LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-742405

Embedded Software Quality Engineering (eSQE), A New Framework for Software Quality Assurance

G. M. Pope

November 28, 2017

Better Software East 2018
Orlando, FL, United States
November 4, 2018 through November 9, 2018

Disclaimer

Embedded Software Quality Engineering (eSQE)

A New Framework for Software Quality Asssurance

By Gregory Pope CSQE

Lawrence Livermore National Laboratory, November 22, 2017

LLNL-CONF-742405

Auspices

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Table of Contents

Table of Figures

Table of Tables

Executive Summary

Software development methodologies have experienced continuous improvement over the past fifty years utilizing advanced languages, increased collaboration, leveraging of automated tools and processes. Increasingly software is finding its way into regulated industries as well as proliferating into every aspect of daily living. However, the software industry is still highly dependent on auditing and assessments to enforce compliance in regulated industries and assuring adherence to quality standards. The auditing and assessment approaches used for contemporary software development have not substantially changed for over a century. This paper discusses the liabilities of software audits and assessments as a motivator for improved software compliance and quality. This paper presents a novel approach for assuring compliance to software standards and quality of software not solely dependent on audits and assessments. The innovative approach is implemented by embedding software specialists into development groups to help developers be more productive and automate compliance to regulatory and quality standards.

The Problem: Assuring the Compliance and Quality of Software.

In regulated industries such as:

1. Transportation,
2. Finance,
3. Medical,
4. Energy,
5. Communications, and
6. Defense

compliance of the software contractor to governing standards is typically mandatory.

Contemporary governing standards often apply to software as well as hardware components of a system being produced. The purpose of a governing standard for software is to assure the processes used to build the software produce the desired emergent properties in the software. Examples of desired emergent properties are:

1. functionality,
2. reliability,
3. portability
4. scalability
5. flexibility
6. maintainability
7. safety, and
8. security.

Combinations of desired emergent properties are commonly combined and called “quality”. Governing standards therefore provide requirements for software contractors to obtain the customer’s desired emergent properties. These governing software standards are ideally written by committees made up of experienced experts in the software development field. Creating a great a standard however, does not assure that it will be followed or interpreted as intended by its users. Assurance that software contractors are complying with the requirements of governing standards traditionally entails internal and external audits and/or assessments[1] of the software development activities being performed.

The audit or assessment process for software has several challenges that make determining the contractor’s level of compliance to a governing standard challenging. The auditing and assessment process may even produce negative consequences that are in opposition to the primary purpose of the audit or assessment. For instance, a poorly executed audit or assessment can make safety software less safe or reduce the level of software quality. Key to making the audit or assessment effective are the auditors and assessors themselves. One of the most important attributes of software auditor or assessor is the requirement to exercise professional judgement. It is near impossible for a governing standard to explicitly cover requirements that directly apply to every possible situation that can arise on a software project. The auditor must be experienced enough in software development themselves to be able to apply professional judgement in these inevitable situations. For instance, determining that a contractor’s software development activity is somewhat different but equivalent to or better than the governing standard’s requirements. A problem can arise however when the software auditors or assessors are not experienced software developers themselves. Software auditors and assessors without a software development background may not be able to apply professional judgement as it pertains to software or be overly dependent on documentation generated from the software development groups to understand how processes are being implemented. If the documentation is incorrect so might the understanding of the non-software experienced auditors.

The traditional software oversight approach of auditing and assessing is seldom regarded as “value added” by most software developers. Writing software process documentation so outsiders can understand software activities is not directly related to product quality, safety, or security. At best, this type of SQA is considered a necessary evil of doing business, at its worst it detracts from developer time needed to add features and improve the software product quality. It is relatively easy for software process descriptions to portray an excellent sounding (but imaginary) software engineering process that is not fully institutionalized nor followed. This paper asks: should the software industry continue to rely on this legacy auditing process to assure the privacy of our financial and medical records, safety of transportation and weapons, and the security of our personal data? Organized auditing had its roots at the beginning of the industrial revolution in 1840 before there was software.[2] This paper suggests that there is perhaps a better way to ensure the desired emergent properties of software and its compliance to governing standards in regulated industries.

Traditional SQA Approaches Have Not Kept Pace

Software development processes have changed substantially in the last 50 years, mostly for the better. Whereas in 1960 a large specialized software application would have been measured in thousands of lines of code, today we depend daily on tens of millions of lines of code in our cars, smart phones, satellites, and financial institutions. Over this period, to cope with growing demand the software industry has seen, tools and techniques have evolved such as Top Down Design and Structured Programming, Higher Level Languages, Object Oriented Languages, Static and Dynamic Code Analyzers, Automated Testing Frameworks, Continuous Integration to name a few. Software development processes have evolved from Ad Hoc, to Waterfall, to the V model, to Incremental and Iterative, to Rational Unified Process, to Extreme Programming, to Agile, to Scaled Agile, Disciplined Agile, to DevOps. However, the traditional software quality assurance approach for compliance and oversight in regulated industries has basically not changed much since 1960. In fact, it mimics the techniques used for hardware in 1840.

Contemporary software quality assurance traditionally uses one of the two following approaches; the "audit and punish approach", or the "call the software test group SQA" approach. Both have several drawbacks.

Problems with the "Audit and Punish" SQA approach

The audit and punish approach is the traditional approach to auditing software projects. When conducted by software practitioners it can provide some value. However, if the auditors are not accomplished software practitioners the process seldom goes well nor provides much value. Real-world examples of what has gone wrong with the "audit and punish" approach conducted by non- software practitioners are listed below:

1. **Cultural Impedance Mismatch:** Auditors who are also experienced software developers can communicate with the software developers they are auditing more effectively. However non-developer experienced auditors may have difficulties in understanding the relevance of the information they receive from those being audited. For instance, during an audit a non-software experienced auditor asks the developer lead being audited to be shown the SRD. The developer lead responds by asking what an SRD is. The auditor says it stands for a Software Requirements Description and the auditor is expecting a thick paper document. The software team leader responds by saying that all software requirements are tracked in the Jira tracker tool. Each requirement has a unique identification number, is prioritized, contains a user story, is linked to a test case, is assigned to a developer, and is allocated to a development sprint. The auditor marks down a finding because the software group does not have an "SRD".

2. **Rapid Obsolesce:** The software industry is constantly producing better tools and techniques for developing software. Software process documentation often lags being updated while the software process may be changing and improving rapidly. Example, the auditor reads the software development process description in a document which describes the software continuous integration process uses the Hudson tool. On interviewing the software team, it is determined that the team uses the Jenkins tool to support continuous integration. The software team explains that Hudson was renamed Jenkins by the vendor. The auditor creates a finding that Hudson tool is not being used to support continuous integration.

3. **Misunderstanding:** The software auditor asks the developer lead to see the sole source justification for choosing C++ as the software development language for the audited project. The developer lead asks what a sole source justification is. The auditor explains that the project needed to solicit a competitive bid and seek proposals from all compiler companies before choosing C++ as the source language. The auditor asks why Fortran, COBOL, or Java were not considered for the project? Why is there no written justification and analysis for why C++ is superior to the other languages? Only when there is sole source justification can the competitive bidding process be omitted for compiler vendors. The developers explain that C++ is an industry standard, that millions of lines of previous code were written in C++, that the developers are experienced in using C++, the C++ compiler is maintained by the company and widely used, the language is Object Oriented, and that it is relatively easy to hire new employees who have recent C++ language experience. The auditor creates a finding because the software group does not have a written sole source justification for using C++.

4. **Expense:** Verbose documentation which explains how the software process is accomplished primarily written for outsiders can add considerable expense, non-productive time, and cost to the software project. There are some important documents for most software projects to have, such as User Manuals and Installation Guides with tutorial examples for users, a concise software quality assurance plan showing how software process activities map to and meet compliance requirements, and architecture diagrams. Other documentation explaining code design, the requirements responsible for the code sections, and the links to tests created for the code sections can be added as comments to the source code. Keeping documentation with the source code is a good practice as it can be updated for new versions and remain physically close to the code it refers too. The risk however is a software auditor who is not able to use an IDE and read the source code could create a finding that the code design is not adequately documented, when in fact it is.

5. **Visibility:** Most of the key indicators of software quality attributes during development are not conspicuous to non-developers. For a highly visual example, the Golden Gate Bridge under construction in 1935, shown in figure 1 below.

Figure 1. Golden Gate Bridge circa 1935

By studying the condition of the bridge in this photograph one might surmise that the bridge is over half-way completed. Even without civil engineering skills the observer could see that the towers are standing and supporting some of the roadway, more importantly it would not be wise to open the bridge to traffic just yet. Contrast the Golden Gate Bridge circa 1935 example to a contemporary government software project example, the opening of the US government's Affordable Health Care web site in going live on October 2013[3]. The software developers and testers who built this web site understood clearly the software was not ready for prime time; the web site had been tested with only 200-300 inside users with test data and froze before enrollment forms could be processed thus failing the test less than 30 days before going live. The website was targeted to help 32 million Americans get insurance. Despite knowing for certain the website could not handle even 200-300 users[4] it went live on October 1, 2013 and not surprisingly crashed. Bureaucrats and politicians were blind to the actual state of the software and claimed to have no idea the website would have problems. This example illustrates the point that it is much harder to look at a software development project and determine its level of completeness and adequacy without software development training and specialized tools. Software is mostly invisible to the casual observer which makes it substantially more difficult to audit than hardware. Figure 2 below illustrates a software project that is underway. Looking at a software development project does not reveal much about the completeness and adequacy of the software being built, to determine the level of quality and completeness, specific skills and tools are needed. An auditor not equipped with these skills and tools would be easy to bamboozle with Power Point slides and hand waving.

Figure 2. Software Development Project

6. **Losada Ratio:** Audits and Assessments can harm the software development team because they ignore the Losada ratio[5]. The idea of this ratio is that more praise than criticism is beneficial in motivating teams. A recent Harvard Business Review article suggested an ideal ratio of six positive comments to each negative comment[6]. Dr. Marcial Losata in his original 2005 paper claimed three to one was a minimum[7]. Most studies agree that criticism is important to give, but also agree that only giving criticism will cause the team's performance to drop. Unfortunately audits and assessment reports tend to focus mostly on the negatives. One reason for this trend is that the standards to which the auditors use for compliance are probably already praise worthy. They are intended to be the best practices for the area being assessed. But auditors do not tend to praise teams for being compliant; they do however criticize teams when the team is not complaint. Praise only comes from exceeding the standards, which in theory should be very hard to do if the standards is already a high one. As a result, the ratio of criticism to praise is usually the inverse (six criticisms to each praise) in an audit report. Therefore, feedback from most audits and assessments alone is not going to help build a better team. Ironically the audit or assessment process could accomplish the opposite result of what is intended.

7. **The Law of Phycological Reactance**: There is ample clinical evidence formally collected since 1966 by Jack and Sharon Brehms[8], that substantiate humans would much rather choose what to do than be told what to do. Anyone who has raised teenagers (or been a teenager) will probably relate to this law. According to the law of Phycological Reactance humans are inclined to not do something they are told they must do. Humans are self-motivated to do

things they believe are generally good. Most regulatory standards however tend to be “one size fits all” and so at times, depending on context, may require things that do not make sense to do. If the standard is perceived to not help make the software better it will be strongly resisted or bypassed altogether. An example would be to require the same level of testing and code review for avionics software as for research software. Fortunately, many standards do provide risk graded approaches, recognizing there are diverse types of software with different consequences of error and used in different contexts.

8. **The Heisenberg Uncertainty Principle:** Introduced in 1927 by Werner Heisenberg (shown below left in figure 3, not to be confused with Walter “Heisenberg” White of Breaking Bad below right) states the more precisely the position of some particle is determined, the less precisely its momentum can be known, and vice versa[9].

