Software Quality Attributes and Architecture Tradeoffs

Mario R. Barbacci

Software Engineering Institute
Carnegie Mellon University
Pittsburgh PA 15213
Sponsored by the U.S. Department of Defense

Seminar Objective

To describe a variety of software quality attributes (e.g., modifiability, security, performance, availability) and methods to analyze a software architecture's fitness with respect to multiple quality attribute requirements.

Carnegie Mellon
Software Engineering Institute

Software Product Characteristics

There is a triad of user oriented product characteristics:

- quality
- cost
- schedule

"Software quality is the degree to which software possesses a desired combination of attributes."

[IEEE Std. 1061]

Carnegie Mellon
Software Engineering Institute

Effect of Quality on Cost and Schedule - 1

Cost and schedule can be predicted and controlled by mature organizational processes.

However, process maturity does not translate automatically into product quality.

Poor quality eventually affects cost and schedule because software requires tuning, recoding, or even redesign to meet original requirements.

If the technology is lacking, even a mature organization will have difficulty producing products with predictable performance, dependability, or other attributes.

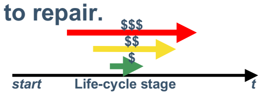
For less mature organizations, the situation is even worse:

> "Software Quality Assurance is the least frequently satisfied level 2 KPA among organizations assessed at level 1",
>
> From Process Maturity Profile of the Software Community 2001 Year End Update, http://www.sei.cmu.edu/sema/profile.html
>
> NOTE: The CMM Software Quality Assurance Key Process Area (KPA) includes both process and product quality assurance.

Quality requires mature technology to predict and control attributes

Effect of Quality on Cost and Schedule - 2

The earlier a defect occurs in the development process, if not detected, the more it will cost to repair.
The longer a defect goes undetected the more it will cost to repair.



[Barry Boehm et al, "Characteristics of Software Quality", North-Holland, 1978.
Watts Humphrey, "A Discipline for Software Engineering", Addison Wesley, 1995.]

Carnegie Mellon
Software Engineering Institute

Effect of Quality on Cost and Schedule - 3

The larger the project, the more likely it will be late due to quality problems:

Project outcome	Project size in function points			
	<100	100-1K	1K-5K	>5K
Cancelled	3%	7%	13%	24%
Late by > 12 months	1%	10%	12%	18%
Late by > six months	9%	24%	35%	37%
Approximately on time	72%	53%	37%	20%
Earlier than expected	15%	6%	3%	1%

[Caspers Jones, *Patterns of large software systems: Failure and success*, Computer, Vol. 28, March 1995.]

From C.Jones 95:

"Software management consultants have something in common with physicians: both are much more likely to be called in when there are serious problems rather than when everything is fine. Examining large software systems -- those in excess of 5,000 function points (which is roughly 500,000 source code statements in a procedural programming language such as Cobol or Fortran) -- that are in trouble is very common for management consultants. Unfortunately, the systems are usually already late, over budget, and showing other signs of acute distress before the study begins. The consultant engagements, therefore, serve to correct the problems and salvage the system -- if, indeed, salvaging is possible."

"From a technical point of view, the most common reason for software disasters is poor quality control. Finding and fixing bugs is the most expensive, time-consuming aspect of software development, especially for large systems. Failure to plan for defect prevention and use pretest defect-removal activities, such as formal inspections, means that when testing does commence, the project is in such bad shape that testing tends to stretch out indefinitely. In fact, testing is the phase in which most disasters finally become visible to all concerned. When testing begins, it is no longer possible to evade the consequences of careless and inadequate planning, estimating, defect prevention, or pretest quality control."

Carnegie Mellon
Software Engineering Institute

Software Quality Attributes

There are alternative (and somewhat equivalent) lists of quality attributes. For example:

IEEE Std. 1061	ISO Std. 9126	MITRE Guide to Total Software Quality Control	
Efficiency	Functionality	Efficiency	Integrity
Functionality	Reliability	Reliability	Survivability
Maintainability	Usability	Usability	Correctness
Portability	Efficiency	Maintainability	Verifiability
Reliability	Maintainability	Expandability	Flexibility
Usability	Portability	Interoperability	Portability
		Reusability	

Carnegie Mellon
Software Engineering Institute

Quality Factors and Sub-factors

IEEE Std. 1061 subfactors:

Efficiency
- Time economy
- Resource economy

Functionality
- Completeness
- Correctness
- Security
- Compatibility
- Interoperability

Maintainability
- Correctability
- Expandability
- Testability

Portability
- Hardware independence
- Software independence
- Installability
- Reusability

Reliability
- Non-deficiency
- Error tolerance
- Availability

Usability
- Understandability
- Ease of learning
- Operability
- Comunicativeness

From IEEE Std. 1061:

“Software quality is the degree in which software possesses a desired combination of quality attributes. The purpose of software metrics is to make assessments throughout the software life cycle as to whether the software quality requirements are being met.

