LA-UR- 09-01250

Approved for public release;
distribution is unlimited.

Title:	Software Archeology: A Case Study in Software Quality Assurance and Design
Author(s):	John MacDonald, Jane Lloyd PMT-4 Los Alamos National Laboratory Cameron J. Turner Colorado School of Mines
Intended for:	2009 ASME IDETC/CIE Conference San Diego, CA August 30-September 2, 2009

Form 836 (7/06)

Proceedings of IDETC'09:

ASME 2009 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference
San Diego, California, August 30-September 2, 2009

DETC2009-XXXXX

SOFTWARE ARCHEOLOGY: A CASE STUDY IN SOFTWARE QUALITY ASSURANCE AND DESIGN

Cameron J. Turner
Colorado School of Mines
1500 Illinois Street
Golden, Colorado 80401
cturner@mines.edu

John M. MacDonald
Los Alamos National Laboratory
PO Box 1663, MS E530
Los Alamos, New Mexico 87544
jmac@lanl.gov

Jane A. Lloyd
Los Alamos National Laboratory
PO Box 1663, MS E530
Los Alamos, New Mexico 87544
jlloyd@lanl.gov

Abstract

Ideally, quality is designed into software, just as quality is designed into hardware. However, when dealing with legacy systems, demonstrating that the software meets required quality standards may be difficult to achieve. As the need to demonstrate the quality of existing software was recognized at Los Alamos National Laboratory (LANL), an effort was initiated to uncover and demonstrate that legacy software met the required quality standards. This effort led to the development of a reverse engineering approach referred to as software archaeology. This paper documents the software archaeology approaches used at LANL to document legacy software systems. A case study for the Robotic Integrated Packaging System (RIPS) software is included.

1. INTRODUCTION

In an ideal world, quality would be engineered into software during the design process just as it is engineered into hardware during design. While modern designs often apply this level of rigor to software as well as to hardware, this has not always been the case. Software was often created so that the system would work, with little thought given to its design or quality. As long as the system configuration (both hardware and software) remained constant and those responsible for the design remained available to deal with problems, the lack of detailed design documentation is not a significant problem. But when changes become necessary or the original people responsible for the software are lost to other programs, the quality of the software becomes important.

Software Quality Assurance (SQA) uses design techniques to instill quality into the design of the software. SQA may also be just thought of as software quality. The goals of SQA include: 1) the instillation of quality into the software, 2) documenting the design of the software so that information can be transferred between qualified practitioners, 3) the facilitating of integrated system testing so that erroneous conditions can be avoided, 4) to enable evaluation of modifications upon the system and revalidation of the system as necessary, 5) provide reliable and quality data, and 6) to reduce the risk of software failure. The following sections further examine the necessity of SQA, its relationship to design methods, and review the implementation of a legacy SQA project for an automation system at LANL.

2. NECESSITY OF SOFTWARE QUALITY ASSURANCE

Many program managers have asked "Why is software quality assurance a necessary component in many engineering systems?" The best answer is that SQA can reduce project costs by preventing hardware/software conflicts, failures or errors, facilitating software changes and upgrades, ensuring that the customer expectations are met, provide assurance that the software implementation is complete and reduction of software associated risks.

Ideally, SQA is implemented during the original design program. However, in many cases, it may be difficult to prove what was done in this regard, if the process was not well-documented. This is often the case in legacy systems. In these situations, additional effort is necessary to recreate the original design documentation. These efforts are known as Software Archeology, a term attributed to Ralph Johnson of The University of Illinois at Urbana-Champaign [Johnson, 2005].

2.1. PURPOSE OF SQA

SQA programs attempt to ensure that the needs of the customer(s) are met by the software. These needs can be described as expected, targeted and unexpected as shown in Fig. 1. The expected requirements are often unstated by the customer – they are "expected" to be present in the software and their absence is a major source of customer dissatisfaction. The targeted needs are those that the customer intends to satisfy through the use of the software. These needs are also expected, or the customer will be dissatisfied with the software, but their

presence is not necessarily a source of customer satisfaction. Unexpected needs are software features that meet needs of which the customer is unaware. The presence of unexpected software features is a major source of customer satisfaction.

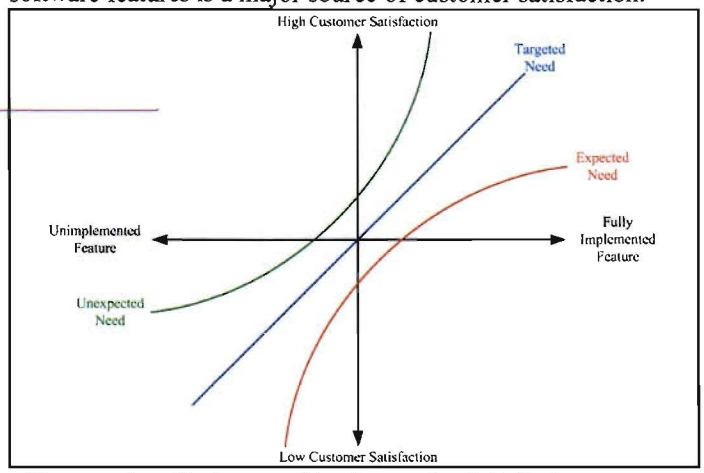

Figure 1. Customer Needs and Satisfaction versus Implementation Quality.

