Achieving Quality Requirements with Reused Software Components: Challenges to Successful Reuse

Second International Workshop on Models and Processes for the Evaluation of off-the-shelf Components (MPEC’05)

21 May 2005

Donald Firesmith
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
dgf@sei.cmu.edu
Topics

- Introduction
- Reusing Software
- Quality Models and Requirements
- Risks and Risk Mitigation
- Conclusion
Introduction

• When reusing components, many well known problems exist regarding achieving functional requirements.

• Reusing components is an architectural decision as well as a management decision.

• Architectures are more about achieving quality requirements than achieving functional requirements.

• If specified at all, quality requirements tend to be specified as very high level goals rather than as feasible requirements. For example:
  • “The system shall be secure.”
Actual quality requirements (as opposed to goals) are often less negotiable than functional requirements.

Quality requirements are much harder to verify.

Quality requirement achievability and tradeoffs is one of top 10 risks with software-intensive systems of systems. (Boehm et al. 2004)

How can you learn what quality requirements were originally used to build a reusable component?

What should architects know and do?
Reusing Software

- Scope of Reuse
- Types of Reusable Software
- Characteristics of Reusable Software
Scope of Reuse

- Our subject is the development of software-intensive systems that incorporate some reused component containing or consisting of software.
- We are *not* talking about developing software for reuse in such systems (i.e., this is not a ‘design for reuse’ discussion).
- The scope is all reusable software, not just COTS software.
Types of Reusable Software

- Non-developmental Item (NDI) components with SW come in many forms:
  - COTS (Commercial Off-The-Shelf)
  - GOTS (Government Off-The-Shelf)
  - GFI (Government Furnished Information)
  - GFE (Government Furnished Equipment)
  - OSS (Open Source Software)
  - Shareware
  - Legacy (for Ad Hoc Reuse)
  - Legacy (for Product Line)
- They have mostly similar characteristics.
- Differences more quantitative than qualitative
Characteristics of Reusable SW

• *Not* developed for use in applications / systems with your exact requirements. For example, they were built to different (or unknown):
  
  • **Functional requirements** (operational profiles, feature sets / use cases / use case paths)
  • **Quality requirements** (capacity, extensibility, maintainability, interoperability, performance, safety, security, testability, usability)
  • **Data requirements** (types / ranges / attributes)
  • **Interface requirements** (syntax, semantics, protocols, state models, exception handling)
  • **Constraints** (architecture compatibility, regulations, business rules, life cycle costs)
Characteristics of Reusable SW

- Intended to be used as a blackbox
- Hard, expensive, and risky to modify and maintain
- The following may not be available, adequate, or up-to-date:
  - Requirements Specifications
  - Architectural Documents
  - Design Documentation
  - Analyses
  - Source code
  - Test code and test results
- Lack of documentation is especially common with COTS SW.
Characteristics of Reusable SW

- Maintained, updated, and released by others according to a schedule over which you have no control
- Typically requires licensing, which may involve major issues
- Often needs a wrapper or an adaptor:
  - Must make trade-off decision that developing glue code is worth the cost and effort of using the component
Component Quality Requirements

- Often overlooked
- Typically poorly engineered:
  - Not specified at all
  - Not specified properly (incomplete, ambiguous, incorrect, infeasible)
    - Specified as ambiguous, high-level quality goals rather than as verifiable quality requirements
- Must be analyzed and specified in terms of corresponding quality attributes
- Requires quality model to do properly
Quality Models

- **Quality Model** – a hierarchical model (i.e., a layered collection of related abstractions or simplifications) for formalizing the concept of the quality of a system in terms of its:
  - **Quality Factors** – high-level characteristics or attributes of a system that capture major aspects of its quality (e.g., interoperability, performance, reliability, safety, and usability)
  - **Quality Subfactors** – major components of a quality factor or another quality subfactor that capture a subordinate aspect of the quality of a system (e.g., throughput, response time, jitter)
  - **Quality Criteria** - specific descriptions of a system that provide evidence either for or against the existence of a specific quality factor or subfactor
  - **Quality Measures** – gauges that quantify a quality criterion and thus make it measurable, objective, and unambiguous (e.g., transactions per second)
Quality Model