The use of software metrics reduces subjectivity in the assessment and control of software quality by providing a quantitative basis for making decisions about software quality.

However, the use of metrics does not eliminate the need for human judgment in software assessment. The use of software metrics within an organization is expected to have a beneficial effect by making software quality more visible.”

Approaches to Quality Attributes

Performance — from the tradition of hard real-time systems and capacity planning.
Dependability — from the tradition of ultra-reliable, fault-tolerant systems.
Usability — from the tradition of human-computer interaction and human factors.
Safety — from the tradition of hazard analysis and system safety engineering.
Security — from the traditions of the government, banking and academic communities.
Integrability and Modifiability — common across communities.

There are different schools (opinions, traditions) concerning the properties of critical systems and the best methods to develop them.

These techniques have evolved in separate communities, each with its own vocabulary and point of view.

There are no metrics or methods for evaluation applicable to all attributes.

Different communities use different models and parameters for evaluation of attributes:

- models are not necessarily mathematical formulas
- models can be based on expert opinions on how to evaluate a quality attribute

Attributes values are not absolute e.g., a system is more or less secure depending on the threat.

Attribute evaluations must be performed within specific

Performance

"Performance. The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage."
[IEEE Std. 610.12]

"Predictability, not speed, is the foremost goal in real-time-system design"

[J.A. Stankovic, "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems," *IEEE Computer*, Volume 21, Number 10, October 1988.]

A misnomer is that performance equates to speed; that is, to think that poor performance can be salvaged simply by using more powerful processors or communication links with higher bandwidth.

Faster might be better, but for many systems faster is not sufficient to achieve timeliness. This is particularly true of real-time systems

As noted in [Stankovic 88], the objective of "fast computing" is to minimize the average response time for some group of services, whereas the objective of real-time computing is to meet individual timing requirements of each service.

- Hardware mechanisms such as caching, pipelining and multithreading, which can reduce average response time, can make worst-case response times unpredictable.
- In general, performance engineering is concerned with predictable performance whether its worst-case or average-case performance. Execution speed is only one factor.

Performance Taxonomy

- Performance
  - Concerns
    - Latency
    - Throughput
    - Capacity
    - Modes
  - Factors
    - Environment
    - System
  - Methods
    - Synthesis
    - Analysis

Concerns

•Latency - time to respond to a specific event

•Throughput - number of events responded to over an interval of time

•Capacity - demand that can be placed on the system while continuing to meet latency and throughput requirements

•Modes - changes in demands and resources over time

Factors

•Environment (external) factors - how much of a resource is needed

•System (internal) factors - available resources and policies

Methords

•Synthesis methods - normal software development steps with explicit attention to performance

•Analysis methods - techniques used to evaluate system performance

Dependability

"Availability. The degree to which a system or component is operational and accessible when required for use."

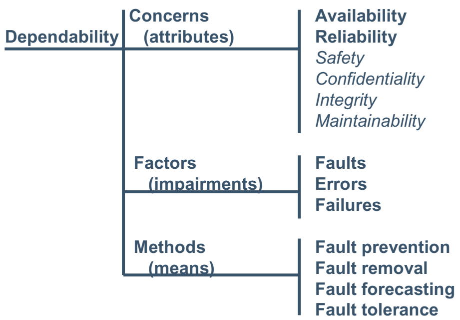
[IEEE Std. 610.12]

"Dependability is that property of a computer system such that reliance can justifiably be placed on the service it delivers"

[J.C. Laprie (ed.) "Dependability: Basic Concepts and Terminology", Volume 5 of Dependable Computing and Fault-Tolerant Systems. Springer-Verlag, February 1992.].

Carnegie Mellon
Software Engineering Institute

Dependability Taxonomy



Although the dependability community includes safety, confidentiality, integrity, and maintainability as dependability concerns, these concerns have traditionally been the focus of other communities, sometimes with different approaches.

Concerns

- •Availability - readiness for usage
- •Reliability - continuity of service
- •Safety - non-occurrence of events with catastrophic consequences on the environment
- •Confidentiality - non-occurrence of unauthorized disclosure of information
- •Integrity - non-occurrence of improper alterations of information
- •Maintainability - aptitude to undergo repairs and evolution

Factors

- •Faults - the adjudged or hypothesized event that causes an error
- •Errors - a system state that is liable to lead to a failure if not corrected
- •Failures - a system departs from intended behavior

Methods

- •Fault prevention - covered by good software engineering practices
- •Fault removal - removing faults during development
- •Fault forecasting - predicting probabilities and sequences of undesirable events during development
- •Fault tolerance - detecting and correcting latent errors before they become effective during execution

Carnegie Mellon
Software Engineering Institute

Modifiability

Modifiability encompasses two aspects:

"Maintainability. (1) The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. (2) The ease with which a hardware system or component can be retained in, or restored to, a state in which it can perform its required functions."