Fig. 1. is similar to a graph produced by the Kano model. The model developed by Professor Noriaki Kano for understanding customer needs and satisfaction [Pyzdek 2003].

The expected and targeted needs of the customer should be readily identifiable and are typically specified by the customer. These needs include the core purpose of the software (a targeted need) and the user interface (expected needs) of the software. However, identifying the unexpected needs, such as fault recovery, conflict handling, and system integration may not come from the end-customer initially. For instance, system integration is a need of the intermediate customer, the system integrator and is unlikely to be an initial concern of the end-customer. Other needs, such as fault recovery and conflict handling may be of concern to the end-customer only once those states are identified. A systematic study of the system is often the best way to identify these expectations.

An SQA program seeks to produce quality software that meets the needs of the customer in terms of functionality, usability, reliability, performance, and supportability. All of these components contribute to the development of quality software. A structured approach to SQA aids the software engineers in the task of identifying expected and unexpected needs that are often unarticulated by the customer. Without this structured approach, these needs would not be translated into requirements, incorporated into the software, or tested and maintained as the software is incorporated into the system leading to general customer dissatisfaction with the software.

The software development process must translate customer needs into software requirements and specifications that can be transformed into actual software code. This process is a series of decisions and assumptions that impact the software design and implementation. An SQA program documents these decisions and assumptions for future use. A SQA program also encourages the development of an integrated testing strategy so that the software can be evaluated versus the customer needs during development and once development is complete. Thus an SQA program is crucial to establishing the ability of a software package to meet the needs of the customer(s). Since software is often an integrated component in a larger system, quality software is a crucial to achieving and maintaining the system's ability to produce quality results (i.e. data, product, etc).

As software ages, an SQA program becomes increasingly important if system quality is to be maintained. Computer hardware often becomes obsolete within years of purchase and even relatively minor hardware or operating system changes can affect software performance, reliability and functionality, with potential negative impacts upon system quality. SQA programs provide a critical mechanism to provide change control and post change verification testing to ensure that customer needs remain satisfied. SQA provides the necessary documentation to design software upgrades, a structured methodology to document changes, and a consistent method for software testing with respect to the needs of the customer.

Software Quality Assurance activities attempt to verify and validate software using well proven engineering methodology. The ends result of SQA activities provides software that is documented, reviewed, tested, maintainable, robust and reliable. SQA also provides an organized approach over the life cycle of the software. Additional benefits are realizable by the stability of the software and added saving of reduced reworks and reduced software failures.

2.2. SQA AS GOOD ENGINEERING PRACTICE

Ensuring software performs as expected on a hardware platform is a good engineering practice and is demonstrated through software testing. Testing validates that the software runs properly and meets requirements specified. Software engineers, like hardware engineers, often operate strictly within their own domain without considering necessary interfaces between software and hardware. System engineering approaches from a global perspective, and endeavors a system-level approach, considering software and hardware to be parts of a system. Both parts of the system need to perform together for the system to meet engineering requirements. This can only be accomplished by collaboration between the engineers responsible for hardware development and those responsible for software development. In addition, not only is software and hardware considered from a system engineers perspective but also items such as safety, reliability, quality, producibility, environment, physical dimensions, maintainability, human factors, economics and technical factors.

For example, a hardware memory failure can impact the ability of the software algorithm to run correctly on a given system. On the other hand, one can have the greatest software algorithm ever conceived of on paper but unless it is properly implemented on a suitable hardware platform it may not prove to be of any value. Both hardware and software are needed to form a system. Software cannot be tested without hardware and hardware cannot be fully tested without software.

What can cause software failures? Software syntax errors such as typing in a "," instead of ";" depending upon the programming language can cause immediate code errors. Another example is if a partial or total hardware failure such as memory or hard drive failure resulting in a partial loss or corruption of data or even a total loss of data due to an unintentional overwriting of data. Timing and latency in a

system can cause a loss of response in real-time systems. For example, software running on a network can become so saturated with activity that it is impossible for software to response.

An electrical power loss will may cause the hardware to cease to function, even if the software is not at fault. However, without power, the software will not function. Electromagnetic energy fields fluxes can cause havoc on compute systems on both the hardware and on the software executing within the processor. Binary bit-flipping or large scale blocks of data erasure can occur due to electromagnetic fluxes. An improperly grounded and protected system can be affected by power surges and lightening strikes affecting hardware and software alike. This type of failure can result in unpredictable results and affect system-wide reliability.