Quality Model

Quality Factor

provides evidence for existence of

Quality Subfactor

provides evidence for existence of

is measured using

Quality Measure

measures

System-Specific Quality Criterion

describes quality of

System
Quality Factors

Quality Model

Quality Factor

- Capacity
- Correctness
- Dependability
- Interoperability
- Performance
- Utility

- Availability
- Defensibility
- Reliability
- Robustness

- Safety
- Security
- Survivability
Quality Requirements

Quality requirement – a mandated combination of quality criterion and quality measure threshold or range

Describes quality of system

Provides evidence for existence of system-specific quality criterion

Provides evidence for existence of quality subfactor

Quality model

Quality factor

Quality subfactor

Quality measure with threshold

System-specific quality criterion

Quality requirement

© 2005 Software Engineering Institute
Some Important Quality Factors

• All quality factors may have requirements that reusable components must meet.
• Today, we will briefly consider the following:
  • Availability
  • Capacity
  • Performance
  • Reliability
  • Robustness
  • Safety
  • Security
  • Testability
Availability

- **Availability** – the proportion of the time that an application or component functions properly (and thus is available for performing useful work)
  - *Measured/Specified* as the average percent of time that one or more functions/features/use cases/use case paths [must] properly operate without scheduled or unscheduled downtime under given normal conditions.
- Becomes exponentially more difficult and expensive as required availability increases (99% vs. 99.999%)
- Many possible [inconsistent] architectural mechanisms
- Requires many long-running tests to verify
- SW dependencies makes estimation of overall availability from component availabilities difficult, even if known
Capacity

• **Capacity** - the maximum number of things that an application or component can successfully handle at a single point in time
  • *Measured/Specified* in terms of number of users, number of simultaneous transactions, number of records stored, etc.
• Cannot be indefinitely large
• Solutions require both hardware and software architectural decisions that may be inconsistent with those of the reusable components
• Reasonably straight-forward to test if required capacity is achieved, but not actual system capacity
Performance

- **Performance** – the execution time of a function of an application or component. Subfactors include:
  - **Determinism** – the extent to which events and behaviors are deterministic and can be precisely and accurately predicted and scheduled
  - **Jitter** – the variability of the time interval between an application or component’s periodic actions
  - **Latency** – the time that an application or component takes to execute specific tasks (e.g., system operations and use case paths) from end to end
  - **Response Time** – the time that an application or component takes to *initially* respond to a client request for a service or to be allowed access to a resource
  - **Throughput** – the number of times that an application or component is able to complete an operation or provide a service in a specified unit of time
Performance 2

• Measured and specified in many different ways
• Not all functions need high performance
• Although certain performance subfactors are vital for safety and security certification and for real time scheduling analysis, these performance subfactors are rarely considered by product suppliers and other developers
• Architectural mechanisms include real-time OS, cyclic executive, no automatic garbage collection, repeated hardware, etc.
• Requires significant analysis and testing to verify
Reliability

• **Reliability** – the degree to which an application or component continues to function properly without failure under *normal* conditions or circumstances

• *Measured/specified* as the:
  - Mean time between failures (MTBF) during a given time period under a given operational profile, whereby MTBF is defined as the average period of time that the application continues [shall continue] to function correctly without failure under stated conditions.
  - [Maximum permitted] number of failures per unit time

• Becomes exponentially more difficult and expensive as required reliability increases