Figure 3. The Heisenberg’s

This principle is also true for software. For example, one of the techniques used to measure test coverage is dynamic code analysis. To measure the test coverage additional code is inserted into the code being measured. The inserted code simply keeps track of which parts of the code under test are executed by the tests. However, the code used to instrument the code under test takes some time to execute and will slow down the code under tests performance when compared to same code without instrumentation. While getting an accurate measurement of code coverage the performance is decreased and so an accurate measurement of code performance cannot be made simultaneously. Likewise, if the code is executed without instrumentation the execution speed of the code can be measured but not the test coverage. For software audits and assessments, the Heisenberg Uncertainty principle also applies. The team being audited or assessed may perform differently when being “measured”. They may engage in tasks that they normally skip to impress the auditors, or they may forget to do tasks they normally do when not under the scrutiny of an auditors. They may also say things that they think the auditors want to hear, but which are not in fact completely true. So, to some extent

the auditors will be challenged to determine how much distortion to the software development process is being created by the audit itself.

In summary, because of the factors listed above an excellent software development process could be audited and found to:

1. Not have requirements
2. Not use tools
3. Be using the wrong source language
4. Be undocumented
5. Be 99% complete
6. Have a demoralized staff, be disrespectful to auditors
7. Have a rebellious attitude
8. Not able to measure progress[10]

How does an organization know if they are experiencing Audit and Punish syndrome? Below in table 1 are statements that may indicate Audit and Punish Syndrome is alive and well in your organization:

A software developer saying ….	” We barely have time to finish the coding and a day or two of test, we do not have time to document anything or talk to SQA auditors”
A software developer saying ….	” Really, an SQA audit, let’s just let them talk to Mark again and show them our Power Points, he really snowed them last time”.
A software project leader saying ….	“Make sure to keep the software quality engineer (SQE) out of our building so we can proceed with our work”
A software auditor saying….	“I do not need any knowledge of software engineering to do my software auditing job; I worked for 20 years in law enforcement (or substitute some other unrelated field) so I am qualified”.

Table 1. Evidence of Audit and Punish Syndrome in Your Organization

Problems with “Calling Testing SQA” Approach

In Silicon Valley (Santa Clara), United States, Silicon Glen (Linlithgow), Scotland, Start Up Delta, (Amsterdam) Netherlands, Silicon Plateau (Bangalore), India, Silicon Wadi (Tel Aviv), Israel, and other geographical regions that are centers for software development, SQA has become synonymous with software testing. Perhaps the SQA group in your organization predominately does testing. Software testing is an extremely challenging field on par with software development and great testers are invaluable to their teams, however it is only one facet of software quality. Another term for Software Testers is Software Development Engineers in Test or SDETs. SDETs use their programming skills to create code for testing purposes. The Agile approach has also created the need for testers who can work within development environments (shift left) to test early and often on partially completed software. The approach of calling testers the SQE group and getting most of the bugs out of the code still does not assure the software product will do what the marketplace desires or provide a happy user experience, it also does not leverage defect prevention techniques. Shown below in figure 4 is a work breakdown pie chart for software development phases. Note that all types of testing are ideally 50% (System, Integration, and Unit Test) of the time spent on a software project. While focusing SQA on testing does support 50% of the project activities such as defect detection and removal, it does not exploit opportunities for defect prevention in the other 50% of the project.

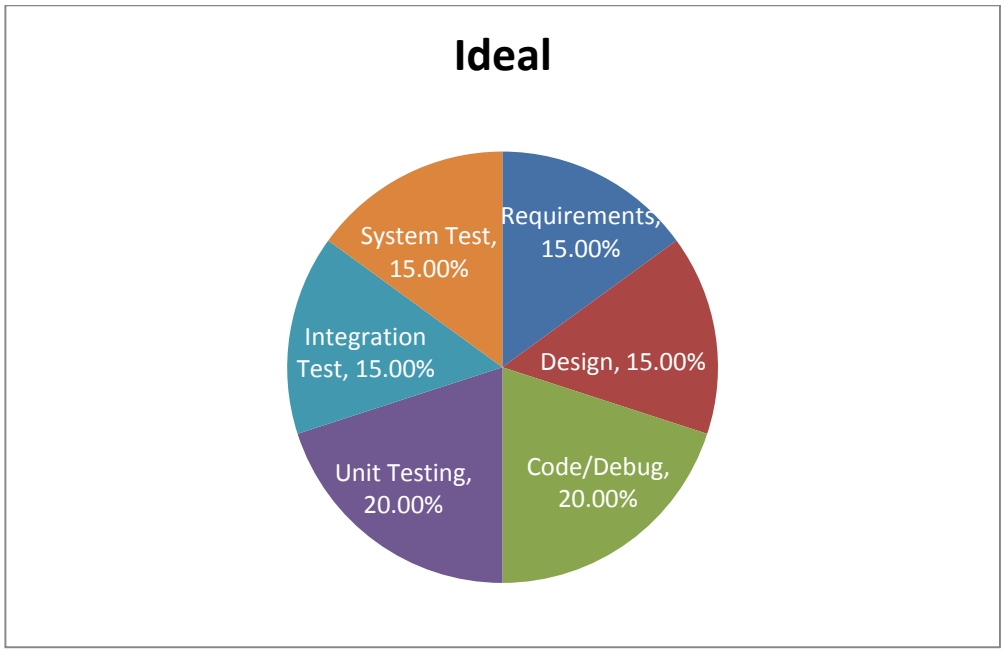

Figure 4. Ideal Software Development Phases

Software projects may not go as planned, especially those with longer schedules (over six months). The pie chart in figure 5 below indicates a common actual allocation of time that develops on a longer traditional V model project where testing winds up being only 25% of the project. This is because of added scope and optimistic assumptions about the difficulties of the coding/debugging phase. The release deadline stays fixed and development time expands into the test phase. As total testing time decreases the effectiveness of the "calling testing SQA" also decreases.

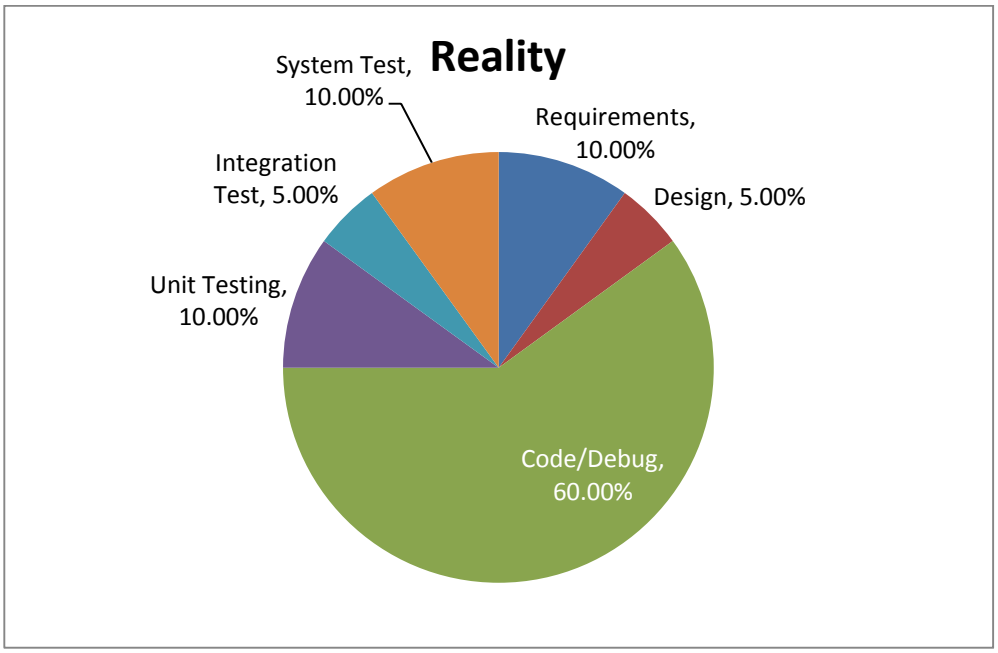

Figure 5. More Likely Reality of Software Development Phases

The development phase moving right as the end date for the software project remains fixed is also known as the accordion effect. This can happen because of optimistic development estimates. Experienced software developers understand that software estimates tend to be optimistic and compensate by multiplying by an expansion factor just as an experienced accordion player selects ample space (to the picture's right in Figure 6) to play their instrument.

Figure 5. The accordion effect

Yet another pitfall is this, as the testing group gains more of the developer's confidence and trust in being able to find bugs, error checking by developers can slack off under schedule pressure, the developers expecting testers to find more of the problems. This is sometimes called the "over the wall syndrome", figure 7.

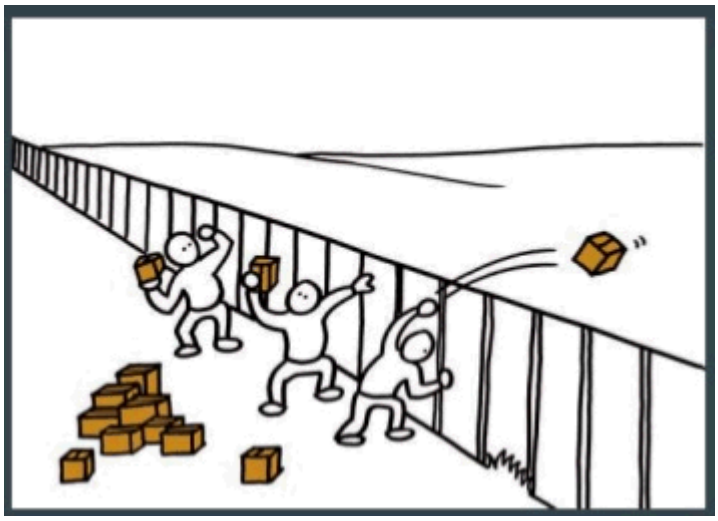

Figure 7. Over the Wall Syndrome

By the time the testers get their hands on the code it contains bugs that could have been found by the developers while debugging and running unit tests. Modern Integrated Development Environments (IDEs) are helpful for developers ease of compiling, linking, and debugging code. This immediacy of being able to compile for developers also creates the temptation to skip careful code desk checking and review by peer developers, depending heavily on the compiler to find problems instead. The downside is the compiler does not understand the user requirements, just the syntax and semantics of the source language. Agile teams consisting of testers and developers together with measuring acceleration (burn down rate of the sprint backlog) can help tune estimates during the project.

Agile approaches also tend to guard against the accordion and "over the wall" time allocation because of shorter (30 days or less) sprints and testing early and often. While software testing has come a long way in the last 30 years, with automated unit testing frameworks, GUI tools, simulators, continuous integration, model based testing, behavior based testing, exploratory testing[11], calling the testing group "SQA" is still focusing on defect detection rather than also taking advantage of defect prevention.

Depending primarily on great software testing for software quality is a known fallacy.[12] Quality is typically an attribute that cannot be tested into a software product. Quality or desired emergent properties must be part of each phase of the software development process. Software development tasks such as elicitation of requirements, use case development, architectural design, coding practices, defensive programming, performance, intuitiveness, safety, and security must be built into the software as it is developed, they cannot be added as an afterthought or bolted on later.

An Improved SQA Approach: Embedded Software Quality Engineers (eSQE's)

What if software quality assurance could be accomplished more effectively than the traditional "audit and punish" or "call testing SQA" approaches? An approach recognized as "value added" by the software developers? An approach that measurably removes defects while also preventing them? To accomplish this a new breed of software professional is required, an embedded Software Quality Engineer (eSQE).

In this paper, an embedded SQE refers to staff who have a skills, knowledge, and abilities which includes software testing and:

1. requirements elicitation and analysis,
2. software design,
3. coding in multiple languages,
4. automated build, test, and release,
5. configuration management,
6. issue tracking,
7. auditing and assessments,
8. review and inspection,
9. compliance to standards,

10. documentation and artifact generation,
11. static and dynamic code analysis,
12. development and target environment architectures,
13. emerging tool familiarity and piloting,
14. software industry practices and success rates.

Embedded SQEs have frequently been former software developers themselves; they understand experientially what it is like to be a software developer. Embedded SQEs share many similar skills with developers, but they apply these skills to support optimizing and automating the software development process instead of generating the primary code. Embedded SQEs may operate effectively in agile teams, traditional teams, hybrid teams, and in mission critical and regulated industries.