Even when code is written precisely to a standard (for portability) when one move to another hardware platform difficulty in the way the compiler or interpreter implements the standard may be found in execution. Patches and repairs code for software is noticeably common. New features and additions are also prevalent in software patches, software versions and just new software. Alpha, alpha Beta code testing is very common on large software projects. However, this level of testing is not always practiced on small-scale and in particular research and development (R&D) engineering projects.

2.3. IMPACT OF A LACK OF SQA

For new software, the lack of SQA puts the entire software project at risk. The Software QA/Test Resource Center website [Hower, 2009] maintains a listing of some of the more significant software failures attributable to a lack of SQA. A few of the more interesting highlights include:

- In January 2009, regulators banned a health insurance company from selling policies due to computer bugs that resulted in erroneous denials of coverage or outright cancelations in coverage to certain patients. These errors threatened the health and safety of beneficiaries.
- A January 2009 news report indicated that a major IT consulting company has spent four years correcting problems caused by an inadequately tested software upgrade.
- In August 2008, more than 600 airline flights were delayed due to a software glitch in the FAA air traffic control system.
- A lack of software testing was blamed for problems that led to privacy breaches into the records of several hundred thousand customers of a large health insurance company in August 2008.
- In December 2007, inadequate software testing of a new payroll system was blamed for $53 million of erroneous payments to employees of a school district.
- An April 2007 subway rail car fire was caused by the failure of a software system to perform as expected in detecting and preventing excessive power usage in the new passenger cars. The subway system had to be evacuated and shut down for repairs.
- A March 2007 recall of medical devices was blamed on a software bug that failed to detect low power levels in the devices.
- A September 2006 news report indicated that insufficient software testing led to voter check-in delays during the primary elections in that state.

There are many additional examples of a lack of control over the development of software that led to unintended failures. Clearly the need exists for producing better quality software. Software development failures also have been documented within the US Department of Energy (DOE) Laboratory Complex and at LANL. A series of 2006 events led to the implementation of new SQA requirements at LANL. These new requirements applied not only to new software projects, but to legacy systems and led to the need to develop methods to apply SQA techniques to legacy systems.

2.4. A RELATIONSHIP BETWEEN SQA AND DESIGN

Figure 2 describes a typical product lifecycle from the early problem identification process through design, production and the eventual retirement.

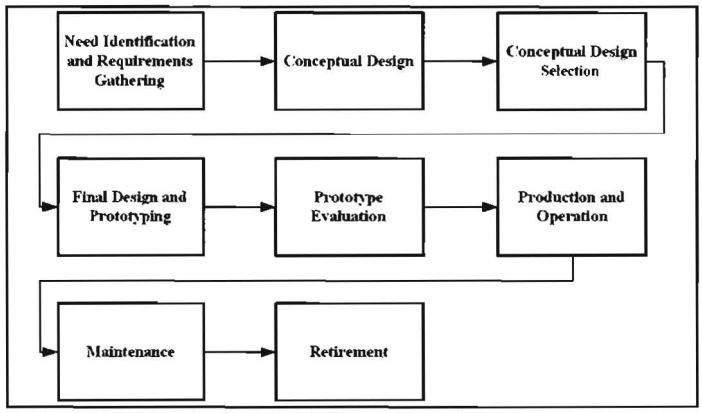

Figure 2. A typical product development process.

SQA naturally fits within a product development process such as that described in Fig. 2. Note the similarities to the ASME Standard NQA-1 process for software development as shown in Fig. 3. NQA-1 is further discussed in Section 3.

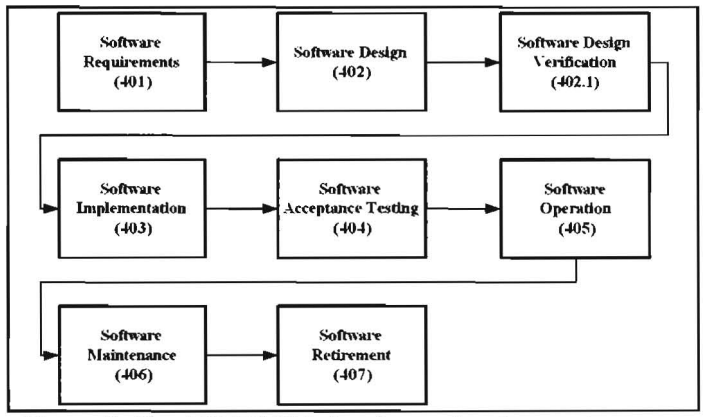

Figure 3. The ASME NQA-1 software development process. [ASME, 1997]. The appropriate sections of the standard are also indicated.

The architectural similarities between a generic development process and an SQA compatible software

development process are very similar. This should not be surprising, but should be expected. SQA is not an additional complexity to be added to the design process, but rather, SQA is a documentation of a structured design process. Properly done, SQA adds very little effort to a design effort, but instead documents the decisions made during that process.

However, when that documentation is lacking, as may be the case with legacy software, SQA may become a more resource intensive process. One option is to simply redesign new software to replace the legacy software, but to fully implement an SQA program during the design cycle. Unless other factors require it, such as a need to replace the existing system, this approach would seem to be extremely costly in terms of both time and effort.