• Many possible [inconsistent] architectural mechanisms

• Requires many long-running tests to verify
Robustness

- **Robustness** – the degree to which an application or component continues to function properly under abnormal conditions or circumstances during a given time period:
  - **Environmental tolerance** (e.g., vibration or power)
  - **Failure tolerance** (fail safety, fail softness – degraded mode)
  - **Fault tolerance** (presence of defects/bugs)
  - **Error tolerance** (erroneous input)
- Becomes exponentially more difficult and expensive as required robustness increases
- Many possible [inconsistent] architectural mechanisms (e.g., fault detection by heartbeat vs. ping/echo vs. exception)
- Requires many difficult and expensive tests to verify
- SW dependencies makes estimation of overall robustness from component robustness difficult, even if known
Safety

- **Safety** is the degree:
  - Of freedom from:
    - Accidental (unintentional) harm to valuable assets
    - Safety *incidents* (*accidents* and *near misses*) that can cause accidental harm
    - *Hazards* that may cause safety incidents
    - Safety *risks* (max. harm times probability)
  - To which the following exist:
    - *Prevention* of accidental harm
    - *Detection* of safety incidents
    - *Reaction* to safety incidents
    - *Adaptation* to avoid accidental harm in the future
Safety

- Safety is becoming more and more critical as more and more systems have safety ramifications.
- Reusable software (e.g., COTS) often does not address safety.
- Safety Integrity Levels (SILs) in the requirements require proportionate Safety Evidence Assurance Levels (SEALs) regarding the development of components to achieve certification:
  - Architecture as well as design, coding, and testing.
Safety

- Reused components have:
  - Different or nonexistent safety requirements
  - Different, incompatible, or nonexistent safeguards
- Poor (inappropriate, incomplete, missing) requirements are the cause of roughly 40% of accidents.
- Therac-25 (6 deaths) and Ariane-5 ($500 million) examples of accidents due to reuse
Security

• **Security** is the *degree*:
  • Of freedom from:
    • *Malicious* harm to valuable assets from attackers
    • Security incidents (successful *attacks*, unsuccessful attacks, *probes*) that can cause malicious harm
    • *Threats* that may cause security incidents
    • Security *risks* (max. harm times probability)
  • To which the following exist:
    • *Prevention* of malicious harm
    • *Detection* of security incidents
    • *Reaction* to security incidents
    • *Adaptation* to avoid security problems in the future
Security

- Security is becoming more and more critical as more and more systems have security ramifications (e.g., private data, nonrepudiation needs, valuable assets)
- Reusable software (e.g., COTS) often does not adequately address security
- Security must be architected into systems, not added on afterwards
- Reused components have:
  - Different or nonexistent security requirements
  - Different, nonexistent, or incompatible security controls
Testability

• **Testability** – the degree to which an application or component facilitates the creation and execution of successful tests

• A function of:
  • Observability
  • Controllability

• Directly at odds with security

• Typically low with blackbox components not delivered with test cases and test harnesses

• Limited to blackbox component testing, system integration testing, system testing, and **quality requirements testing**
Summary of Risks

• Reusable component is built to different quality requirements than current system.
• Components often have incompatible architectural approaches to support achieving important quality requirements.
• Difficult and expensive to verify achievement of quality requirements by reusable components
• Difficult to obtain safety and security certifications for reused components and resulting systems
• Glue code is neither always adequate nor inexpensive.
Risk Mitigation

- Do not assume that reuse will necessarily be cheaper, faster, or better.
- Negotiate quality requirements with ranges as well as hard thresholds if practical.
- Demand credible evidence from supplier to support reusability analysis.
- Talk to users of the reusable components to learn from their experiences.
Risk Mitigation

• Do not overlook quality requirements / attributes when assessing the appropriateness of “reusable” components.

• Perform major reuse readiness assessment of the reusable components that includes verification of quality requirements:
  • Technical analysis
  • Prototyping
  • Testing

• Plan for the significant cost (schedule, effort, expense) of performing a real readiness assessment.
Conclusion

If you are not concerned, you have probably not paid sufficient attention.