Embedded SQEs work as a valued part of a software development team alongside developers and testers. They focus on tools and process automation, testing automation, build and release, compliance, and general process improvements. They buffer the development groups from external SQA audits and assessments. Developers often have ideas and suggestions to improve their processes, but seldom have the time to implement them during a project. This is where the eSQEs add value, by implementing these suggestions or ideas with automated tools and improved processes. A skilled eSQE can not only automate processes to be faster and less error prone, but build in artifacts that automate compliance requirements.

While ratios of software testers to developers can run as high as one to one, ratios of eSQEs to developers are much lower. On the order of ten software developers and testers to one embedded SQE. ESQE's may operate exclusively within one large development team or be shared among multiple smaller teams.

Three Dimensional SQA

There are several traditional definitions of software quality. They typically focus on the software product meeting and exceeding customer expectations. Most definitions of software quality do not consider the quality of the developer's experience providing the software. This is one of the areas where embedded Software Quality differs from traditional quality. How can a software development team meet or exceed customer expectations if the process and tools they use to develop the software are not of high quality as well? All too often developers are challenged with unrealistic schedules, inadequate resources, disruptive management, and onerous oversight. Embedded SQA recognizes that the developer's experience must also be a high quality one. Embedded SQA embraces three dimensions of Software Quality:

1. The product exceeds customer expectations and has a risk appropriate defect rate
2. The user's experience with the product is a positive one
3. The software developer's experience developing the product is a positive one

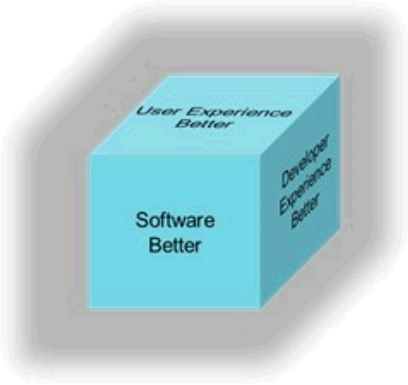

Figure 6. Three-Dimensional Software Quality

The eSQE focuses on obtaining and balancing all three dimensions of software quality. Happy developers develop better code[13]. There are many examples of successful companies in other industries that understand this basic principle; one of the most successful in the United States is Southwest Airlines. Since their founding in 1967 the airline has grown into the largest domestic passenger carrier in the United States. Accomplishing this in a very cut-throat industry that has seen many failures and mergers over this timeframe. One of the main principles of Southwest Airlines is to respect and treat their employees well who in turn treat their customers better. In order of importance, Southwest ranks employees first, customers second, and shareholders third. "We believe that if we treat our employees right, they will treat our customers right, and in turn that results in increased business and profits that make everyone happy," [14] Entrepreneur Rob Walling listed the nine things developers want more than money, leading the list was the work environment, specifically "it's critical to have buy-in to do things the right way, and not just the quick way. As one developer I talked to put it "Quality is as important as feature count and budget".[15]

Embedded Approach

The term embedded approach for software quality assurance had its genesis in journalism during the Vietnam conflict. Without the news censorship of previous wars, embedded journalists traveling with combat units could observe, film, and report on what was going on directly. The embedded concept required the journalists to acquire additional military training and equipment to travel with the fighting units, see figure 7.

Figure 7. Embedded News Reporting

Technology was basic on the battlefield in 1965, audio tape recorders and film cameras required four or five days of processing before they could be released. The result however of this embedded journalism was to get unfiltered information to the public, both the good and the bad. As a result, the public had the war brought into their living rooms on television nightly. The embedded journalist's reports were not always consistent with the official Pentagon or White House version of how things were going. Embedded journalism allowed the American public and Congress to make more informed decisions about the wisdom of continuing the conflict. This form of reporting has continued in the Serbian, Iraq, and Afghanistan conflicts taking advantage of newer technologies which permits reporting in high definition color via satellite links in near real time.

The amount of technology available for contemporary software development has also enabled the concept of eSQE's. Embedded SQE's develop a very accurate and detailed understanding of the development process being used because they are working within it. However, the eSQEs do much more than just provide reporting when required, on a routine basis they conduct several value-added functions for the development team which allows the developers to focus on development and the eSQE to focus on process improvement, automation, and compliance.

Supporting Agile Principles

Embedded SQEs support several Agile Principles[16], adding an eSQE role to a development team enhances the ability of the team to continuously improve techniques and tools, which supports many the Agile principles. Specifically:

Agile Principle 1 is to satisfy the customer with early and continuous delivery of valuable software. This requires roll out of tools and automation and the maintenance of these items which the eSQE can facilitate.

Agile Principle 5 is to build projects around motivated individuals. This is accomplished by the eSQE assisting developers with the support they need for process improvements, tool maintenance, build and

test automation, work flow automation and creation of compliance artifacts, and avocation of customer requirements.

Agile Principle 6 is to promote a sustainable development. Use of automation for code building and testing allows for early defect detection and removal. Upstream tracking of user requirements and tasking allows early assessment of resource needs and helps eliminate the need to work to overly optimistic schedules. Use of static analysis identifies structural issues early, dynamic analysis assures test coverage is optimized. The eSQE can be an important in maintaining this level of automation.

Agile Principle 9 is continuous attention to technical excellence and superior design. The eSQE is constantly on the lookout for new and improved tools and techniques and can pilot these innovations in the developer's environment before deciding on whether the team should commit to the tool or process.

Risk Grading

An important part of Software Quality is using the appropriate amount of software development process rigor depending on the software's consequences of failure. Software code that controls an aircraft system or nuclear power plant should be developed with greatest rigor since consequences of error could result in multiple fatalities. Software written to provide financial information should be done so with a good deal of rigor as large financial penalties could result. Scientific codes developed to support research should also have a moderate level of rigor to not waste other scientist's time or embarrass the institution publishing results. Codes developed using the least amount of rigor would be codes whose consequence of error is trivial.

The risk grading itself must be uniform and systematic, removing as much subjectivity as possible. A good risk grading method contains a series of questions for the eSQE and Developers to answer to get a risk score. The risk grading can be conducted using a web based or automated tool available to all software projects. The risk score level received after answering the questions translates to the amount of software development rigor that must be applied to the software development effort. The processes required are delineated in templates tailored to each risk level. This approach allows developers to be following regulating standards without having to understand all the nuances in the standards themselves. ESQE's can be instrumental in establishing a risk graded approach to software development and contributing to establishment of a risk grading tool and the associated templates for Software Quality Assurance Plans. An example risk grading tool in the form of a spreadsheet with instructions on use can be downloaded at the link in the end note.[17]

Software as an Ecosystem

Risk grading applies to compilers, libraries, feeder codes (including open source codes), and codes that may create and supply data to and support the primary code as shown in figure 8.

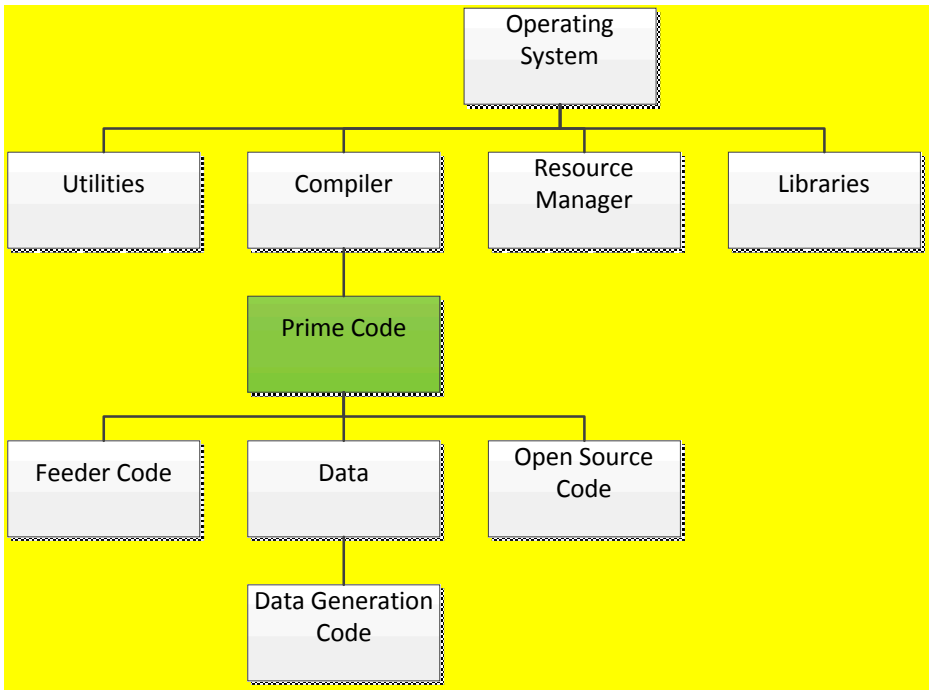

Figure 8. Software Ecosystem

Most contemporary software is written in high-level languages dependent on many levels of abstraction. The dependencies of the prime code quality on libraries and feeder codes must also be vetted against a risk grading process. A prime code that is of high quality that partially depends on a library or data that is derived from lower quality software may inherit their shortcomings if not mitigated. This applies to software that is acquired through open source, collaborations, or purchased from suppliers. When considering code security, the entire code ecosystem must be considered, not just the new code being written. One of the skills of an eSQE to investigate the pedigree of the support codes or dependencies on support codes with known vulnerabilities. Static analysis tools and security scanning tools can be integrated into the development and testing process to vet supporting codes in the ecosystem that surrounds the prime code. ESQE's can support development by installing, using, maintaining, triaging, and interpreting reports from these tools to help understand the risks involved in depending on other codes in the ecosystem. ESQE's can also investigate supporting open source code licensing requirements to be aware of any restrictions or obligations the open source license places on those that use or link to it.

Lifecycles to Frameworks

Over the last fifty years various software development techniques have gained and lost favor. Software found its way into military systems during the 1970's, the DoD was one of the first agencies to develop and enforce software standards. The DoD created the idea of a software lifecycle where software stages

cascaded in unidirectional series from initial requirements to design, to implementation (coding and debugging) to verification (integration and system testing), and finally after release and acceptance the maintenance phase. as shown in figure 9.

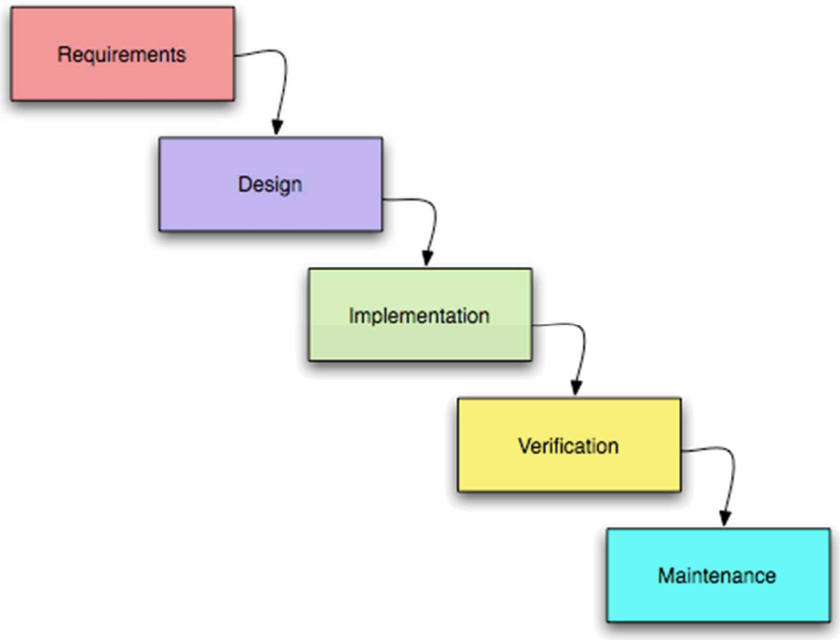

Figure 9. Original Waterfall Lifecycle circa 1970

The lifecycle prescribed by the DoD was called the “waterfall” and was either wisely or mistakenly widely adopted, depending on whose version of history is believed. The purpose of the waterfall lifecycle model was to detect erroneous requirements, design, code, and testing prior to the delivery of the software to the DoD. This was accomplished by having requirement reviews, design reviews and to a lesser extent code reviews and test reviews as the software was being developed. The DoD adopted the waterfall lifecycle model as a countermeasure to the sizable percentage (85%) of delivered software that did not work well. Payment to DoD software contractors was contingent on completion of the afore mentioned lifecycle reviews. After approval in a phase (for instance the requirements), the requirements could not easily be changed[18].