The alternative approach is to use reverse engineering techniques to evaluate the quality of the legacy software, and to redevelop the supporting documentation. Here again, a reverse engineering design process can be applied to software, much as it is applied to hardware. The tasks include:

- Redeveloping the original customer needs that led to the original software development;
- Translating those needs into requirements and specifications;
- Mapping the requirements and specifications into the functional form of the design;
- Developing appropriate testing procedures to confirm that the requirements and specifications are met in the software as implemented; and
- Producing appropriate maintenance, upgrade and retirement plans and procedures.

The similarities in activities between quality assurance and design procedures are striking. Specific techniques utilized at LANL to reverse engineer legacy software will be noted in the case study in Section 4. The common fear of most engineers when faced with a new quality assurance program that there will be additional effort and the design process will suffer is probably unfounded. What is required is a simple documentation of the activities that have already occurred.

3. SOURCES OF SQA STANDARDS

Several professional organizations have arrived at standards for SQA programs. Among them are the American Society of Mechanical Engineers (ASME), Nuclear Quality Assurance Level 1, referred to as NQA-1 [ASME, 1997][1]. NQA-1 forms the basis for most of the relevant DOE and LANL standards and requirements for SQA. In addition to NQA-1, relevant IEEE computer engineering standards such as IEEE 1228-1994 [IEEE, 1994], Department of Defense standards such as MIL-STD-882D [DOD, 2003] and standards from the American Society for Quality (ASQ) were used to further refine the meanings of the ASME standards.

3.1. REGULATORY DRIVERS

DOE SQA programs are driven by regulations in 10 CFR 830.122. This code specifies a Quality Assurance Plan (QAP) and indicates that the QAP must address management, performance, and assessment criteria. Additional requirements are imposed for software if its location or use may affect the safety and/or security of a facility. Professional standards including ASME-NQA-1 have been codified into this code.

10 CFR 830.122 [CFR, 2002] resulted in DOE Order 414.1C [DOE, 2005], which is specific to quality assurance, safety software, and software defined as computer programs, procedures, and associated documentation and data pertaining to the operation of a computer system within DOE nuclear facilities. LANL translated this order into a LANL LIR 308-00-05.1 entitled "Software Quality Management" revised on December 29, 2006. This document has been further superseded by additional procedures and requirements.

At each level, the details of SQA implementations become increasingly specific. The SQA program developed for the Advanced Recovery and Integrated Extraction System (ARIES) project, and used as the basis for the legacy work on RIPS, has successfully pass audits several times and found to be in accord with all of the relevant DOE and LANL procedures.

3.2. THE ARIES APPROACH TO LEGACY SOFTWARE

The Advanced Recovery and Integrated Extraction System is a program active at LANL since the mid-1990s. The program runs a series of gloveboxes, many of which contain integrated automation and processing systems [Turner, 2008, 2009] for which extensive customized software was created. The ARIES glovebox lines convert nuclear materials from retired nuclear weapons into forms suitable for packaging for long-term storage, international inspection and for reuse as mixed-oxide (MOX) reactor fuel [McKee, 2008]. Because of the potential to reuse ARIES material in nuclear reactors, the need for an SQA program was recognized by the ARIES project long before other programs at LANL realized the need.

However, ARIES still had hundreds of thousands of lines of code for which the necessary documentation of the software quality was incomplete. To correct this deficiency, the ARIES program embarked on an aggressive software reverse engineering program to establish a defensible SQA pedigree for its legacy software systems. This program quickly became known as a software archaeology effort and is the basis for the case study in the following section.

This project used the ASME NQA-1 standard as a baseline for what information needed to be identified, documented and retained for both legacy and new software systems. These requirements are shown in Fig. 4. Some elements are only required of new software. Others are optional, and their need is determined during the development of the initial software project plan through a risk analysis.

The ARIES SQA plan has been through several internal and external audits and has earned glowing reviews each time. Furthermore, the program did review issues which had not been previously identified during system integration, acceptance testing or system operation. Most importantly, the project has increased confidence in our end-users that our product is produced to meet the required specifications.

[1] Note that there are more recent versions of NQA-1, however, the DOE Orders specifically reference NQA-1-1997, and so therefore the SQA program is based on this version.

4. SOFTWARE ARCHEOLOGY: RIPS CASE STUDY

The Robotics Integrated Packaging System (RIPS) is one of six major processes that make up the ARIES glovebox line. RIPS occupies one of the gloveboxes in the line and is responsible for packaging nuclear materials produced by the other systems into stainless steel cans that meet the DOE 3013 packaging standard [DOE, 2004]. The cans are automatically packaged with two robotic systems and uses five independent subsystems to complete the process. These systems are controlled by no less than six separate computer systems and interface with six additional "intelligent" instrumentation subsystems. [Turner, 2008, 2009]

