of the developing product allows for early analysis of design decisions and confidence in the architectural integrity. In addition, these tools also use various methods and metrics to document the size and complexity of the developing software. This allows the acquirer and developer to monitor progress and detect changes that will impact program cost and schedule.

Some of the criteria to look for when deciding on what tool to use

- built-in rules compatible with the project's standards
- ease of managing the rules (including deleting, creating, and reporting)
- ease of integration with other software development tools being used
- automation of running and reporting of results

2.3 Testing

In addition to the implementation phase of the software development process, static analysis and other tools are necessary to integrate, verify, validate, and certify the delivered system. The acquirer needs to understand how these processes are executed as the results are quite often key inputs to the acquirer's final acceptance process. Integration and verification (also called integration

and test) are quite often planned and executed by the same team of engineers. This sometimes causes confusion as these are separate processes with a different set of goals.

Integration is the activity after coding when the components are combined to make a working system. A good practice here is to build up the system step by step from only a few components to create a baseline working system. The ability to have a simpler, working baseline version aids in identifying defect root causes. The key objective of this process is to make sure the interfaces that were designed on paper and then implemented are working together. If a formal interface technique like IDL for Common Object Request Broker Architecture (CORBA) or Simple Object Access Protocol (SOAP/HTML) for web services is used, then there are static analysis tools available to verify the actual implementation by checking that the interface definitions and semantics are consistent. These tools define the interfaces by using a common definition language; allowing the components to be built automatically with specialized tools, they can compare models from the design phase with the actual implementation.

Verification quite often is merely considered the testing phase after integration. This is not true. Verification includes the testing activities from the start of development and includes test planning, test procedure development, unit testing, test automation (or not), test coverage, system testing, and more. A challenge that many acquisition programs face is developing testable requirements.

Quite often, it turns out that the verification is not clean and leads to various types of rework. A common practice is to write the requirements without a clear idea as to how they will be verified at the end of the development process. By using automated tests, the test can be written early in the development process to help define the requirement and allow it to be run frequently during the development, integration, and test phases.

The dynamic analysis tools discussed below are used in both integration and verification. Automation (discussed further in section 3.1) is usually developed and debugged for initial use during the integration stages of the project and will be further employed during verification and possibly the certification activities. A key benefit to the acquirer is that these same automated tests become valuable as regression test suites during the sustainment phase of the product life cycle.

2.3.1 Dynamic Analysis

Dynamic analysis is an important type of tool usually used during integration and verification that helps to check for various qualities of the code such as performance, security, and availability. The use of dynamic analysis must be planned for from the beginning of the design phase. Dynamic analysis uses data gathered from additional instrumentation code whose purpose is to capture the events in real time. While the capability needs to be included early in the design process, the system needs to be sufficiently integrated for dynamic analysis to be useful. Thus, defects are found later in the process than those discovered by static analysis tools. Dynamic analysis checks for

- performance degradation
- memory management issues like leaks, bad allocations, and memory corruption
- threading and interlock conditions that can impact performance or cause run-time failures

- race and deadlock conditions
- security vulnerabilities
- code coverage monitoring

These are important features to the acquirer, and the use of dynamic analysis should be planned for early in the system design phase even though the actual analysis does not occur until late in the development cycle.

2.3.2 Use of Static Analysis in Certification

Quite often, large acquisition programs face certification challenges by an independent organization. This accreditation or authorization to be utilized for the intended purpose happens frequently in systems intended for government, military, medical, and financial use. Certification is an approval process for the system or parts of the product, usually with a strong focus on security. These processes typically involve checking that the code meets selected coding standards either through manual or automated means.

A key activity of certification usually involves independent static analysis of the code base. If this analysis is executed against a different coding standard or using a different set of static analysis rules and patterns than the developer used, it will tend to show a large number of potential defects —many of which will be later identified as false positives. For each positive, the developer and certification authority must then negotiate to determine whether the flagged code is in fact in compliance (false positive), may remain in place as a documented deviation, or requires repair. According to CERT's recent *Source Code Analysis Laboratory (SCALe) for Energy Delivery Systems* report,

> Deviations should only be used infrequently because it is almost always easier to fix a coding error than it is to provide an argument that the coding error does not result in vulnerability…. Depending on the analyzers used, it is not uncommon for code bases to have substantial numbers of false positives in addition to the true positives that caused the software to fail conformance testing. False positives must be eliminated before a software system can be certified. However, analyzing the code to determine which diagnostics are false positives is time consuming and labor intensive. Furthermore, this process needs to be repeated each time the code base is submitted for analysis. Consequently, preventing the issuance of diagnostics determined to be false positives can reduce the cost and time required for conformance testing in most cases [Seacord 2010].