The waterfall assumed, incorrectly as it turns out, that most all software requirements are known at the beginning of the project and once these known requirements are frozen, will seldom change. The waterfall failed to support the human trait of hindsight being 20/20 and the advantageous characteristic of software being more malleable than hardware. On very simple software projects the waterfall lifecycle was tolerable, but as software projects grew in code size and complexity the waterfall changed to the V model, the double and triple V model, then the incremental V model, then the iterative model, the iterative incremental model, the Rational Unified Process (RUP), Extreme Programming, Agile, and DevOps. These refinements to the software development all took advantage of feedback from the following phase back to the previous phase and/or from one development iteration to the next. For

instance, additional insights gained from designing the software could be used to modify the requirements; insights gained from writing the code could be used to improve the design, insights gained from early product iterations could be used to improve the next iteration. The term lifecycle eventually faded away and was replaced by the term frameworks. Software lifecycles evolved to frameworks because software (unlike hardware) is less expensive to change and redistribute to users after the initial version is released.

The thinking underscored by the Agile Manifesto[19] in 1991 is that software requirements change frequently during software development and the software development process should welcome and accommodate change. The emphasis is on good code and not creating documentation. How can the traditional SQA function operate in an environment where things change rapidly? Any documentation created separate from the code is going to be obsolete hours after being created. If the SQA function depends on this documentation it will be continually out of date, multiple versions behind. Therefore, any SQA method that depends on up to date documentation is doomed to fail. The role of the eSQE is to develop automation and tools to create artifacts and records that are generated by performing the development activity, thus assuring these artifacts are up-to-date and accurate records of the software development activities that have taken place.

Contemporary frameworks should emphasize prevention and early discovery of errors. Automation should be adopted to speed up processes and to generate up to date records as a byproduct of doing the activity. An example of a contemporary framework is the agile-scrum framework shown in figure 10. Helping to tailor a framework for developing software and standing up tools to automate the process are areas where an eSQE can contribute to the development team. The eSQE should also attend industry conferences and symposiums to gain insights into the software development processes used by industry leaders and tools available from vendors to support these processes.

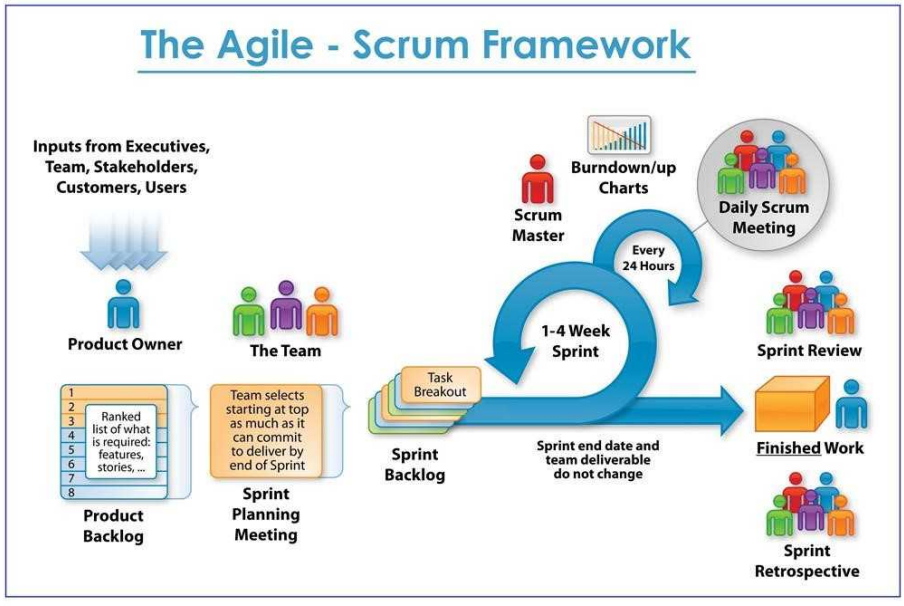

Figure 10. Agile – Scrum Framework[20]

Requirements and Issue Tracking

Requirement elicitation for software has evolved from simple verbal requests, to written documents, to tracking tools. Requirement tracking tools (usually containing a data base) allow for statement of the requirements (feature or function or user stories descriptions) and contain requirement meta data such as enumeration, prioritization, definition, dependencies, task assignment, build target, links to code and tests. Requirements may have a one to many relationships with tests or code sections. Likewise, a single test or code section may support multiple requirements. The data associated with each requirement can be queried and displayed in reports. Requirements may be grouped into a feature list for a release or (release backlog) grouped for a sprint on the path to the release (sprint backlog). Requirement definition information can be copied to source code comment fields and release notes accompanying a release. Using a tracking tool, changes to requirements and tests can be updated easier, because they are linked through the tracking tool, such as Jira[21]. When users inevitably require additional features, they may also be entered and tracked for inclusion in future releases. Contemporary requirement tools accommodate Agile concepts, supporting epics, user stories, feature lists for releases and sprints, and Kanban[22] boards to manage tasks in scrums. A requirement management tool may be the same tool as used for issue tracking.

Issue tracking assures that bugs discovered by developers, testers, or users are fixed and retested. This is especially important if the developers, testers, and users are not collocated. Issue tracker reports usually require data such as prioritization, issue description, platform configurations, screen shots, memory maps, feature(s) effected, dependencies, repeatability, task assignment for repair, and status for each issue (e.g. reported, assigned, fixed, tested, closed). Prioritizing and assigning who will fix and retest each issue can be accomplished during project staff meetings, scrums, or in a dedicated meeting sometimes called the configuration control board (CCB). Issue trackers may be used to track issues in the code, issues found in code reviews, or issues in the user documentation, such as the user guide. The issue tracking tool may be queried to produce aging reports (amount of time an issue has been open without being fixed) and other useful information. Contemporary issue tracking tools can receive information directly from an application being used in the field if the application has on board issue reporting capability. A problem report can be generated from the user's application which automatically captures the platform and environment information helpful to developers in recreating a repeatable issue in a controlled environment. Information about the issues can be sent from the application in the field to the issue tracking tool.

Both requirements management and issue tracking tools accommodate rapid software change as well as produce queries and automated reports that are useful as compliance records. As technology advances, so do the capabilities of requirements and issue tracking tools, both in terms of new versions of existing tools and newer and more capable tools. The eSQE researches advances in tools and can initially set up requirements and issue tracking tools for a project team, tune the tools for the project, upgrade tool versions, or switchover to newer tools. Installation and maintenance of the requirements and issue tracking tools (or a combined tool) by the eSQE relieves developers working on the prime code of this burden. Further research on requirements tools[23] and Issue tracking tools[24] can be found in the endnotes.

Design

Software design has always been a challenge for software development projects. In the days prior to wide use of higher level languages like FORTRAN and COBOL, assemble language programming usually required a distinct design phase and a tool such as a flow chart was valuable. Flow charts were drawn using a template (see figure 11) to trace out common shapes onto paper.

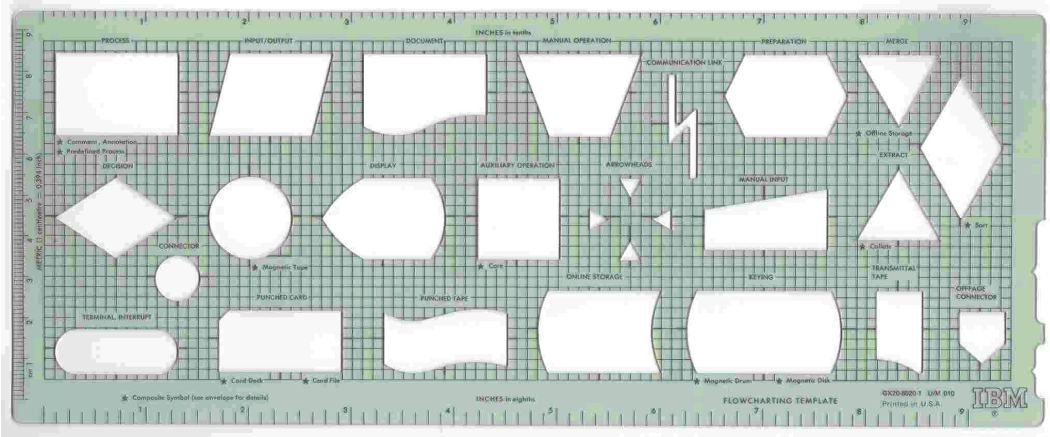

Figure 11. Vintage IBM Flow Chart Template

The assembler code structure seldom bore a physical resemblance to the problem being solved. Concepts such as top-down design[25] , structured programming[26], modularity[27], coupling[28], cohesion[29] and simplified code cylomatic complexity[30] were developed to guide developers in designing software in assembler language. However, with the development of higher level languages, the need for a separate design phase has greatly diminished. Structured programming concepts, encapsulation[31], and inheritance[32] is built into modern languages. Contemporary Object Orient[33] languages allow for much of the design function to be accomplished using the source code, for instance in defining classes[34]. The source code physically looks much more like the problem space, eliminating the requirement for flow charts. Except for high-level architectural design, most software efforts today can and often do proceed from requirements directly to code. In contemporary software development, the code itself can be viewed as the detailed design, an abstraction of an executing program. Analogous to a blueprint being an abstraction of a physical structure. Relying on the code for the detailed design eliminates the version control issue discussed in the following paragraph.

Even when the design step was a necessity in the past, design documentation version control was problematic. The initial design was done using the design tool (e.g. flow chart) and then from the design the code would be manually written. The problem was as the code was evolving and being repaired after test, the design (flow chart) would rarely also get updated, thus causing the documented design to become obsolete. There are several model-based tools which can generate source code automatically

from modeling languages (a.k.a. forward engineering) such as Unified Design Language (UML)[35]. There are also tools that allow source code to be automatically converted into UML or flow charts (reverse engineering). There are even a few tools that claim to do both forward and reverse engineering as well as keeping multiple derived artifacts consistent (round trip engineering) [36]. However, on closer inspection the code that is generated is a usually a class definition or other high-level construct and methods or procedures are required to be coded in the manual fashion.

The eSQE is available to research contemporary design tools to see if they provide advantages over directly coding from requirements in a high-level language, to acquire and pilot these tools, and to see how they work on the developer's platforms with the actual source code. If most source code has already been developed without the benefit of forward engineering design tools, switching over to take advantage of them is requires substantial efforts, however if starting with all new code the design tool may be advantageous. The reverse engineering tools however can be used with any existing code bases to provide "as built" design documentation if required for maintenance or compliance purposes. See the end notes for a list of code generation tools[37] and an example code to flowchart/UML tool[38].