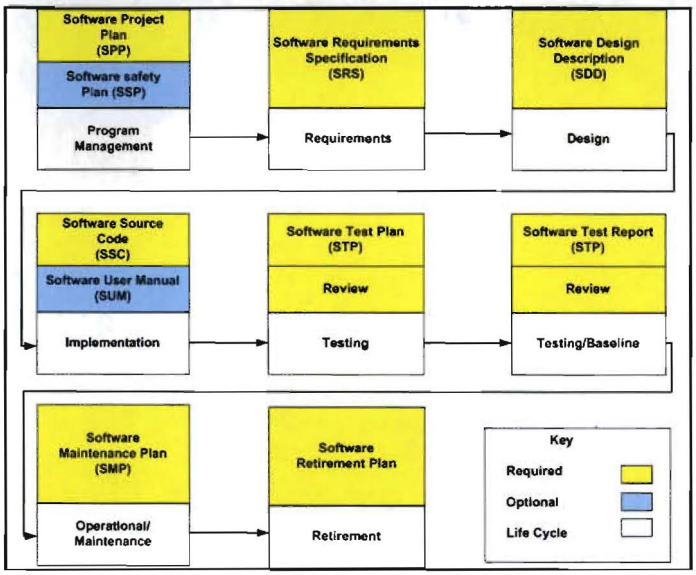

Figure 4. The required elements of the ARIES SQA program.

The RIPS glovebox, Fig. 5, is divided into three chambers called the hot side (which is radioactively contaminated), the cold side and the fluid processing side. Materials to be packaged in RIPS arrive in the hot side in a crimped convenience can. One robot then loads the convenience can into a 3013 stainless steel can which is then welded shut in a helium atmosphere. The welded can is inspected, leak checked and placed in an electrolytic decontamination chamber to be radioactively decontaminated (see Fig. 6 and 7).

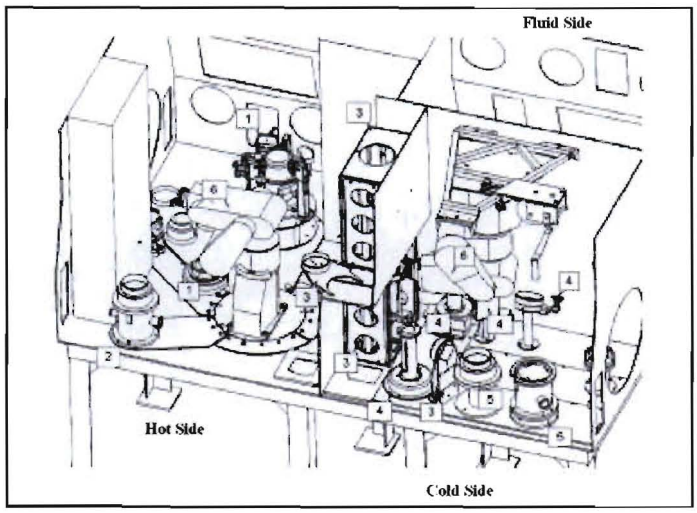

Figure 5. Schematic of the ARIES RIPS Module.

Within the electrolytic decontamination chamber, the surface of the can is electropolished which removes contamination from the surface of the can. The chemicals used to electropolish the can are recycled in the fluid processing chamber of the glovebox and the removed contamination is collected and removed from the system. Once the process is complete, the can is transferred to the cold side of the glovebox for final processing.

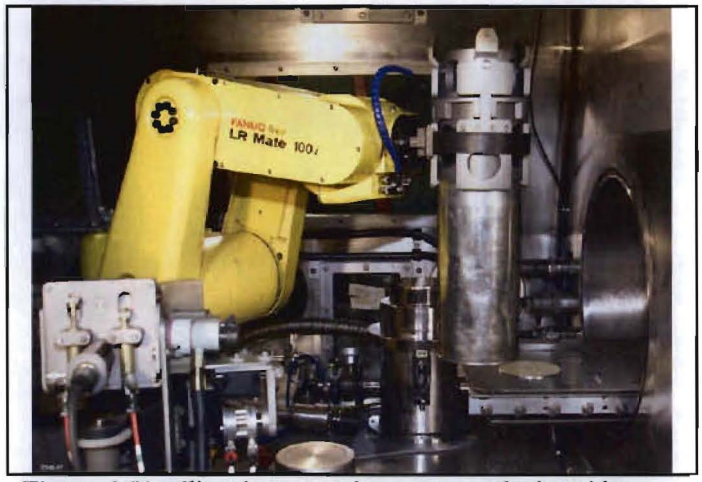

Figure 6. Handling the convenience can on the hot side.

Figure 7. Handling the welded 3013 can.

On the cold side, Fig. 8, a second robot conducts a radiation survey on the surface of the can, and conducts a second leak check to confirm that the can remains sealed. Once these checks are completed, the can is released from the RIPS module and taken to the next process in the ARIES process.

Figure 8. RIPS Cold Side Activities.

The operation of the RIPS system is controlled by a master PC, which can delegate control of the system to either robot, the welding subsystem, the electrolytic decontamination system, or the two leak check systems. The master computer and each of these subsystems include software which needed to be evaluated. In addition, the radiation checks use three additional "intelligent" instruments to survey the surface of the can. These instruments also included software that needed to be addressed.