The ongoing challenge by the acquisition team is to continually certify new releases of the code. In attempting to obtain certification in a timely fashion, the certifying organization sometimes participates early in the acquisition process. Acquirers, security, certification, and test teams should make sure the right coding standards and specifications are part of the contract and enforced by the software developer on a daily basis by using appropriate static analysis tools along with other system engineering practices.

3 Overarching Software Development Practices

When considering many of the various software development process areas, mature tools and practices with proven techniques are quite often available. Executing the process areas well across the whole development life cycle is important to the timely delivery of quality software. However, for the purpose of this technical note, we focus on two practices—automation and peer review. followed by a quick look at defect management and quality assurance that support the automation and peer review practices.

Quite frequently automation and peer review practices are skipped or deferred during the development process in an attempt to meet cost and schedule. As these practices have been widely recognized over the years to pay off in terms of quality, they should not be dismissed lightly. From the acquirer's point of view, automation and peer review—when executed appropriately—can supply confidence-building insight into the acquisition.

3.1 Challenges of Automation

Christie et al define automation as "…computer-based support for the flow of work between individual tasks. Processes (large or small) are said to be automated if manual control of task initiation or sequencing is transferred to the computer [Christie 1996]." The implementation and complexity of automation changes over time due to new and improved tools and techniques. But in effect, software development and test automation tools perform what were previously manual processes and allows them to be completed without human intervention.

The concept of automation can be applied across the implementation, integration, and test processes used to develop a software product. COTS tools are available that can quite often easily be incorporated into most development processes. Common automations include

- automated test frameworks or test harnesses which enable developers to test the code from unit to integration testing
- user interface automation to enable the repetition of user text and mouse input along with verification of the expected events from each input
- COTS products geared toward online application testing

Planning for the use of automation early in the life cycle brings several key advantages to both the developer and acquirer:

- Repeatability of testing allows for consistent regression testing, leading to confidence and predictability in early and operational releases.
- Automated frameworks can enable the development and/or sustainment team to add new features efficiently and simplify the remediation of defects.

Deciding whether and how to implement automation tools and techniques is as important as implementation decisions within the product itself. Although there are many opportunities to use automation within software development processes, the decision to automate a particular technical process usually involves significant tradeoff analysis. The automation process alone creates the need for the development organization to develop expertise in the use of any automation tools they employ, requiring staff training and possibly the hiring of new talent. In addition, creating and maintaining the automation can become a project unto itself. Maintaining automation scripts and software has the same types of challenges associated with maintaining other software products and therefore must be considered when deciding to automate. Automation tools themselves necessitate requirements, design, and documentation with the challenge to keep the code updated and documented accurately. Quite often, automation objectives also require a make-versus-buy decision with all of the associated tradeoffs to be considered.

Automated tools are extremely important in creating a high-tempo development process. Automated check-in and analysis of code bases can quickly point to where changes have caused issues, and attention can be directed more readily to those areas. After incorporation of basic automation, further concepts are worthy of pursuit. Automated analysis of higher order issues associated with component interaction, interface dynamics, and model-driven architecture are some examples. In most of today's development methodologies using iterative approaches, the ability to automate is considered a necessity for success. When automation is planned for and applied early, the benefits ripple across the product's life cycle.

3.1.1 Automatic Code Generation

Automatic code generation is used in combination with high- and/or detailed-level design and modeling tools. Once the design is finalized, the code is generated using the design and implementation details supplied by the design tool. In the case of a high-level design tool, the skeleton or outline of the code is automatically generated to allow manual creation of the actual functions or methods. Depending on the level of detail in the design or modeling tool, the actual software for the functions or methods may in fact be created.

Many of the activities that a programmer does while writing code offer opportunities for automation. Detailed design tools allow the developer to specify the software's parameters and common algorithms in a format that is easily viewed and maintained. This allows a code generation tool to produce the code.

A key consideration when generating code is how that code will be sustained. The generated code can be considered a starting point for manual programmer modifications such as creating a framework of code modules from an initial high-level design. A better practice is to use the design or modeling tool to make changes which requires maintenance of the tool and the associated model throughout the sustainment period. Another good practice is to clearly identify how the generated code was written to enable future evaluators and sustainers to make effective decisions.

Automatic code generation enables the developer to easily employ a defined, repeatable process in a consistent coding style. This gives the acquirer confidence that development is done in a repeatable fashion, following good programming practices that are geared toward preventing defects.