"Flexibility: The ease with which a system or component can be modified for use in applications or environments other than those for which is was specifically designed."

[IEEE Std. 610.12]

Carnegie Mellon
Software Engineering Institute

Modifiability Taxonomy

- Modifiability
  - Concerns
    - Extensibility
    - Simplification
    - Restructuring
    - Time to deploy
    - Functional scalability
    - Functional flexibility
  - Factors
    - Component complexity
    - Component size
    - Scope of modification
  - Methods
    - Modularity
    - Encapsulation
    - Software practice

Concerns

- Extensibility - adding/enhancing/repairing functionality
- Simplification - streamlining/simplifying functionality
- Restructuring - rationalizing services, modularizing/optimizing/creating reusable components
- Time to deploy - time taken from specifying a requirement for new capability to the availability of that capability
- Functional scalability - ability to scale both up/down in terms of users, system throughput, availability, etc.
- Functional flexibility - turning an existing capability to new uses, new locations, or unforeseen situations

Carnegie Mellon
Software Engineering Institute

Usability

"Usability. The ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component."
[IEEE Std. 610.12]

Usability is a measure of how well users can take advantage of some system functionality.

Usability is different from utility, a measure of whether that functionality does what is needed.

Carnegie Mellon
Software Engineering Institute

Usability Taxonomy

- Usability
  - Concerns
    - Learnability
    - Efficiency
    - Memorability
    - Errors
    - Satisfaction
  - Factors
    - Tradeoffs
    - Categories of users
  - Methods
    - Usability Eng. lifecycle
    - Lifecycle stage methods
    - Architecture mechanisms

Jakob Nielsen, "Usability Engineering", Academic Press, AP Professional, Cambridge, MA, 1993.

Concerns

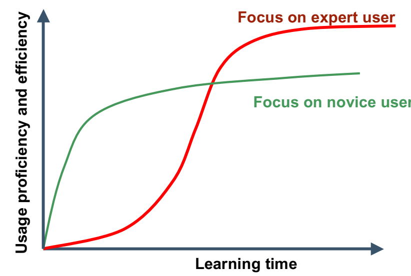
- Learnability - easy to learn; novices can readily start getting some work done
- Efficiency - efficient to use; experts have a high level of productivity
- Memorability - easy to remember; casual users do not have to learn everything every time
- Errors - low error rate; users make few errors and can easily recover from them
- Satisfaction - pleasant to use; discretionary/optional users are satisfied when and like it

Factors

- Tradeoffs - depending on the situation, usability might be

Carnegie Mellon
Software Engineering Institute

Factors: Learning Time Tradeoffs



[J. Nielsen, Usability Engineering, Fig. 2]

Learning curves for systems that focus on novice or expert users. J. Nielsen, Usability Engineering, Fig. 2.

- It is not the case that a system is either easy to learn but inefficient or hard to learn and efficient. A user interface can provide multiple interaction styles:
  - users start by using a style that is easy to learn
  - later move to a style that is efficient
- Learnable systems have a steep rise at the beginning and allow users to reach a reasonable level of proficiency within a short time.

Most systems have learning curves that start out with the user being able to do very little at time zero, when they start using it.

Some systems are meant to be used only once and need to have zero learning time:

- Walk-up-and-use (e.g., museum information systems, car-rental directions to hotels)
- Systems that require reading instructions (e.g., installation programs, disk formatting routines, tax preparation programs that change every year)

Carnegie Mellon
Software Engineering Institute

Factors: Accelerator Tradeoffs

Accelerators or shortcuts are user interface elements that allow the user to perform frequent tasks quickly, e.g.:

- function keys
- command name abbreviations
- double-clicking
- *etc.*

System can push users to gain experience:

- expert shortcuts in the novice menus
- On-line help
- analyze users' actions and offer alternatives

Users normally don't take the time to learn a complete interface before using it; they start using it as soon as they have learned to do "enough" -- measures of learnability should allow for this and not test for complete mastery of the interface.

Carnegie Mellon
Software Engineering Institute

Factors: Intentional Deficiency Tradeoffs

Efficiency might be sacrificed to avoid errors, e.g.:

- asking extra questions to make sure the user is certain about a particular action

Learnability might be sacrificed for security, e.g.:

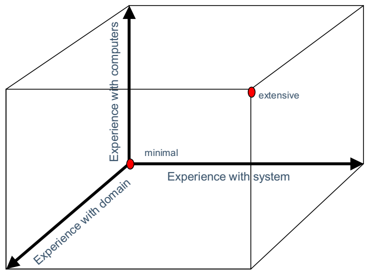
- not providing help for certain functions e.g., not helping with useful hints for incorrect user IDs or passwords

Learnability might be sacrificed by hiding functions from regular users, e.g.:

- hiding reboot buttons/commands in a museum information system

Carnegie Mellon
Software Engineering Institute

Factors: Categories of Users



[J. Nielsen, Usability Engineering, Fig. 3]