4.1. SOFTWARE PROJECT PLAN

The Software Project Plan (SPP) is the initial step in the reverse engineering process used by ARIES. The purpose of the SPP is to document the original customer expectations for the software and to establish a plan to complete the SQA process. With RIPS, an early consideration was how to deal with the "intelligent" instruments.

These instruments contain internal software, often in the form of firmware, which processes the sensor data and provides a result to a local user interface and to the main system through a network connection. On one hand, the firmware is software which needed to be validated. On the other hand, this software can only be modified by the vendor, and since the systems are located in a secure facility, the firmware configuration is controlled. Furthermore, the instruments were subject to a calibration plan, which also served to verify that the firmware and the sensor are functioning correctly. Consequently, it was determined early in the reverse engineering process that these intelligent instruments already were quality controlled and did not need to be reverse engineered.

Also, the safety significance of the software had to be analyzed. The safety analysis was conducted with a formalized questionnaire completed by the responsible system engineers that resulted in a determination of the level of safety significance of the software. RIPS was determined not to be safety significant software and therefore did not require a Software Safety Plan (SSP). It was also determined that a separate maintenance plan was not necessary since the system maintenance plan was also being developed at the time and thus both documents could be integrated together. Finally, the development of a retirement plan was deferred since there are no immediate plans to retire this new system.

The SPP identified the major customer expectations for the system, including:

- The automated welding of the 3013 containers;
- The automated verification that the weld sealed the container;
- The automated decontamination of the outer surface of the container; and
- The automated verification of satisfactory decontamination and that the container remains intact.

In addition, the SPP began to develop the structure of the software system and the major hardware and software components resulting in the diagram in Fig. 9.

The development of the SPP used design techniques that included interviews of the operators and engineers responsible for the system (customer interviews), high level functional analysis diagrams, system configuration diagrams, and literature reviews (of available system documents).

4.2. REQUIREMENTS & SPECIFICATIONS

The next phase of the SQA process for RIPS involved the development of the very general customer expectations given above into a detailed set of engineering requirements and specifications. These included both requirements for system functionality during normal and abnormal operations as well as requirements for data interfaces and network connections between the various hardware systems. All of this information was documented in a Software Requirements and Specifications (SRS) document.

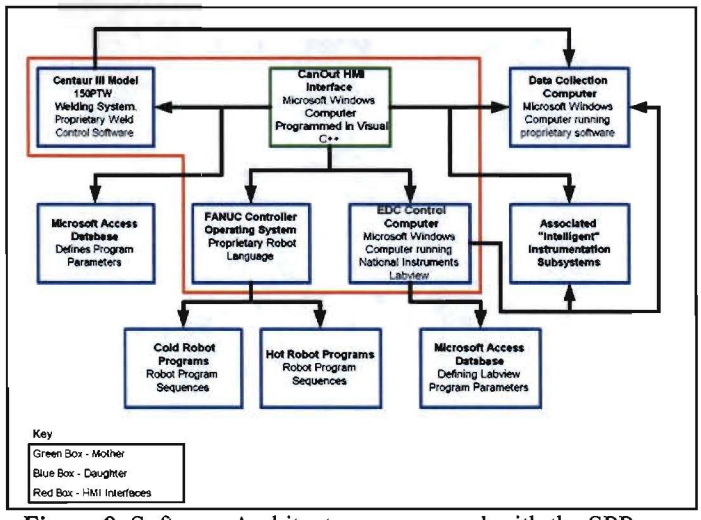

Figure 9. Software Architecture uncovered with the SPP.

This analysis led to the development of a Requirements Traceability Matrix (RTM), organized hierarchically into 33 major categories and encompassing more than 500 individual requirements. Each requirement was also associated with a particular component for Fig. 9. This matrix was used in subsequent documentation.

In constructing the RTM, a detailed diagram of the operation of the top level software system (called CanOut) was generated. This flow chart was instrumental in the development and organization of the RTM and in the development of testing procedures. This diagram is shown in Fig. 10.

The methods used to generate these descriptions included task decomposition, and diagramming methods that are akin to

a function structure. A grammar of movements and actions was developed to enable a complete description of the individual actions required by the subsystems.

4.3. SOFTWARE DESIGN DOCUMENT

The Software Design Document (SDD) associated each requirement and specification with particular software and hardware components. It is at this phase of the project where specific hardware was called out for the system, network connections were identified and data exchange protocols were specified. This phase is not that dissimilar from how an SDD might be created for new software with one exception. In this case, no software was written. Instead, the code was reviewed to determine which module was responsible for each requirement. This information was added to the RTM.