Static Analysis

Static analysis is accomplished using specialized tools which parse and process source code to find common structural[39] errors that compilers may overlook, for instance not initializing a variable before use, dereferencing a null or stale pointer, overrunning an array, unreachable code, or using an unverified input variable to index an array. Sophisticated analyzers can find problems caused between multiple functions and provide trace-backs. For instance, a variable not initialed in one function but used in another. Structural errors also introduce cyber vulnerabilities, so by detection and repairing these structural errors cyber vulnerabilities are substantially reduced. Static analyzers can also look for cloned code[40], code that has been cut and pasted to multiple locations in the software. Cloned code becomes risky when an error is detected in that code snippet. The challenge is finding all the other cloned snippets and repairing them as well, or better yet creating a class or method that replaces the cloned code so one repair handles all instances of the code. The static analysis tools can find most structural code problems much faster and cheaper than testing may find them, therefore the tools can add value to the software project. The most common static analyzers are for Java, C, C++, and C#. Tools such as Klocwork[41], Coverity[42], Clang[43], ROSE[44], and Parasoft[45] are such tools. A more complete list can be found at the in the endnote below.[46]

Static analyzers are not without their short comings however. Firstly, to be static analyzed the code must be built in the static analyzer's tool environment. This may take some time to replicate the build environment. Secondly there are false alarms or multiple findings that are related to a single issue. Thirdly there is the challenge of first use. If a code has never been static analyzed chances are there will be many findings, some very important, some not. The first use case requires a process known as "triaging" performed by the eSQE to not overwhelm developers with too much information. The triage process determines which findings are important for the developers to know about and need attention,

and which can be deferred. The triage process may also weed out false alarms and repetitive findings that have a single root cause. The eSQE through skillful filtering can omit findings in library codes or open source codes that are not written by the supported development group. These findings can be referred to suppliers or open source forums.

An experienced eSQE may go beyond triaging the tool's static analyzer reports and repair the more obvious issues, then rebuild and retest the code. Repairs that are not obvious or higher risk are best accomplished by the code's author. The eSQE may also write custom checkers to add to the set of checkers that come with the tool. A custom checker may look for specific issues that are unique to a platform or to language extensions. As with other tools, the eSQE is researching developments in the static analysis tool market, updating versions of the existing tool to take advantage of new features, and maintaining the tool for develop use. The eSQE can also make the static analysis tool available via client versions to the developers as they write their code as well as write scripts that allow automated static analysis and report generation on a periodic basis as part of automated testing.

Software Code Review

Conducting static analysis is a cost-effective way of finding structural code issues, however it does not find functional issues, such as requirements that are missing from the code, requirements that are not implemented correctly in the code, code sections that do not have corresponding requirements, standardization issues, design optimization, and other issues that peers are likely to find. Code reviews can vary in size and complexity. The simplest is the desk check, having a second developer review the source code of the author. The next step up in rigor is the peer review where multiple developers review the source code. The highest step up in rigor is a structured peer review that adds users or stakeholders and where specific roles are assigned to those participating in the review, such as moderator, author, scribe, reader, reviewer. One such structured code review is the Fagan inspection.[47]

In the past code reviews were often held as face to face meetings. Face to face meetings created the need to the schedule the review staff at the same time of day and to meet in a conference room. The challenge becomes finding a time when all the reviewers are available and in some cases even finding an available conference room can be a challenge. Contemporary code review collaboration tools have eliminated the need to meet at a certain time and require a conference room. Code needing review can be passed to reviewers along with any requirements or supporting documents that may be useful to reviewers. Each reviewer can conduct their review at a time convenient for them annotate the source code with comments and return them to the author. A list of contemporary code review tools can be found using the end note[48].

There are some guidelines for code reviews that should be followed, the most common is to limit the amount of code being reviewed on any one review to under 300 lines. Also, to allow sufficient time to conduct a review. The code review ideally is an egoless event, where the focus is finding issues, not proving correctness. The focus is on the correctness of the code to the requirements, not the personalities of the authors and reviewers. Egoless reviews are easier when management is not

involved. The findings from the code review using a collaborative tool should be archived, as well as archiving the evidence of successful repair and retest.

The embedded SQE can support the code review process in many ways. Researching and rolling out a collaborate tool, setting up the tool, maintaining the tool, assuring the artifacts generated from conducting the code review are archived.

Design by Googling

One of the conveniences of the internet is googling web pages to search for code snippets that may already implement a function that needs to be coded. Once the developer locates a code snippet or function that appears to accomplish what is desired it can be cut and pasted into the code being developed. This is a variation on the practice of looking for code snippets and examples in text books, and retyping the text book example into the code being developed. While there are advantages to not reinventing the wheel while coding, there are some risks in employing screen cut and pasting that developers must mitigate.

There is no global verification authority that assures all code snippets on the internet are correct or correct for the purpose the using developer has in mind. Some websites do have comment sections that may help reduce risk by additional contributors refining the code and confirming successful use. In any case, when using code from the internet do not assume it is correct, be sure to test it carefully for the way it is intended to be used in your code. Be sure to desk check the code to look for extraneous code or data that may be a back door or virus. The same holds true for copying code from a text book, however there are two additional risks with text book code. One is in retyping the code an error can be added, and secondly text book example code is notorious for being incorrect. Apparently, text book editors are not software developers. In any case the same rules apply, test the text book code carefully for the intended purpose. Embedded SQE's can be helpful in searching for and testing web code snippets and text book code.

Automated Build / Test / Release

Embedded SQEs are heavily involved with or in some cases manage the automated build / test / release processes for developers. This supportive role requires the embedded SQE to have knowledge of common scripting languages (e.g. Perl, Python, Bash) and build languages (e.g. autoconf[49], CMake[50], Apache Ant[51]) and tools (e.g. Jenkins[52], Bamboo[53], Cruise Control[54]), also to determine the cause of build and test errors and take actions to resolve them. Frequently build problems are caused by things other than a problem with the source code, for instance running out of memory on the host computer, permissions not set correctly in the operating system, paths of support libraries no longer being valid, versions of support libraries no longer being valid, a system time out occurring before the build is complete, or compatibility problems between the compiler version and the libraries version. Likewise

test failures can occur for reasons not related to source code problems. Common reasons for test failures are the wrong versions of tests are run, permissions set incorrectly, time out before tests have completed, insufficient memory allocated, and expected result's baseline locations incorrect, wrong math libraries used causing rounding or truncation errors, compiler optimizations.

The embedded SQE may be a maintainer of the testing frameworks used by the developers, adding features and capabilities for them to automate processes, such as code coverage, static analysis, profilers, smoke tests, continuous integration, reporting tools, e-mail lists, access, permissions, baseline updates, and developer desired features. As platforms change the test framework may also be impacted and require modification. For codes that must support numerous platform types and variations, the embedded SQE may support a platform testing facility consisting of the required supported platform types or virtualized platforms. The build requirements for each platform type may consist of numerous supported compilers, libraries, and versions.

The embedded SQE also researches build and test tools and pilots the promising ones to take advantage of new features and tools. There are many excellent tools to support the compile, build, test, release process. A full list is provided in this endnote.[55] The embedded SQE may also design system level tests or work with users and domain experts to create suites of regression tests. Embedded SQEs may also be tasked to develop or incorporate new types of test reporting tools. A common test reporting tool is one that can analyze test reports, finding failures or patterns of failures, or provide visual display of test results which may make test failures more pronounced using pattern recognition.

Multi-Level Test Coverage

Dynamic analyzers can be used to determine the test coverage achieved when tests are run on the executing source code. Use of dynamic analyzers for code coverage is done periodically since it is an invasive technology which alters how the code executes and will slow it down. Code coverage can be determined at the statement or function level. Code coverage can also be accomplished without special tools at higher levels, such as feature coverage, by building into the source code conditional code consisting of counters that determine how many times a feature has been exercised. While not as fine grained as statement or function coverage achieved with dynamic analysis tools, built in feature coverage does not alter the way the code operates nor slow code execution down. The embedded SQE may work with developers to add the "on-board" source code to determine feature coverage, or build and instrument the source code with a dynamic analyzer tool to measure function and statement coverage.

Determining the test coverage helps in optimizing test suites to assure they are not missing code sections or redundantly testing others. Dynamic analyzers are often challenging to operate correctly, subject to all the challenges of building code and not being compatible with certain compilers or compiler features. The embedded SQE can trouble shoot problems with dynamic analyzers. The eSQE can also research and pilot dynamic analysis tools such as Lcov[56], Gcov[57], and the Intel Coverage Tool[58] or any number of tools referenced by the end note below.[59]

Dynamic analysis tools can also be used to find issues missed by tests and not findable using static analysis. These issues would include race conditions, memory management, memory leaks, finding randomly occurring problems. Examples of dynamic analysis tools that can find memory problems are Rational Purify[60] and Valgrind[61]. A more complete list of profiler tools can be found following this endnote.[62] The eSQE can help research and roll out tools which help with profiling and memory management.

Test Report Data Mining

Data mining usually requires the addition of a relational data base to the test report generation process. Test reports generated by testing frameworks themselves may be quite verbose and include thousands of tests. Determining the correctness of the test reports can become a sizable job and automated with data base queries. Missing one failed test among thousands of passed tests is easy to do without the ability to search and query electronic records. In addition to immediate queries, test results over time can be archived in a relational data base and queried ex post facto to produce useful trending information about patterns of test failures or correlations between test failures over time. These patterns could be related to platform types, modes, compilers, libraries, versions, time-of-day, values drifting, test dependencies, order of tests, or other information that may help developers determine the sources of failures. As an actual example, a set of tests that fail at the same time of day for many days may indicate the server they are running on is rebooting daily to reset memory leaks. The embedded SQE may assist in the incorporation of a relational data base for archiving and querying test reports and mine the data base looking for important trends in archived test reports and developing useful trend reports.

Profiling and Performance Testing

Profiling tools are a type of dynamic analyzer that require code building skills to use. The embedded SQE can research and pilot promising tools that work on the developer's platforms and with the developer's code. Profiling can also be used to determine where best to add parallelism to the code, find problems with multi-processing, and multi-threading. An example of profiler or performance measurement tools are Tau[63] and as shown in figure 12 Intel VTune[64].

Embedded SQEs may set up and run profiling tools which indicate where the code is spending its time. This information may be useful in determining places to optimize code performance or spotting errors in code based on slower than expected performance (get the right answer but it takes longer than expected). Performance testing in the world of large parallel computers has become a necessity. Super computers now consist of millions of cores and several types of processors (heterogeneous architectures) such as CPUs[65] and GPUs[66]. A more complete list of profiler tools can be found at this endnote[67]

Figure 12. Display Using Intel VTune to look for Hot Spots on a Supercomputer

Performance testing or load testing is also thought of as critical for web based applications that service large numbers of users. Tools exist to provide simulated loads of users interacting with web pages to measure response times[68], capacity[69], load balancing[70], and stress testing[71]. Tools that can be used for this purpose are found following the URL at the end note.[72]

Software Configuration Management

The embedded SQE must have knowledge of how to use developer SCM tools to build, test, conduct static and dynamic analysis, and repair codes. The embedded SQE may also be involved with adding capabilities to an SCM tool. One such method is by using “hooks” contained in the SCM tool. The embedded SQE can code scripts that perform various tasks automatically for the developers. For instance, on code commit, run a set of smoke tests, or before allowing code commit, assure a code review has been completed. Researching new and promising SCM tools and piloting them in the development environment is an ongoing task for the embedded SQE. From time to time new and better SCM tools emerge and after a successful trial period the embedded SQEs can perform the migration tasks from the old tool to the new one. Excellent software configuration management tools are

available for software development such as GIT[73], Subversion[74], and Mercurial[75]. A full list of code repository tools is at the endnote below[76].

Disaster Recovery

It is important to assure that the source code, supporting development codes, and libraries are backed up on a nightly basis and periodically backed up to a different geographical location. Often the IT organization does this as a matter of routine policy, however the embedded SQE can check to make sure that the archives and repositories used for development are covered in the back up plans. This also includes the provision for offsite storage. In addition to assuring that the backup is being done both locally and offsite the embedded SQE should assure that a periodic retrieval is performed to assure the ability to restore the project files can be accomplished should an actual emergency occur.

Complexity of Compliance

In regulated industries, such as DoD, Medical, Transportation, Aerospace, and Nuclear the embedded SQE has the additional challenge to assure that the software being developed is fully compliant with applicable standards and orders. It would be unrealistic to think that each developer could be deeply knowledgeable about applicable standards and orders and keep up with revisions to them, and in some cases, decipher the legalese used to write them. The embedded SQE focuses on the flow down of the compliance information and assures that the software development process is fully compliant with these requirements by creating compliance tables listing requirements and how each is met by the software development process being used. This table is normally part of the Software Quality Assurance Plan. The skilled embedded SQE will automate as much of the compliance requirements as possible so that proof of compliance (records) are generated automatically by performing the action to the greatest extent possible. For instance, proof of running and passing tests is contained in archived test reports automatically generated by the testing automation.