Dimensions in which users' experience differs, *J. Nielsen, Usability Engineering, Fig. 3*

- •Experience with the specific user interface is the dimension that is normally referred to when discussing user expertise.
  - •In reality most people do not acquire comprehensive expertise in all parts of a system, no matter how much they use it.
  - •Complex systems have so many features that a given user only makes extensive use of a subset
  - •An expert could be a novice on parts of the system not normally used by that user and need access to help for those parts of the interface

- •Experience with computers also has an impact on user interface design. The same utility might have to be provided with two different interfaces
  - •Utilities for system administrators vs. home computer users (e.g., disk defragmentation
  - •Experience with other applications "carries over" since the users have some idea of what features to look for and how the computer normally deals with various situations (e.g., look for a "sort" function on a new word processor because is common in spreadsheets and databases)
  - •Programming experience determines to what extent the user can customize the interface using macro languages in a way that is maintainable and modifiable at a later date
  - •In addition, programmers' productivity can range by a factor of 20!

Carnegie Mellon
Software Engineering Institute

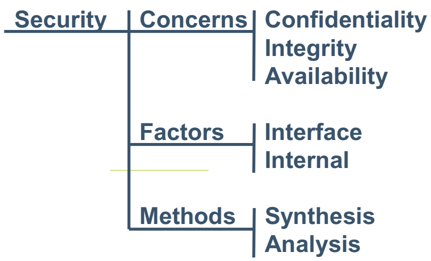
Security

"Secure systems are those that can be trusted to keep secrets and safeguard privacy."

[J. Rushby, *Critical System Properties: Survey and Taxonomy*, SRI International, Technical Report CSL-93-01, May 1993]

Carnegie Mellon
Software Engineering Institute

Security Taxonomy



Historically, there have been three main areas which have addressed security:

•government/military applications

•banking and finance, and

•academic/scientific applications.

In each case, different aspects of security were stressed, and the definition of individual security attributes depended upon the stressed security aspects.

Concerns - The traditional main concern of security was unauthorized disclosure of information. Secondary concerns were the ability to protect the integrity of information and prevent denial of service:

•Confidentiality - data and processes are protected from unauthorized disclosure

•Integrity - data and processes are protected from unauthorized modification

•Availability - data and processes are protected from denial of service to authorized users

Factors

•Interface (external) factors - security features available to the user or between systems

Carnegie Mellon
Software Engineering Institute

From Security to Survivability

Large-scale, distributed systems cannot be totally isolated from intruders - no amount of “hardening” can guarantee that systems will be invulnerable to attack.

We design buildings to deal with environment stress such earthquakes as well an intentional attacks such as a break-in.

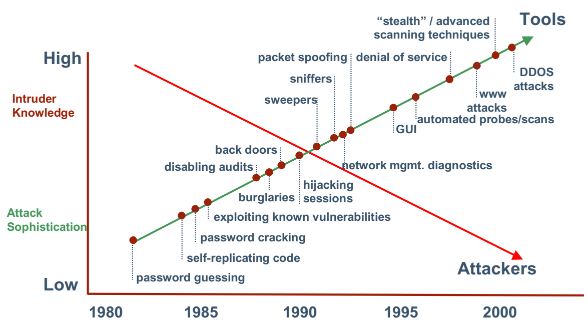
We need to apply a similar approach to software where the faults are malicious attacks.

Extend security to include the ability to maintain some level of service in the presence of attacks.

Success is measured in terms of the success of mission rather than in the survival of any specific system or component.

Carnegie Mellon
Software Engineering Institute

Dynamic Changes in Attacks



 page 25

Figure 1.2 in J.H. Allen, et al., "State of the Practice of Intrusion Detection Technologies," CMU/SEI-99-TR-028, Software Engineering Institute, Carnegie Mellon University, 1999.

"In the 1980s, intruders were the system experts. They had a high level of expertise and personally constructed methods for breaking into systems. Use of automated tools and exploit scripts was the exception rather that the rule. Today absolutely anyone can attack a network - dues to the widespread and easy availability of intrusion tools and exploit scripts that duplicate known methods of attack."

Carnegie Mellon
Software Engineering Institute

Attacks are *faults*

There are prevention, detection and recovery techniques:

- the threat assessment has to include assumptions about the attacker
- system responses should take advantage of known attack patterns (intrusion aware design)
- cost/benefit and tradeoffs analysis requires knowledge of the impact of the attacks

 page 26

Solutions should include prevention measures, detection of attacks, and describe how the system should react to such events.