3.1.2 Continuous Integration and Test

Continuous integration can be effective for almost all types of software developments. Continuous integration is enabled by using a configuration management system with the ability to automate the build process. Successfully built code is then automatically tested to verify that the newly built product works correctly. Automating the build and unit test environment is common in many of today's software development processes. This enables daily build and integration along with automated static analysis scans and unit tests. A greater level of automation enables quick discovery of defects and enables the removal of defects closer to the insertion point.

With some scripting, the code can be generated, compiled, statically analyzed, unit tested, and delivered into an integration environment automatically within hours. Errors found during these steps can be sent to the appropriate developer for immediate action. With this kind of response, the time between implementation and system testing can be made extremely short so that developers and testers can focus on their primary activities. The development team's goal should be to improve this process so that defects are found and removed as soon after writing the code as possible.

3.1.3 Test Automation

Instituting test automation enables the developer to shorten implementation and system test cycles, giving the acquirer greater insight through the automatic collection of accurate and revealing feedback into the state of the developing product. As described in previous sections, automation of the static analysis and unit testing can easily be built into the development process. Tools also exist to enable automation at a higher level so that regression and functional testing can be done on a frequent basis. The ability to create a test that is repeatable without human intervention allows both the developer and acquirer to gain confidence in the system more quickly. As described in the previous section, automation does not come for free. The ongoing maintenance of an automated test framework can be a project unto itself. There are many books, courses, and tools available on this subject.

3.2 Peer Review

For the purposes of this technical note, the authors chose a broad definition of peer reviews. Peers include not only the technical team members but also include stakeholders such as the acquisition team members and the end users. Widening of the peer review concept beyond the bounds of the technical team encourages earlier looks by acquirers and customers/end users so functional and user experiences can be dealt with sooner.

> Peer review is the evaluation of creative work or performance by other people in the same field in order to maintain or enhance the quality of the work or performance in that field.
>
> It is based on the concept that a larger and more diverse group of people will usually find more weaknesses and errors in a work or performance and will be able to make a more impartial evaluation of it than will just the person or group responsible for creating the work or performance [LINFO.org 2005].

Participation by the acquirer in peer reviews allows for deeper insight into

- the development team's level of understanding of the requirements
- the design decisions that are made by the development team
- the quality and sustainability of the developed software early in the life cycle
- coverage and depth of the planned verification activities

Peer review methodologies encourage getting more eyes looking for problems. As discussed in section 2.1, various methodologies can be used throughout the development life cycle to review the architecture, design, and developed code. These techniquesinclude (but are not limited to) formal inspections, walkthroughs, and pair programming. The concept of peer reviews is to geta few people in a room to focus on a small section of code. Today there are also some workflow management tools that allow geographically or time-dispersed teams to hold peer reviews.

These methods are great ways to share the knowledge and increase communication between the acquisition and development teams. Each of these methods has benefits and weaknesses and relies on the team's communication and social capabilities. The use of static analysis tools and code beautifiers before the peer-review process begins will help the group focus on the tough requirements, design, implementation, and defect resolution issues.

3.3 Common Software Development Processes

The tools and practices in addition to automation techniques discussed earlier in the paper are effective additions to many common software engineering processes. This section explores how they can be incorporated into configuration management and quality assurance.

3.3.1 Configuration Management

Configuration management (CM) is a very mature area in which the market offers a wide variety of solutions ranging from those that may be free, open source, and/or commercially marketed. Acquirers should make sure their development organizations integrate these tools into their daily work processes to manage the identification and control of the software being created. Some of the interesting measures that acquirers should consider reviewing are frequency of check-ins, software size, and version information. Proper use of these tools along with the appropriate reports is a basic confidence builder for the acquisition team that the developer is doing this most basic software engineering process appropriately.

One of the more recent features available in CM tools is the automation of steps such as check-ins and other workflow elements (e.g., initiating cycles of static analysis, compiling, and builds upon check in). A challenge in making automation successful is to use a CM tool that makes the implementation of this automation simple and effective in the contractor's development environment, which may mean seeking out easy-to-use interfaces, instantaneous check-in times, and effective remote access.

The acquisition team relies on the developer to create a CM plan to manage the software-related work products in addition to developing code according to the plan. Furthermore, remember that acquirers also bear responsibility for CM on programmatic documents. To enable the acquisition

team to participate effectively in peer reviews of the developing system, appropriate remote access should be easy and accessible as the product evolves.