Wikis for Records and Artifacts

The embedded SQE working with code team leads also generates compliance documents that show how the flowed down requirements are being implemented by the code team. Documents are typically the Software Quality Assurance Plan, which may include Software Configuration Management, Testing, and Disaster Recovery or these topics may be in separate documents for larger code teams. These compliance documents are reviewed, signed, and kept under configuration management. The challenge with these documents has traditionally been they become obsolete over time as process improvements and updated tools are rolled out to support the software development process. To mitigate this compliance documents obsolescence issue, compliance document versions can also be kept in electronic wiki form and modified in near real time as these improvements are made. The electronic wikis pages

that have changed are periodically transformed into documents automatically and reviewed, signed, approved, and become the latest version of the compliance document. Setting up and maintenance of the wiki tool is a task for the embedded SQE. Examples of wiki tools[77]are found in the end notes.

Audits and Assessments

Utilizing embedded SQEs may not eliminate the need for performing periodic external audits in regulated industries or larger companies. Embedded SQEs can act as a buffer between external auditors and the development teams being audited. Embedded SQE's can demonstrate to the external auditors the compliance flow down, streamline communications, provide documentation useful to external auditors, and answer most auditor questions, all without taking up a large block of time from developers. By having external software auditors interface primarily with the eSQE's rather than directly with developers many of the traditional problems with audits and assessments are eliminated or lessened. Embedded SQEs may also serve as external auditors of other enterprises. Their combination of Computer Science, Software Engineering, and Software Quality Assurance makes them well qualified for serving on these auditing teams.

The eSQE's solve the impedance matching problem because they are also experienced auditors themselves, and intimately familiar with compliance requirements and the practices of their development team(s). For instance, when the external auditor asks for an SRD the eSQE will understand that the external auditors are looking for records to show that the software requirements are tracked and linked to the code and tests. The eSQE reminds the external auditor the "D" in SRD is for description not document. The eSQE explains that the automated tracking of requirements in a tool such as Jira allows requirements to be access controlled, queried, sorted, linked, configuration managed, and rapidly respond to changes. The external auditor is impressed and issues a noteworthy practice instead of a finding for the SRD.

Embedded SQEs are more frequently utilized to conduct periodic internal audits (is the audited code team compliant with processes they claim to be doing?) and assessments (are code team's implementations of required practices good practices?). Internal audits prepare code teams for and assure levels of compliance required for external audits. Having internal audits which are code team friendly finds weaknesses in processes and noteworthy practices that can be shared with other code teams. Embedded SQEs may also conduct external audits on code team suppliers, open source codes, and collaborators.

Fixing Snarky Problems

Embedded SQEs may become involved in researching and fixing annoying problems encountered by developers that developers may not have time to chase down. For instance:

1. Researching why a test fails in optimized compiler mode but not in normal mode,

2. Why tests pass and fail intermittently,
3. Comparing statistical library packages and provide results to developers,
4. Writing code to explore better ways of using platform resources,
5. Writing code to investigate intermittent problems by forcing them to happen more frequently.

Investigations such as those listed above have led to discovering bugs in compilers, libraries, feeder codes, test frameworks, commercial tools, and open source codes. The eSQE can become an import resource in tracking down and fixing problems that trouble developers.

Tool Meisters

One of the keys to improving the software development process is to automate redundant tasks, such as requirement tracking, issue tracking, design, configuration management, code reviews, compile and build, multiple levels of testing, and release, to name a few. Most tools have built in work flows or "hooks" that can be used to control the order of tasks and seamless integration of tools. While contemporary automated software development tools offer tremendous benefits, they also require installation, training, maintenance, upgrades, configuration, and tailoring to a specific development team's needs. Most tools work well using the salesperson's toy demonstration code, however getting them to work effectively on the myriad of real world platforms, compilers, languages, libraries, networks, and versions of each is not an easy task. An embedded SQE can be the logical person to set up a new tool or migrate from an older tool to a better tool. The skills required to do this task are the same skills needed to write the prime code. Many process improvements do not happen because the developers might already know there is a better way to do something but do not have the bandwidth to set up the new tool or process.

Tools also require routine maintenance, beyond upgrading to the latest supported versions. When a tool does not work as it should the embedded SQE can dig into the problem for the developers and determine a work-around or fix. This is especially important for software products or releases that must be built and execute on a large variation of platform types.

The Customer's Voice

Embedded SQEs also conduct interviews with users and help desk staff to better understand the needs of the customer and look for patterns of user feedback. There is a mythical belief in many organizations that if users are not complaining about the product they must be happy with it. While lack of complaints could mean user satisfaction, it could also mean that customers are so unhappy with the software product they have stopped using it. Besides gathering information about the software's use from the customer's point of view, interviewing customers builds goodwill. The interviewed customer will feel listened to and appreciate that they have a champion for their concerns within the supplier organization. Often user insights can lead to feature enhancements which substantially improve the

users experience and the usefulness of the software product. In some cases, new and unanticipated uses of the product may emerge expanding the software's market or user base.

Help Desk Data Mining

Help desk data mining can find problems that may not be detectable with software development or testing tools, for example an incompatibility with other applications. One of the challenges with software is the infinite combination of other software that exists on user's platforms. Detecting patterns of help desk issues containing platform and application configuration information may surface a problem so preventive or mitigation measures taken.

For instance, many years ago Borland built an integrated development environment built around Turbo Pascal. It also built a desk top suite of office applications that competed with Microsoft Office. The Borland Office suite had a Word processor application called WordPerfect, a spreadsheet application called Quattro Pro, and a relational data base called Paradox. Borland eventually sold these products to Novell in 1994 who then sold them to Coral, a Canadian software company in 1996, however Borland retained the Turbo Pascal IDE product. When these products were all produced by Borland, installation was mostly trouble free. Important to note however that one of the folders that was installed onto a customer's platform was named "Borland". After the Office products migrated from Novell to Coral, they still retained an installed folder called "Borland". However, the contents of the folder Borland had been changed by the new owners. Therefore, when anyone using Coral Office on their platform installed an application built using Turbo Pascal, Coral Office stopped working. The reason of course was the Borland folder contents were overwritten by the newly installed application.

A large international shipping company found this out after they built a shipping application for their customers using Turbo Pascal as the development environment. Customers using Coral Office who installed the Turbo Pascal application caused the Office suite to stop working. The good news was they could now prepare a shipment automatically; the unwelcome news was they could no longer use their word processor, spreadsheet or data base after installing the shipping application. Reinstalling Coral Office would restore the Office applications and break the shipping application. This problem was identified by mining the shipping company's help desk inquiries and noticing a high incident of complaints from customers who installed the shipping application and used Coral Office. The solution was simple enough; rename the folder named Borland to something else in the shipping application install file. This is a good example of mining the help desk data base to find quality problems, use as test cases, and something eSQEs can do.

Assist Closing GAPS

In regulated industries, there are compliance standards with requirements that typically must be met for software development. A finding is generated during audits or assessments when records and evidence

of compliance to a requirement cannot be adequately demonstrated. There are several reasons why this could happen:

1. the required activity has not been implemented,
2. the required activity has yet to be fully implemented,
3. the required activity has been discontinued,
4. required activity is being done but there is no record to prove it is being done,
5. the auditor has made a mistake and the activity is being done and recorded.

The gap identified by the auditor between the required activity and what is actually being done would need to be closed to permit the software development process to become compliant to the governing standard's requirements. Embedded SQE's can create a plan to close a gap found in an audit or assessment. They can also assist or implement the corrective actions, which may involve rolling out a new process or new automated tools. The eSQE may also follow up with the auditor on the technical and factual accuracy of the audit report if the finding seems incorrect or unjustified. ESQE's can also conduct internal audits or be part of an internal auditing team to identify and close any gaps prior to a future external audit.

Researching and Germinating

ESQEs devote a portion of their time to searching for new tools and techniques. ESQEs should make it a point to get out of the office once or twice a year to attend technical symposiums and trade shows to gather information from industry spokespersons, other labs, commercial companies, and tool vendors. There are also several forums and blogs to locate suggested tools and obtain user feedback on them. An eSQE should search for new and promising tools, the latest version of an existing tool, compiler, operating system, library, or new technique that may have features useful for their development team. The eSQE may wish to download a demonstration version of the tool or software for use on a branch of their team's actual code or start a small pilot program with the new technique.

A common challenge when considering new tools is they invariably sound good in the advertising descriptions and work well with sample data or demonstration code. However, the question is will the tool or technique work in the code team's real environment? To determine this the eSQE may choose to download a demo version of the tool and try it out in the development team's actual environment. If the tool or technique appears to work satisfactorily in the code team's actual environment, the tool or technique can be demonstrated to the developer team by the eSQE. The development team can determine if there is further interest in rolling out the tool or technique. In addition to the desired features of the candidate tool, consideration should be given to the impact of making the change to the tool or technique, ease of switching to the new tool or technique, and availability of exporting and importing tools. New products and techniques are frequently being launched in the software development and testing community. Developers may not always be able to keep up with all the latest tools and techniques when focused on their internal commitments. ESQEs should also share among

fellow eSQEs their experiences with tools, how to install them, set them up, maintain them so that other code teams can leverage the work already completed in installing the tool or technique.

Test Tuning

Another area that eSQEs can support is test tuning. Test tuning addresses the challenge of how to optimize testing of moderate to large complex software. Fully testing moderate to large sized software is impossible to accomplish within a human's life span because the number of different possible paths and combinations of valid data quickly multiply into very large numbers. Given that testing everything is not possible, what are the best things to test? This is the purpose of test tuning. Test tuning views software as having three properties:

Property 1. All the things the software can do (Product Features)

Property 2. All the things the user of the software is likely to do (User Priority)

Property 3. All the areas of the software that are tested (Testing Priority)

Fortunately, the Pareto principle[78] applies to software, which is roughly 80% of the time 20% of the software is used. The eSQE should become aware through user interactions what the most used 20% of the software they support is. Then assure the test coverage is prioritized to that 20% of the code. Since it is virtually impossible to test all the software, it makes sense to test the part of the software most users care about. The Venn diagram in figure 13 graphically illustrates the test priority, user priority, and everything the software can do, product features.

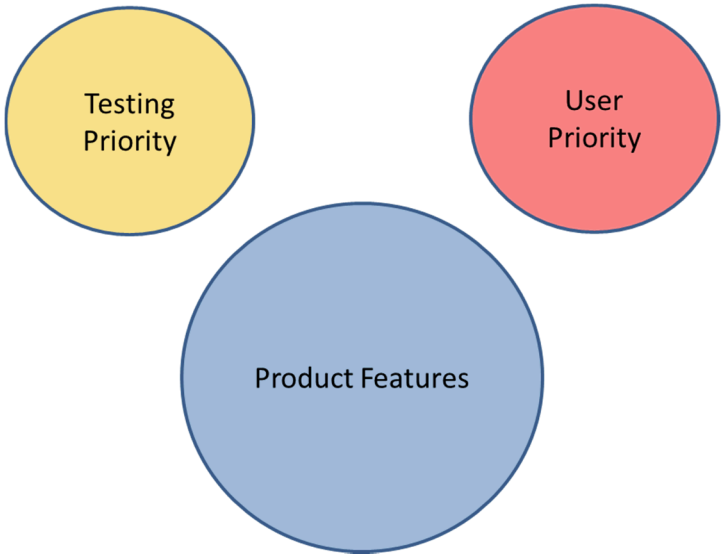

Figure 13. Testing Priority, User Priority, Product Features

The Venn diagram in figure 14 illustrates a mismatch of testing priority and user priority. Although several tests are run they do not cover all the features the user is most likely to use. Therefore, it highly likely that despite many tests being run and passing, this software is going to have a large defect discovery rate after release to users.