Carnegie Mellon
Software Engineering Institute

Survivability Taxonomy

- Survivability
  - Concerns (attacks)
    - Resistance to attacks
    - Recognition of attacks
    - Recovery after attack
  - Factors (strategies)
    - Strategies for repelling attacks
    - Strategies for detecting attacks and evaluating damage
    - Strategies for limiting damage, restoring information/functionality within time constraints, and restoring full services
  - (attack) Methods
    - Direct (internet/network) attacks
    - Social engineering attacks
    - Insider attacks
    - Trusted sites/agencies attacks

N.R. Mead et al, *Survivable Network Analysis Method* (CMU/SEI-2000-TR-013), Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, September 2000.
http://www.sei.cmu.edu/publications/documents/00.reports/00tr013.html

Carnegie Mellon
Software Engineering Institute

Safety

To paraphrase the definition of dependability:

"Dependability is that property of a computer system such that reliance can justifiably be placed on the services it delivers."
[J.C. Laprie, 1992]

Safety is that property of a computer system such that reliance can justifiably be placed in the absence of accidents.

Safety vs. Dependability

Safety is not the same as dependability:

- **dependability is concerned with the occurrence of failures, defined in terms of internal consequences (services are not provided)**
- **safety is concerned with the occurrence of accidents or mishaps, defined in terms of external consequences (accidents happen)**

The difference of intents between safety and dependability — "good things (services) must happen" vs. "bad things (accidents) must not happen" — gives rise to the following paradox:

•if the services are specified incorrectly, a system can be dependable but unsafe — for example, an avionics systems that continues to operate under adverse conditions yet directs the aircraft into a collision course

•a system might be safe but undependable — for example, a railroad signaling system that always fails-stops

Safety Taxonomy

- **Safety**
  - **Concerns**
    - **Interaction complexity**
    - **Coupling strength**
  - **Factors (impairments)**
    - **Hazards (conditions)**
    - **Mishaps (consequences)**
  - **Methods**
    - **Hazard identification**
    - **Hazard analysis**
    - **Implementation methodologies**
    - **Implementation mechanisms**

Concerns:

•Interaction complexity - the extent to which the behavior of one component can affect the behavior of other components

•Component coupling - the extent to which there is flexibility in the system to allow for unplanned events

Factors:

•Hazards - conditions (i.e., state of the controlled system) that can lead to a mishap

•Mishaps - unplanned events that result in death, injury, illness, damage or loss of property, or environment harm

Methods:

•Hazard identification - Develop a list of possible system hazards before the system is built

•Hazard analysis - identifies risk mitigation steps after identifying a hazard

•Implementation methodologies - Avoid introduction of errors during the development process and, if unavoidable, detect and correct them during operation.

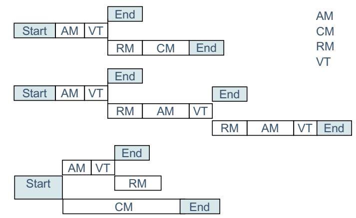
•Implementation mechanisms - Prescribe or disallow specific states or sequences of events

Combinations of Methods

Methods can be combined to achieve the quality benefits while reducing effort.

Methods can come from different quality attributes.

Improving Performance



AM	Aggressive mechanism
CM	Conservative mechanism
RM	Recovery mechanism
VT	Verification test

Variations in optimistic protocols
[F. Pedone, "Boosting System Performance with Optimistic Distributed Protocols," *IEEE Computer*, Volume 34, Number 12, December 2001.]

From Pedone, 2001:

"Optimistic protocols aggressively execute actions based on best-case system assumptions. When the optimistic assumptions hold, the protocol executes far more efficiently than a pessimistic protocol. However, when the assumptions do not hold, the optimistic protocol may execute more slowly than a pessimistic protocol because repairing the incorrect actions can impose performance penalties. Using optimistic protocols unquestionably involves trade-offs, but if a protocol is well designed and the optimistic assumptions hold frequently enough, the gain in performance outweighs the overhead of repairing actions that execute incorrectly."

Optimistic protocols are techniques to increase performance that share some elements with Recovery Blocks and (Dual) redundancy.

Pedone's article offers examples of various distributed protocols: Optimistic atomic broadcast, Optimistic virtual synchrony, Optimistic two-phase commit, Distributed optimistic concurrency control, and Optimistic validation of electronic tickets.

For each of these examples, Pedone defines the optimistic assumption, the aggressive mechanisms, the verification test, the recovery mechanisms, and the conservative mechanism.

Carnegie Mellon
Software Engineering Institute

Improving Fault Removal

Fault removal techniques like (exhaustive) formal verification might not be practical while (incomplete) testing might miss detection of some errors:

- **both techniques can be combined provided there are consistency checks between the model and the actual code**
- **model verification might suggest areas to test and testing some execution paths might suggest changes to the model**
- **in addition, fault avoidance techniques (e.g., coding rules) might enhance coverage of both verification and testing**

[Sharygina, N., and Peled, D., "A Combined Testing and Verification Approach for Software Reliability", Springer-Verlag Lecture Notes in Computer Science, pages 611-628, 2001.]