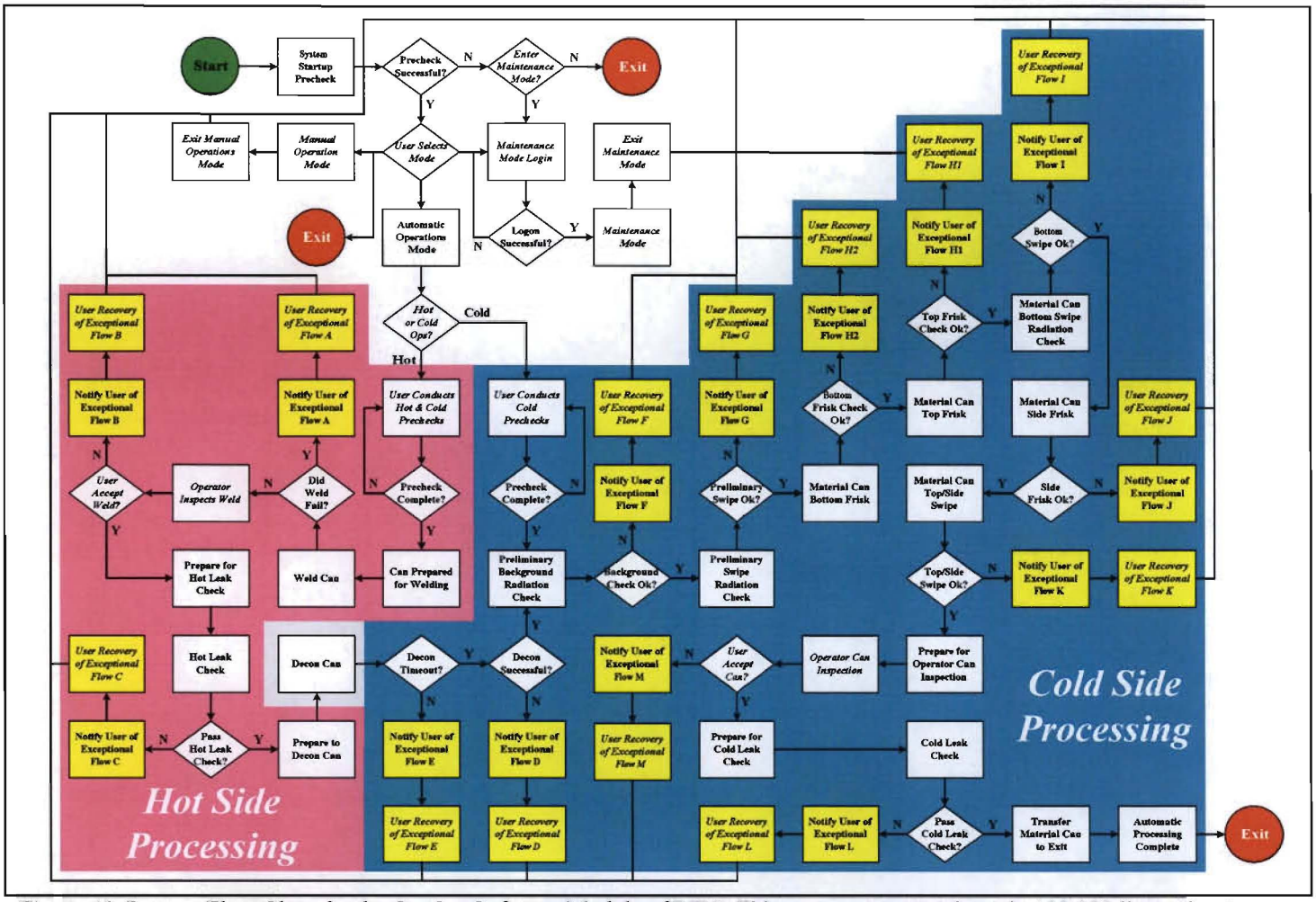

Figure 10. Process Flow Chart for the CanOut Software Module of RIPS. This represents approximately 100,000 lines of code.

4.4. SOFTWARE TESTING

Perhaps the most important documents resulting from the SQA effort are the Software Test Plan (STP) and the Software Test Report (STR). The STP allows the integrated system to be tested to validate that the system performs as expected. While many of the tests for this system were recognized and tested during acceptance testing, about 10-15% of the software requirements escaped testing during the acceptance testing process. Most of the requirements that were missed were relatively minor requirements.

Again, the RTM is the basis for developing the STP although full advantage of the previous testing system acceptance testing was made. Each possible test was evaluated against the tests previously performed. Most of the functionality of the software had been implicitly tested during the integrated system acceptance tests. However, many of the software functions, such as the access control requirements had never been explicitly tested. The access control requirements had been tested accidently, but never deliberately, nor documented.

Surprisingly, lessons were learned during the software testing. Some tests produced unexpected results. One example was a failure message that was reported when a fault was deliberately caused, but the message reported was not what was expected by the test. All of the test results, whether indicating success or failure, were reported in the STR. Erroneous conditions required explanation and an accept-as-is or correct-with-change-order decision.