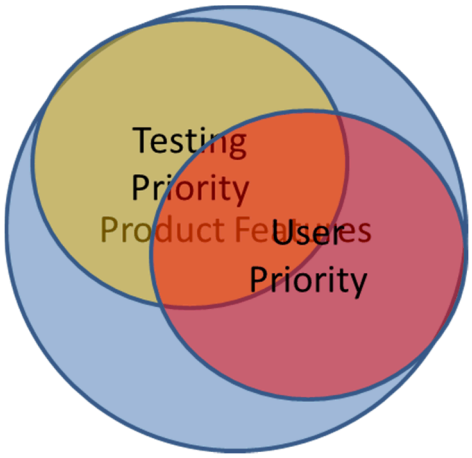

Figure 14. Testing Not Tuned to User Priorities

The Venn diagram in figure 15 illustrates the tests mostly cover the user's priorities. This is an example of test tuning; illustrating that it often is not a matter of how many tests are run, but rather the test priority is covering the part of the product most important to users.

Data for determining the test tuning can be obtained by running dynamic analyzers on the software under test to determine test coverage. User priority can be determined by interviewing users or using a profiling tool in the operational environment to determine what parts of the software is used most frequently. Tuning can also be used to determine how long each test case takes, to determine which test cases take the least time to run with the most code coverage. These tests may be good candidates for smoke tests run via the SCM tool hooks when new software is committed to the development branch. Also, over time, several test cases may evolve that are redundant and just retest areas of the software that have already been covered. These redundant test cases, especially the most time-consuming ones are candidates for elimination. Tuning the tests using dynamic analyzers and profiling tools is another area where eSQEs can contribute.

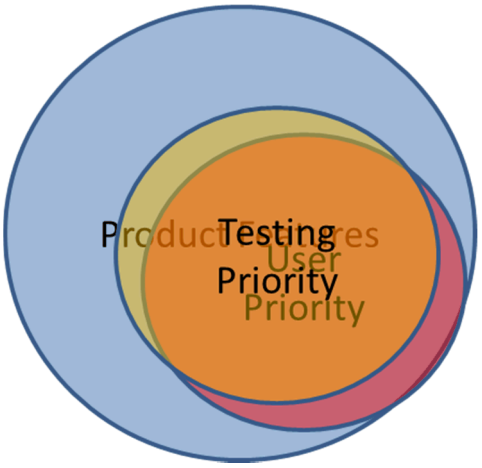

Figure 15. Testing Tuned to User Priorities

Productizing of Research Code

The purpose of research software code is experimentation, testing, and delivering proof of concept. It may be of low quality, and can be uncomfortably unstable and non-interoperable. It is used by individuals and confined groups. The purpose of production software code is live deployment and use. It, ideally, is of high quality, robust, maintainable, and scalable. It's usually serving end users.

There is a proper place for both research and production codes. For researchers, it may not be efficient to constrain creativity by insisting on a level of software development rigor more appropriate for production codes. Conversely it would be risky to release research codes to the public or use them in situations where there is a high consequence of error.

Research codes can and often do lead to computer science breakthroughs by initially demonstrating a proof of concept or prototype that appears to "work" on a specific problem, used by a specific person, on a specific platform. However, the research code may only be useful to the author or a small group of peers. To transform the research code into a useful software product for a larger population additional work typically needs to be accomplished. These additional tasks may involve adding new features to the code as well as tasks to make the code more robust, portable, extensible, usable, reliable, understandable, and supportable. Adding features (or adding functional requirements) is more easily understandable and supported by stakeholders. Adding the afore mentioned non-functional requirements to transition from a research code to a production code is a less understood and often less supported by stakeholders.

The eSQE can be one of the primary resources for supporting the transformation of research codes to productized code for a wider audience by supporting a defining, adding, and reducing process. A separate paper has been written to address the numerous tasks that may need to be accomplished to transition a code[79]. These productization tasks are shown in table 2.

Defining
1. What code the product will consist of
2. The codes purpose(s)
3. The codes intended users
4. A risk appropriate development process
5. Risks and mitigation strategies
Adding
1. Plan to allocate staff, milestones, budgets
2. A mechanism to track progress for each task
3. Task transparency for stakeholders
4. Independent testing
5. Enhanced automated testing
6. Static analysis and code repair
7. Enhanced release process
8. Enhanced Software Configuration Management (SCM)
9. Industry standard testing
10. Enhanced test code coverage
11. Enhanced dynamic analysis
12. Enhance relevant user documentation
13. Understanding of what similar or competing code can do
14. Understand customer expectations
15. Getting customer feedback
16. Understanding of market served and positioning
Reducing
1. Installation complexity
2. Technical debt
3. Non-essential code

Table 2. Productization Task for Research Software

But wont Embedded SQEs start thinking like the Developers?

In the early 1970's there was a lot less software in the world. Software was in use for commercial engineering applications (mostly Fortran), business applications (mostly COBOL), and the Department of Defense. For commercial engineering applications, the few programmers that existed worked closely with engineers who used the software to replace hand calculations, slide rules, and calculators. The programmers and users worked closely together to understand requirements, write and debug code, and fix problems the users encountered. The consequences of error for this type of engineering software would be faulty designs of jet engines, aircraft, or structures. Careful vetting of engineering codes was intuitive because of the relatively high consequence of failure. In many ways, the process used was Scrum-like because there were frequent short meetings between programmers and stakeholders. In the business sector, such as insurance companies, software consequences of failure could have major financial impacts, so again the software was carefully vetted. Much of the DoD

software being written was real-time assembler language and mission critical. DoD software was utilized in applications such as surveillance, weapons, and battlefield communications systems. For DoD applications, the consequences of failure were also quite high for obvious reasons. In the 1970s networks were not commonplace. Most mainframe computers were standalone, and most software was written with programmers and users in very close geographical proximity.

In this early era of software development, there was little literature available about how to develop quality software; however, because of the obvious consequences of failure for this early software much care was taken during is development. For instance, for DoD software, after a developer wrote code, peer developers would try everything they could think of to break the code. Next the code would run against simulators to stress the code for extended periods of time. Despite all being developers, testing was successfully accomplished. Code size was much smaller, 50 thousand lines of code was a huge program size in that era. Code size was severely limited by platform memory size, which was still dependent on expensive ferrite core memory. Main internal memory size on mainframe IBM 360's of this era was between 256k and 1024K, much less than the memory on a modern graphics card. However, in the later part of this decade larger software projects became subject to cost and schedule overruns and delivered software that did not work as expected.

To compensate for the decline in software productivity and quality in the late 70's and early 80's the DoD adopted standards such as DoD-2167. The concept of having separate and independent testers and developer's groups evolved. The testers would have knowledge in the product domain, but not necessarily be developers themselves. The thinking of this era was that independent testers would have a user's perspective (the software is a black box) and benefit by not knowing how the code worked internally. When the developers finished developing (or reached a certain phase of completeness) the testers would start testing it. This approach had a serious shortcoming because the testers would wind up asking the developers endless questions about how the software was supposed to work. This lead to arguments about how the software should work and often the strongest personalities won out and users suffered.

Enhancements to this process were made in the 1990's by having the independent testers develop tests from requirements while in parallel the software was being coded to the requirements, so testers would not have to ask developers so many questions about what the software was supposed to do when released. The requirements would be used to arbitrate how the software should operate. The developers focused on unit testing, that is writing tests themselves to check code modules they had written while the independent testers focused on system level tests against the requirements. Unit tests tended to find code structural problems and logic errors, while system level tests tended to find deviations from requirements, performance issues, user interface issues, platform issues, and negative testing issues. The field of software testing grew to include efficient ways to design good tests, perform tests, and automate tests with tools using scripting techniques and graphic user interface manipulation.

As test automation grew in the mid 1990's testers found themselves in need of some developer skills to write test scripts and use automated test tools effectively. Writing code to test code is a conundrum. Especially when the testers writing the test code were less experienced in programming than the

authors of the code they were testing. Why would the test code have fewer bugs than the code being tested? To mitigate the conundrum capture/replay was employed as a feature in test automation tools so that test scripts could be automatically generated by testers running tests manually on the graphic user interface. When capture/replay tools were in record mode the tester would move the mouse, click buttons, or keystroked test scripts were automatically generated which could be replayed to reproduce what the tester had just done. To leverage the value of the automated test code editing of the recorded script to add things like checkpoints, loops, decision points, and table-driven data was possible. So, testers needed to fortify their customer specific knowledge with many of the same skills developers used.

Enter the next century and the agile manifesto[80] of February 2001 and the concept of delivering a working software product early and often. This principle began a change for software testers, now testers would not be given a completed product to test, but instead a part of the finished product to test. This required the testers to acquire additional developer skills, so they could accomplish testing on code that was partially complete using many of the development environment tools such as integrated development environments (IDEs) and testing frameworks, drivers, simulators, debuggers, and stubs.

The need for testers to have developer skills has grown to the point where the industry was 40 years ago, when people that did testing were developers. The testers still need to have expertise in the software's user domain and are not the authors of the code they are testing, but they need developer's skills to write code to test the developer written code. Embedded SQA expands on this tester trend to apply to all the elements of software quality, not just testing.

Organization

The eSQEs ideally should not report to the same management chain as the developers. The eSQEs should report up a parallel management chain. Such as part of a Verification and Validation Group, Systems Engineering Group, or Quality Organization, separate but equivalent in hierarchy to the software developer manager. An eSQE's job performance is not measured in terms of lines of code written, but instead based on impacts of improvements made to the development productivity and quality. These impacts may be implemented as tool roll outs, tool upgrades, automated scripts implemented, analysis run, defects discovered, code fixes, more reliable test process, improvements in test coverage, happier users, gaps closed, fewer defects found after release, process and productivity improvements, increased developer satisfaction, etc. If an embedded SQE is doing a good job the development groups will want them to be part of their team.

Staffing

Most successful eSQE's have been developers at some time in their careers. They should have previously demonstrated experience implementing process improvements to software development. They should

already have experience with software development tools such as SCM. compilers, build tools, issue trackers, debuggers, and code static and dynamic analyzers. They should be inquisitive and enjoy learning new innovative technologies and methodologies. They should be team players and have a sincere desire to help others. They must be solution oriented, they are problem solvers by nature, not complainers. They are rewarded by helping the development team succeed, they do not seek out attention individually.

Good developers who are burned out from coding or are looking for a career change may make excellent eSQE's. Marginal developers will make bad eSQE's. ESQE staff without developer experience will have difficulty with many of the primary skills needed to be an eSQE such as creating code, understanding code, analyzing code, creating testing scripts, and automating the testing of code.

The number of eSQE's needed varies somewhat based of the consequence of error of the code being written. An ideal ratio of eSQE's to developers would be about 1:10, that is one eSQE for every 10 developers. A minimum would be 1:16 or one eSQE for 16 developers. ESQE's can service more than one developer group if the groups are small. Also, eSQE's can become expert of certain tools and be shared across multiple development groups for rolling out or configuring tools. However, it is best if each eSQE has a home developer group.

Return on Investment of Embedded Software Quality Engineers

The addition of eSQEs to developer teams may require a return of investment calculation to justify these resources. Assuming a ratio of one eSQE per 10 developers and/or testers and that the cost of the eSQE resource is the same as the cost of a developer. Furthermore, assume the cost of a defect that gets into the field is some monetary value that can be itemized and calculated with a table such as table 3 below. The time spent, hourly rates, currencies, and the number of process steps (rows) to make repairs may vary between enterprises. For instance, a business that repairs a bug on its website may have a different cost than a business that must distribute a patch to external customers. This ROI model assumes there is some cost associated with errors escaping to users and an average value can be calculated on the listed direct costs.