For an example of combination of the three methods see:

•N. Sharygina, J. C. Browne and R. Kurshan, "A Formal Object-Oriented Analysis for Software Reliability:Design for Verification", *Proceedings of The European Joint Conferences on Theory and Practice of Software (ETAPS) 2001, Springer-Verlag Lecture Notes in Computer Science (LNCS) 2029, Pages 318-332, 2001.*

•Sharygina, N., and Peled, D., "A Combined Testing and Verification Approach for Software Reliability", *Proceedings of Formal Methods Europe (FME) 2001, Springer-Verlag Lecture Notes in Computer Science (LNCS) 2021, pages 611-628, 2001.*

Carnegie Mellon
Software Engineering Institute

Improving Fault Tolerance

Most methods focus on "errors of commission" and not on "errors of omission":

- detecting departure from intended behavior (error of commission) offers cues that help error detection
- detecting that a system does not do what it is not supposed to do (error of omission) is more difficult
- however, there is empirical evidence that exception failure, an error of omission, causes 2/3 of system failures!

[F. Cristian, "Exception Handling and Tolerance of Software Faults," pages 81-107 in Software Fault Tolerance, M.R. Lyu, (ed.), Wiley, Chichester, 1995.]

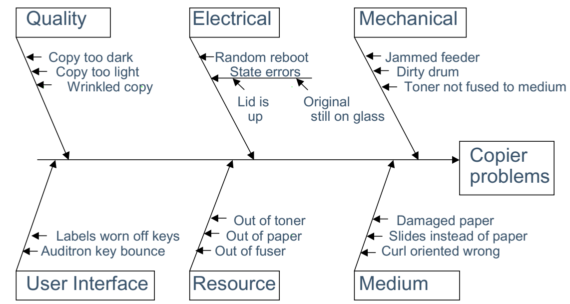
Testing for Exception Failure

Hazard analysis techniques that can be used to develop check-lists that would improve testing for exception handling errors:

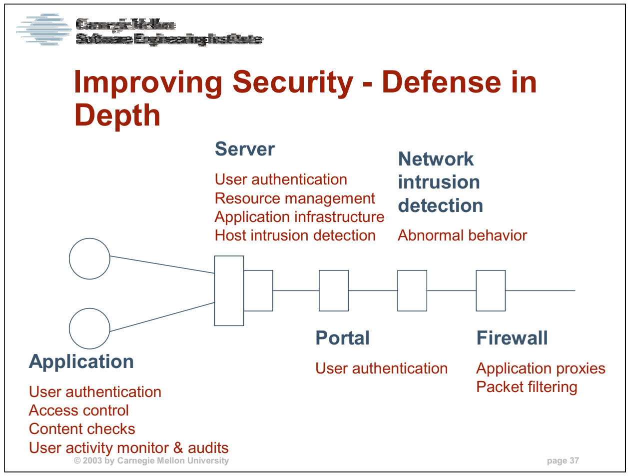
- check-lists must be system specific or context dependent, otherwise the list would be too long and difficult to use
- a fishbone diagram provides useful visual cues

[R.A. Maxion, and R.T. Olszewski, “Eliminating Exception Handling Errors with Dependability Cases: A Comparative, Empirical Study,” IEEE Transactions on Software Engineering, Volume 26, Number 9, September 2000.]

Example Fishbone Diagram



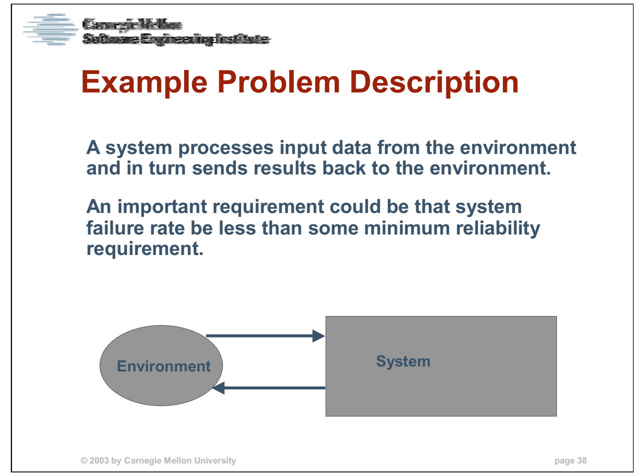
[Figure 3 in Maxion 2002]



Defense in depth is a popular approach to providing information assurance. In this example,

- A firewall can provide protection by controlling the services and machines can be externally accessed. A firewall can also restrict which external IP addresses can access internal resources.
- Network intrusion detection software can monitor network traffic for abnormal behavior.
- A server can provide additional protection in terms of host-based intrusion detection to monitor general user activity on the sever. A standard technique is to provide only the minimal set of services required (say no ftp, telnet, mail) so that an attacker is limited in the techniques they can apply.
- Finally the application can also provide a level of defense. Many attacks such as an email virus are attacks which exploit the data. So the application which understands the details of the data exchange and the data content is in a good position to mitigate such attacks. The email system is in the best position to monitor email attachments for viruses.