A jointly agreed upon CM plan including use of automation features and easy access by both parties is important to gain efficient and effective interaction between the developer and acquirer. For example, such a plan enables a program manager to understand how the CM system maintains linkages between technical and programmatic documentation

3.3.2 Defect Management

Defect management processes help answer some of the most common questions that acquisition teams ask: What is the quality of the product and when will it be finished? Tools range from homegrown databases to integrated commercial off-the-shelf (COTS) products for tracking defects from start of development, through deployment, and into sustainment. In addition to managing defects on a daily basis, these tools help gather measurements that help the development and acquisition teams gauge the quality of the developing product. Data such as defect density and defect detection/removal efficiency are important for knowing the quality of the system and for deciding when to release it.

This class of tools enable *root cause analysis*, which is a key activity that requires the analytical capabilities of the technical skills of all the team members. Root cause analysis seeks to identify the source of a problem so that classes of defects can be found and corrected in both technical and process-related areas. Proper use of defect management tools will help to minimize the engineering effort needed to gather information, measure, and report patterns in defects.

3.3.3 Quality Assurance

Quality Assurance (QA) takes many forms from company to company. An important element from the acquirer's point of view is that the QA processes and methodologies can help the acquirer gain insight into the developer's progress. QA monitors the adherence to the defined development process in addition to quality checks of the developed artifacts, supporting the acquisition team's goal of a quality product. Since the acquisition team has ultimate responsibility for what is being developed, they should witness key development team QA activities to gain confidence that the development team is working to the planned development process. Activities can include

- observation and participation in peer review and evaluation activities
- observation of development team verification and validation activities
- observation and review of process compliance reviews

In addition, the acquisition team can gain confidence that many of these basic software engineering processes are well-instituted if the tools are used properly on a regular basis. Several examples include the following:

- requirements management repositories – quickly check and visualize the tracing between parent and child requirements along with tracing to design and test elements
- static analysis – constantly enforce the coding standard

- configuration management – quickly check and visualize baselines for all types of design artifacts from requirements to design to code to test

All of the tools and practices discussed in this technical note, including automation, enable QA and the acquisition team to gain insight quickly, thus allowing time to focus on design and decision activities over rote tasks.

4 Summary and Concepts for Further Consideration

The use of the tools and practices discussed in this technical note should be expected and encouraged by the acquisition team as they are mature, common tools used in today's software development. As such, the tools supply key insights into the quality of the developing product along with indicators as to when the product will be completed.

4.1 Justification for Improvement

Unfortunately, quite frequently the tools and practices discussed in this technical note are considered extra and unnecessary when schedules are tight and limited funding is available. There are budget and schedule considerations to using these tools and practices such as tool costs including the number of licenses, ongoing maintenance of the tool, and training costs over the whole software development cycle. In addition, the dedicated time to execute many of these practices and to manage these tools is also perceived as an ineffective use of costs and schedule.

Improved quality is sometimes hard to measure, but it tends to manifest in less effort needed due to

- deliver the product more predictably, which leads to lower development cost and schedule
- reduced defects in the released product, thus leading to lower maintenance costs
- improved development processes, leading to lower enhancement costs through the ability to add new features more easily

4.2 Acquisition Strategy and Adapting to Change

A common challenge to large projects and programs is that they usually last many years. This fact alone is likely to guarantee that requirements, people, tools, and practices will change multiple times during the development effort. If the acquisition strategy does not account for adapting to change, then it is setting up the acquisition for failure. While there is the feeling of safety by keeping the development environment constant, not upgrading the tools and practices in a protracted project forces the development team to work with older, less effective tools on what is typically a much more complicated system.

Why are opportunities to improve key to process improvement? It is the planning to improve throughout the project that enables the benefits of process improvement; chiefly, improved quality of the final product. Process improvement can take place only when tools and practices are allowed to change. If the change cycle takes years for each modification, then very little improvement can happen. This means that the acquirer should find an acquisition strategy that allows for development tools and practices to occur frequently while measuring the results.

Consider

- frequent update of COTS tools to keep them current and allow the use of the latest efficiency features

- ongoing sustainment of in-house developed tools to keep them useful
- training for new members of the acquisition and developer team in existing tools and practices
- updating and the associated training for new tools and practices

4.3 Acquirers and Developers

As an acquirer, instead of asking your development team typical status-related questions such as ”When will you be done?,” the authors recommend trying some of these questions:

- What is painful about the current software development process?
- What code quality improvement suggestions have you made recently?
- What automation and peer-review improvements are you piloting?
- Is quality assurance verifying adherence to previously agreed upon processes?