The ROI model calculated in this example will use the cost of a defect derived from the cost model in table 3 from an example organization. The amount averages $6,612 per escaped defect that is deemed important enough to fix, with the cost breakdown shown in table 3. Next is the cost of developers (their salary), assuming a $75 per hour or $150,000 labor cost annually. Assuming an eSQE costs the same as an average developer, adding an eSQE must prevent 23 ($150,000/$6,612) defects from escaping annually, or approximately 2 per month to breakeven. The breakeven number may vary depending on organization type. It should be noted that some escaping defects could cost much more than the average $6,612. For instance, in regulated industries a software defect could cost one or more human lives, large losses of money, loss of goodwill, drop in stock price, fines, recalls, closure, etc. For the purposes of this ROI example averages will be used.

Given the specific tasks an eSQE performs as described in the preceding sections, is the possibility of finding or preventing at least two defects per month within a group of 10 developers and/or testers highly likely? Some of the tools mentioned in the preceding sections such as static and dynamic analyzers can find thousands of potential defects in a few hours. Authoring scripts to automate tests or rolling out continuous integration tools support testing early and often, using failed tests to discover defects, conducting code reviews may find deviations from the requirements in the code or missing requirements (requirements not implemented in the code). However, preventing and discovering defects is only one way to evaluate return on investment.

Direct Cost of Defects					
Task	Time (Hrs.)	Rate ($/Hr)	Employees	ODC	Direct Cost
Tech Take Problem Report	1	$41.00	1	$50.00	$91.00
Tech Log Problem Report	0.5	$41.00	1		$20.50
Tech/SE Simulate Problem	8	$62.50	2		$1,000.00
Tech/SE Discovers Problem	4	$62.50	1		$250.00
Tech/SE Inform Engineering	1	$62.50	2		$125.00
Engineering Evaluation	8	$75.00	2		$1,200.00
Eng'g Isolates Cause	4	$75.00	2		$600.00
Eng'g Designs Solution	8	$75.00	2		$1,200.00
Eng'g Test Solution	4	$75.00	2		$600.00
TAC/SE Validates Solution	4	$62.50	2		$500.00
Q/A Regression Tests	2	$62.50	1		$125.00
Notify Customers	4	$41.00	1	$50.00	$214.00
Cut New Release	4	$62.50	1		$250.00
Revise Documentation	2	$41.00	1		$82.00
Distribute Patch	4	$41.00	1	$200.00	$364.00
Total	58.5				$6,621.50

Table 3. Cost of a Defect Escaping to the User[81]

In regulated industries, proof of compliance to governing standards using records of software development activities is essential. Inability to do this could result in fines, litigation, or a suspension of the company from the marketplace. The penalties and costs associated with these onerous actions may easily exceed the cost of eSQE's and justify the added resources. This is especially true in cyber security, where software defects contribute to vulnerabilities in developed code or supporting code to be exploited by hackers. In some recent cases millions of records containing personal data of individuals have been stolen resulting in companies having to provide credit monitoring services to their customers. In addition to cyber security, software is finding its way into consumer goods, such as automobiles that drive themselves, home security, patient monitoring, point of sale transactions, all increasing the consequences of failure for the public.

Observations

The embedded SQE concept has been used with success over the past decade at the Lawrence Livermore National Laboratory primarily on the Advanced Simulation and Computing (ASC) project. It has taken many years to refine the process and identify characteristics of successful eSQEs. The concept is branching out to smaller research projects and next generation efforts. Probably the most challenging

part of the embedded SQE implementation has been the search for and recruitment of qualified staff. Fortunately, one eSQE can support many developers and good ones are considered extremely valuable to their teams. Part of the unique challenge recruiting at LLNL has been the domain knowledge of our one of a kind supercomputers and complex physics simulations in addition to the computer science experience needed. In an environment with more commonly used platforms and less scientific knowledge needed recruiting qualified eSQEs should be easier. It is important to note that the techniques presented in this paper are empirical and not theoretical and should therefore have reduced risk for roll out in other organizations finding themselves in regulated industries or wanting to improve software quality standards.

Conclusion

This paper has emphasized that regulated industries have evolved to include a large amount of software. The software industry has made remarkable advances in technology over the past 50 years to grow from an obscure domain of small science and actuarial applications to supporting hundreds of millions of lines of code we each depend on daily[82]. The consequences of buggy or vulnerable code is constantly in the news wreaking havoc in the lives of millions of people. Despite advancements in software development, compliance to standards in regulated industries and adherence to quality standards has not kept pace. This paper presents a detailed approach which has been proven successful in one of the most challenging software development environments. It is hoped that our lessons learned can help your organization implement some or all our techniques to transform organizations from audit and punish to three-dimensional quality and a safer world for all.

End Notes

[1] *The term audit is defined as comparing what is supposed to be done as defined in the governing standard to what is actually done. The term assessment includes the definition of the term audit, but also includes an evaluation of the goodness of the governing standard's requirements and the goodness of the method used to implement the governing standard. A passed audit means what is required to be done is in fact done. A passed assessment also means what is being required is what is being done and are reasonable things to be doing.*

[2] *The evolution of auditing: An analysis of the historical development*, LEE Teck-Heang1, Azham Md. Ali, Dec. 2008, Vol.4, No.12 (Serial No.43) Journal of Modern Accounting and Auditing, ISSN1548-6583, USA

[3] http://www.ibtimes.com/obamacare-website-failure-analysis-why-site-crashed-so-often-during-2013-launch-2097354

[4] http://www.cbsnews.com/news/obamacare-website-failed-in-tests-just-before-launch-date/

[5] The Losada ratio was proposed by Marcial Losada and Barbara Fredrickson when they identified a ratio of positive to negative affect of exactly 2.9013 as separating flourishing from languishing individuals in a 2005 paper in American Psychologist. However in 2012 Nick Brown, a graduate student collaborated with physicist Alan Sokal and psychologist Harris Freidman to question Losada's math. While 2.9013 may not be mathematically valid, many other studies do indicate a ratio of praise to critical feedback

[6] https://hbr.org/2013/03/the-ideal-praise-to-criticism

[7] Fredrickson BL, Losada MF (2005). "Positive affect and the complex dynamics of human flourishing.". *Am Psychol*. **60** (7): 678–86. PMC 3126111. PMID 16221001. doi:10.1037/0003-066X.60.7.678.

[8] *Instant Influence*, Michael V. Pantalon, PhD. May 2011, Little, Brown, and Company, ISBN 978-0-316-08334-8

[9] Heisenberg, W. (1927), "Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik", Zeitschrift für Physik (in German) 43 (3–4): 172–198, Bibcode:1927ZPhy...43..172H, doi:10.1007/BF01397280.. Annotated pre-publication proof sheet of Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik, March 23, 1927.

[10] The author wishes to note that the eight factors which distort software auditing and assessments described in this paper are extracted from real life experience and not made up. The factors are exacerbated when one or more of the auditing team is not an experienced software developer.

[11] http://www.satisfice.com/articles/et-article.pdf

[12] https://blogs.msdn.microsoft.com/imtesty/2007/04/04/testing-is-not-responsible-for-quality/

[13] https://www.grok-interactive.com/blog/happy-developers-are-good-developers/

[14] http://www.businessinsider.com/southwest-airlines-puts-employees-first-2015-7

[15] http://www.softwarebyrob.com/2006/10/31/nine-things-developers-want-more-than-money/

[16] http://dimsboiv.uqac.ca/8INF851/web/part1/introduction/The_Agile_Manifesto.pdf

[17] http://www.silverbuckshot.net/software_risk_grading_tool

[18] http://www.umsl.edu/~hugheyd/is6840/waterfall.html

[19] https://en.wikipedia.org/wiki/Agile_software_development

[20] http://www.c-sharpcorner.com/UploadFile/d9c992/the-agile-scrum-framework/

[21] https://confluence.atlassian.com/jiracorecloud/task-management-with-jira-780866594.html

[22] https://confluence.atlassian.com/agile/glossary/kanban-board

[23] http://www.softwaretestinghelp.com/requirements-management-tools/

[24] http://www.softwaretestinghelp.com/popular-bug-tracking-software/

[25] https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design

[26] https://en.wikipedia.org/wiki/Structured_programming

[27] https://en.wikipedia.org/wiki/Modular_programming

[28] https://en.wikipedia.org/wiki/Coupling_(computer_programming)

[29] https://en.wikipedia.org/wiki/Cohesion_(computer_science)

[30] https://en.wikipedia.org/wiki/Cyclomatic_complexity

[31] https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)

[32] https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)

[33] https://en.wikipedia.org/wiki/Object-oriented_programming

[34] https://en.wikipedia.org/wiki/Class_(computer_programming)
[35] https://en.wikipedia.org/wiki/Unified_Modeling_Language
[36] https://www.visual-paradigm.com/features/code-engineering-tools/
[37] https://en.wikipedia.org/wiki/Comparison_of_code_generation_tools
[38] http://www.aivosto.com/visustin.html
[39] "Bug Taxonomy and Statistics" Appendix, section 3xxx, SOFTWARE TESTING TECHNIQUES, second edition, Van Nostrand Reinhold, New York, 1990, by Boris Beizer
[40] https://en.wikipedia.org/wiki/Duplicate_code
[41] https://www.klocwork.com/
[42] http://www.coverity.com/
[43] https://clang-analyzer.llvm.org/
[44] http://rosecompiler.org/
[45] https://www.parasoft.com/product/insure/
[46] https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis .
[47] https://en.wikipedia.org/wiki/Fagan_inspection
[48] https://en.wikipedia.org/wiki/List_of_tools_for_code_review
[49] http://www.gnu.org/software/autoconf/autoconf.html
[50] https://cmake.org/
[51] http://ant.apache.org/
[52] https://jenkins.io/
[53] https://www.atlassian.com/software/bamboo
[54] http://cruisecontrol.sourceforge.net/
[55] https://en.wikipedia.org/wiki/List_of_build_automation_software
[56] https://codeflu.blog/2014/12/26/using-gcov-and-lcov-to-generate-beautiful-c-code-coverage-statistics/
[57] Ibid
[58] https://software.intel.com/en-us/articles/how-to-use-intel-compiler-code-coverage-to-determine-the-code-exercised
[59] https://stackify.com/code-coverage-tools/
[60] https://en.wikipedia.org/wiki/Rational_Purify
[61] https://en.wikipedia.org/wiki/Valgrind
[62] https://en.wikipedia.org/wiki/List_of_performance_analysis_tools
[63] https://www.cs.uoregon.edu/research/tau/cca/
[64] https://software.intel.com/en-us/-getting-started-with-intel-vtune-amplifier-xe-2017
[65] https://en.wikipedia.org/wiki/Central_processing_unit
[66] https://en.wikipedia.org/wiki/Graphics_processing_unit
[67] https://en.wikipedia.org/wiki/List_of_performance_analysis_tools
[68]https://www.ibm.com/support/knowledgecenter/en/ssw_aix_61/com.ibm.aix.prfusrgd/doc/prfusrgd/ch15body.htm
[69] http://searchsoftwarequality.techtarget.com/tip/Performance-and-load-stress-tests-Two-types-of-capacity-tests
[70] https://en.wikipedia.org/wiki/Load_balancing_(computing)
[71] https://www.apicasystem.com/blog/load-testing-vs-stress-testing-vs-capacity-testing/
[72] https://www.educba.com/web-performance-testing-tools/
[73] https://www.atlassian.com/git/tutorials/what-is-git
[74] https://subversion.apache.org/
[75] https://www.mercurial-scm.org/
[76] https://en.wikipedia.org/wiki/List_of_version_control_software
[77] https://en.wikipedia.org/wiki/Comparison_of_wiki_software
[78] https://en.wikipedia.org/wiki/Pareto_principle
[79] *Productizing Research Software,* LLNL-CONF-734945, Gregory Pope, Lawrence Livermore National Laboratory, February 2017
[80] http://agilemanifesto.org/

[81] A Software Quality Engineering Maturity Model, Gregory M Pope, Ellen M Hill August 19, 2010, ASQ Silicon Valley Quality Conference 2010, LLNL-CONF-413143

[82] The author wishes to acknowledge that there are probably still some inhabitants of planet earth that do not depend on software for anything. They have yet to experience identity theft, a virus crashing their computer, or the joy of pop-up ads.