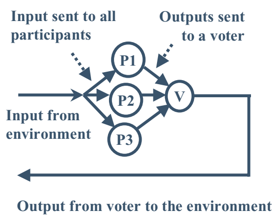
User authentication can be applied either an a portal to control access to the entire site or on a server or application.



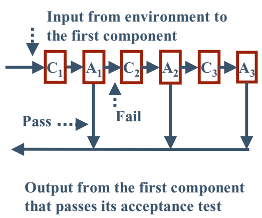
Carnegie Mellon
Software Engineering Institute

Approaches to Dependability

Triple-Modular Redundancy (TMR)



Recovery Blocks (RB)



 page 39

In Triple-Modular-Redundancy (TMR) three components perform redundant (but not necessarily identical) computations. A voter chooses the "correct" result as the output from the system

- if the voter detects a faulty participant, it ignores that participant from then on and continues operating with the remainder
- if the voter can not make a decision, the voter (and system) fail-stops

In Recovery Blocks (RB) multiples components perform computations in sequence. After each computation is completed, an acceptance test is conducted and if the component is deemed to have worked properly, the results are accepted. If the component is deemed to have failed, the original state is recovered and a different component starts the computation. If none of the components passes their acceptance tests, the system has failed and it stops.

Carnegie Mellon
Software Engineering Institute

TMR Dependability Analysis

The reliability of a TMR system is:

$$R_{TMR}(t) = 3e^{-2\lambda t} - 2e^{-3\lambda t}$$

The Mean-Time-To-Failure of a TMR system without repairs is:

$$MTTF_{TMR} = \left(\int_0^\infty 3e^{-2\lambda t}dt - \int_0^\infty 2e^{-3\lambda t}dt\right) = \frac{3}{2\lambda} - \frac{2}{3\lambda} = \frac{5}{6\lambda}$$

The MTTF of a TMR system with repairs is:

$$MTTF_{TMR} = \frac{5}{6\lambda} + \frac{\mu}{6\lambda^2}$$

λ and μ are the failure and repair rates, respectively.

 page 40

For discussion of the reliability and MTTF equations, see [Siewiorek and Swartz, Reliable Computer Systems, Second edition, Digital Press 1992]

In this example there are many possible reliability block diagrams, depending on the hardware resource allocation and the software architecture (structure and behavior of the software components):

- An initial reliability block diagram could be deduced from the structure given that the reliability of each component (Rp1, Rp2, Rp3, R_V) has been specified.
- If components share resources, their reliabilities are not independent (they have common-mode failures) and the shared resources must be represented in the block diagram.

Finally, depending on the nature of the "voting," the system reliability can vary:

- a majority voter requires agreement between at least two components to determine the correct output
- an averaging voter computes the average of the three inputs (perhaps subject to some "reasonability" test)
- a priority voter might assign weights to different components (for example, the component executing the simpler or better known algorithm might have a higher weight)

Carnegie Mellon
Software Engineering Institute

Tradeoffs Between Dependability and Performance in TMR

If the components share a processor the latency depends on how many components are working:

- **performance calculations should be based on worst-case i.e., all components are working**
- **voter can decide when to send output to constrain latency variability**

Carnegie Mellon
Software Engineering Institute

RB Dependability Analysis

For a 3-component recovery block system :

$$R_{RB}(t) = e^{-\lambda t} \sum_{i=0}^{2} C^{i}(1 - e^{-\lambda t})^{i} \qquad MTTF_{RB} = \frac{1}{\lambda}\left(1 + \frac{c}{2} + \frac{c^2}{3}\right)$$

Where c is the acceptance test coverage.

- **If c=1 (test never fails):** $MTTF_{RB} = \frac{11}{6\lambda}$
- **If c=0.5 (test fails half the time):** $MTTF_{RB} = \frac{4}{3\lambda}$
- **If c=0 (test always fails):** $MTTF_{RB} = \frac{1}{\lambda}$

Recovery Blocks implements a different type of redundancy. Several components process information from the environment but only one at a time. In the case of a three-component Recovery Block system:

•the voter selects component P1 if P1 is working; or else it selects P2 if P2 is working; or else it selects P3 if P3 is working; or else it shuts down (the system fails).

•since the voter must make a decision based on just one component's results, the voting is more complicated and takes the form of an "acceptance test"

In the worst case, with an acceptance test coverage c=0, the MTTF of the recovery block system is $1/\lambda$. That is, if the primary module's acceptance test always fails (i.e., never detects an error) the MTTF is just that of the primary module.

In the best case, with perfect acceptance test coverage c=1, the MTTF of the recovery block system is $11/6\lambda$

•This is almost twice that of the TMR system without repairs.

•This is the case in which all modules have acceptance tests that never fail to detect their errors.

•This might not be a reasonable assumption in all cases.