In the case of the failure message, investigation revealed that the software was reporting a correct error response. Because of limited communication channels, two different fault conditions generated a common error message to the main software system. This hardware decision meant that the software could not distinguish between the two different failure causes. Hence the message that indicated a failure had occurred, but did not indicate that there were two sources. This new information was not only documented in the STR, but also was used to update the Software User Manual (SUM). In this case, the test result was accepted-as-is.

The development of an STP also plays an important role in the configuration management of the software. Once it was established that the existing software was functional; the software was baselined and the configuration frozen. A formal change process is now in place to prevent software and hardware changes without an associated software impact review and a subsequent testing after the change is implemented. This testing uses the tests developed in the STP as the basis for evaluating system modifications. Subsequent testing will generate additional STRs.

Again the methods used to develop the STP and STR are the same regardless of whether the software is existing or new. However, the use of the RTM as the basis for establishing the necessary tests provides a systematic means of demonstrating that the project requirements have been met. The result is a software package backed by demonstrable performance metrics and capable of exceeding customer expectations.

Both the STP and the STR are subjected to a review process that includes system experts, software experts, and the responsible management for the operations and engineering of the system. These reviews provide a valuable check on the rigor of the STP and the conclusions drawn in the STR. The review committees must also approve of any accept-as-is or correct decisions proposed by the responsible system engineers. This rigorous process forces the engineering underlying the system to be completely documented and produces an auditable document record of critical design decisions. This is one of the main goals for SQA.

5. CONCLUSIONS AND LESSONS LEARNED

Even in a tightly controlled and regulated environment such as a DOE nuclear facility, there remains room for improvement. The implementation of a SQA program within the ARIES project required some creative problem solving approaches. Since adequate documentation was not produced during the initial software design and implementation, additional resources were required to reverse engineer the design. These reverse engineering efforts, colloquially called "software archaeology," revealed previously lost design decisions and features that had been lost.

These discoveries have enhanced our present understanding of systems such as RIPS. However, perhaps most importantly, the results of the SQA project have provided a foundation for the evolution of the software within ARIES that did not previously exist. Hardware and software upgrades can now be pursued with a much better understanding of their potential impacts upon the system, which should lead to a more predictable design and integration process. Furthermore, with an established set of software testing procedures in place, any upgrades or modifications to the software can be systematically evaluated before the system is returned to operation.

The relationship between a quality assurance program and design should be a strong one. Many of the features and requirements are similar. Strong formal design methods, when well documented are the backbone of quality assurance documentation. And if that documentation is lacking, reverse engineering methods are the central to the recovery, recreation and rediscovery of what has been lost. Good engineering design practices make quality assurance programs easy to implement and quality assurance programs can nurture good engineering practice.

ACKNOWLEDGEMENTS

This paper is approved for release by Los Alamos National Laboratory under LA-UR-09-XXXX. The assistance and support of Los Alamos National Laboratory and the Division of Engineering at the Colorado School of Mines is greatly appreciated. In particular, the authors would like to recognize Joe Lewis, Sonya Lee, Stan Zygmunt, Mark Swoboda, Max Evans, Bill Everett, and Paul Graham for their contributions to this project and extend their appreciation for their contributions and feedback on this paper. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of Los Alamos National Laboratory.

REFERENCES

ASME. (1997). *Quality Assurance Requirements for Nuclear Facility Applications*, ANSI/ASME NQA-1-1997 Standard, American Society of Mechanical Engineers, New York, New York.

CFR. (2002). *Quality Assurance Criteria*, 10 CFR 830.122, US Government, Washington, DC.

DOE, (2004). *Stabilization, Packaging and Storage of Plutonium-Bearing Materials*, DOE-STD-3013-2004, U.S. Department of Energy, Washington, DC.

DOE, (2005). *Quality Assurance*, DOE Order 414.1C, US Department of Energy, Washington, DC.

Hower, R. (2009). The Software QA/Test Resource Center, http://www.softwareqatest.com/qatfaq1.html, last accessed February 22, 2009.

IEEE. (1994). *IEEE Standard for Software Safety Plans*, IEEE 1228-1994, IEEE Computer Society, New York, New York.

Johnson, R. (2005). "Reverse Engineering and Software Archaeology," *Software Tech News*, **8**:3, pp.7-13.

DOD, (2003). *DOD Standard Practice for System Safety*, MIL-STD-882D, US Department of Defense, Washington, DC.

McKee, S. (2008). "ARIES at 10," *Actinide Research Quarterly*, pp. 1-6, LALP-08-004.

Pyzdek, T. (2003). *Quality Engineering Handbook*, CRC Press, 6000 Broken Sound Parkway, NW, (Suite 300) Boca Raton, FL 33487, USA

Turner, C. and Lloyd, J. (2008). "Automating ARIES," *Actinide Research Quarterly*, pp. 32-5, LALP-08-004.

Turner, C., Harden, T., and Lloyd, J. (2009). "Robotics for Nuclear Material Handling at LANL: Capabilities and Needs," *Proceedings of the 2009 IDETC/CIE Conferences*, San Diego, CA, August 30-September 2, 2009, submitted for review.