Consider these simple example strategies:

- Look for developers to use static analysis on code check-in as part of the software development process, and explain the types of analysis that they perform. This process can be automated so that the impact is minimal and bugs are dealt with before unit and regression-level testing.
- The use of static and dynamic analysis tools as part of the development process can provide acquisition and certification organizations with key insights. A close relationship with the development team can allow the acquirer and certifier the opportunity to encourage the use of appropriate coding and testing standards and practices on a continual basis. The execution of these processes can then be audited and inspected to gain confidence and certify that the software meets some subset of the required standards. The risk of failing certification at the end is greatly reduced.
- The acquisition team should receive frequent, interim integration and test reports from the development team to provide confidence in the expected quality, cost, and schedule. A formal report that is completed at the time of delivery can be used for the certification process. This should be a non-event as the interim reporting process will discover problems early in development.

Usually the contractor has a variety of tools and practices in their toolkit already—backed up by corporate investment enabling deep experience and knowledge. An acquirer would do well to consider asking questions about where in the life cycle the developer employs the available techniques and tools. While the acquirer is rarely in a position to dictate about the tools and practices used, just asking the question and setting expectations helps to motivate the acquirer into using and enabling them.

This paper gives to both the acquirer and developer a better understanding of the benefits of using these tools and practices. There should be obvious justification now for the added cost and effort in training. The success of using these tools requires close involvement between the acquirer and developer organizations. In the past, insight by the acquisition team was focused at key mile-

stones. The use of automation and analysis tools and practices today allows for a collaborative environment, which reduces the risk, conflict, and rework that has plagued software developments in the past.

Bibliography

URLs are valid as of the publication date of this document.

[Boehm 1988]
Boehm, Barry W. & Papaccio, Philip N. "Understanding and Controlling Software Costs." *IEEE Transactions on Software Engineering 14*, 10 (October 1988): 1462-1477.

[CERT Secure Coding Standards 2011]
CERT Secure Coding Standards. https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards (2011).

[Christie 1996]
Christie, Alan M., Levine, Linda, Morris, Edwin J., Zubrow, David, Belton, Teresa (Nolan, Norton and Co.), Proctor, Larry (Nolan, Norton and Co.), Cordelle, Denis (Cap Gemini Segoti), Ferotin, Jean-Eloi (Cap Gemini Segoti), Solvay, Jean-Philippe (Cap Gemini Segoti), & Segoti, Jean-Philippe (Cap Gemini Segoti). *Software Process Automation: Experiences from the Trenches* (CMU/SEI-96-TR-013). Software Engineering Institute, Carnegie Mellon University, 1996.

[CMMI Product Team 2010]
CMMI Product Team. *CMMI® for Acquisition, Version 1.3* (CMU/SEI-2010-TR-032). Software Engineering Institute, Carnegie Mellon University, 2010.

[Humphrey 2009]
Humphrey, Watts. *The Watts New Collection: Columns by the SEI's Watts Humphrey* (CMU/SEI-2009-SR-024). Software Engineering Institute, Carnegie Mellon University, 2009.

[LINFO.org 2005]
"Peer Review Definition." http://www.linfo.org/peer_review.html

[Seacord 2010]
Seacord, Robert, Dormann, Will, McCurley, James, Miller, Philip, Stoddard, Robert W., Svoboda, David, & Welch, Jefferson. *Source Code Analysis Laboratory (SCALe) for Energy Delivery Systems* (CMU/SEI-2010-TR-021). Software Engineering Institute, Carnegie Mellon University, 2010.

[Wiegers 2001]
Wiegers, Karl E. "Inspecting Requirements." StickyMinds.com Weekly Column. http://www.stickyminds.com (2001).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	December 2013	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Using Software Development Tools and Practices in Acquisition	FA8721-05-C-0003

6. AUTHOR(S)

Harry Levinson and Richard Librizzi

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2013-TN-017

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Acquiring software-reliant systems from an external resource can be time consuming, costly, and often unreliable. During independent technical assessments and customer engagements, the Soft-ware Engineering Institute observed many contractors who are not utilizing mature, readily available software development tools when creating code for government programs. These tools include static analysis, test automation, and peer review techniques. In addition to the aforementioned tools being important for software developers, they offer significant insight and confidence to customers who acquire software products.

There are many benefits from making these tools and practices integral to software development and acquisition processes, including:

- reduced risk: programs deliver more predictably
- improved customer satisfaction: products developed experience fewer field defects
- lower cost of ownership: programs experience lower life-cycle maintenance costs when de-ployed operationally

This technical note provides an introduction to key automation and analysis techniques, the use of which the authors contend will benefit motivated acquirers and developers.

14. SUBJECT TERMS	15. NUMBER OF PAGES
software acquisition, software development tools, software development practices, static analysis, automation, peer review, software quality	30

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102