Tradeoffs Between Dependability and Performance in RB

Latency variability is greater:

- components perform different algorithms (execution time varies)
- acceptance tests are component-dependent (execution time varies)
- when a component fails, there is a roll-back to a safe state before the next alternative is tried (previous execution time is wasted + time to restore state)

Additional Tradeoffs Between Dependability and Performance

TMR and RB repair operations also affect performance:

- running diagnostics
- restarting a process
- rebooting a processor

Dependability Sensitivity Points

If a component has a failure rate of one per 1000 hrs. and a repair rate of one per 10 hours (λ=0.001, μ=0.1):

The Mean Time To Failure for the alternatives are:

- Non-redundant component = $1/\lambda$ = 1,000 hours
- TMR without repair = $5/(6\lambda)$ = 833 hours
- RB with 50% coverage = $4/(3\lambda)$ = 1,333 hours
- RB with 100% coverage = $11/(6\lambda)$ = 1,833 hours
- TMR with repair = $5/(6\lambda) + \mu/6\lambda2$ = 17,500 hours

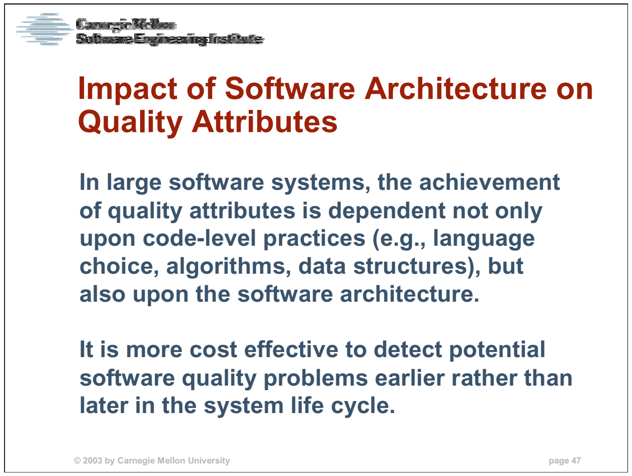
The choice of "voting" technique (i.e., TMR or RB) constitute a sensitivity point for dependability.

Risks in TMR and RB

Depending on the TMR approach to repairs, different risks emerge:

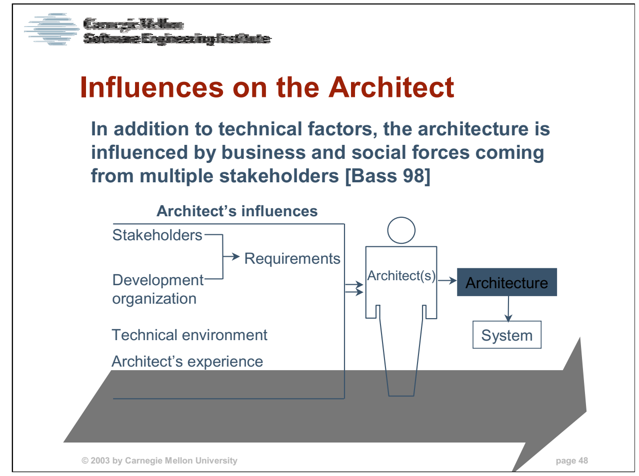
- a TMR system without repair is less dependable that just a single component!
- a TMR system with very lengthy repairs could be just as undependable

The RB time to execute components, tests, and recoveries varies and could present a performance risk if the deadlines are tight.



When the software architecture is specified, designers need to determine:

- the extent to which features of the software architecture influence quality attributes
- the extent to which techniques used for one attribute support or conflict with those of another attribute
- the extent to which multiple quality attribute requirements can be satisfied simultaneously



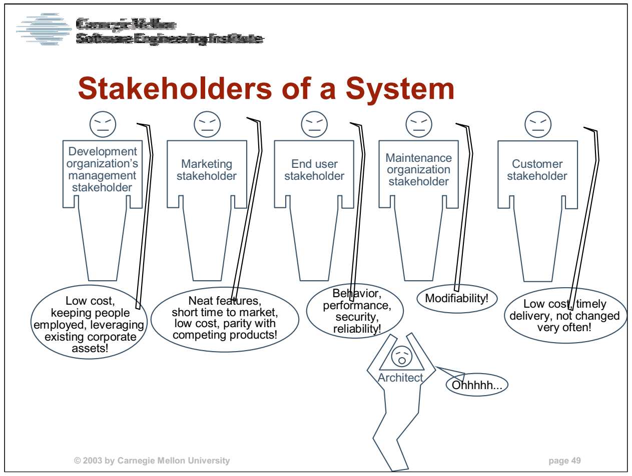
The ABC works like this:

- Stakeholders and organizational goals influence and/or determine the set of system requirements.
- The requirements, the current technical environment, and the architect's experience lead to an architecture.
- Architectures yield systems.
- Systems, and their successes or failures, suggest future new organizational capabilities and requirements. They also add to the architect's experience that will come into play for future system designs, and may influence or even change the technical environment.

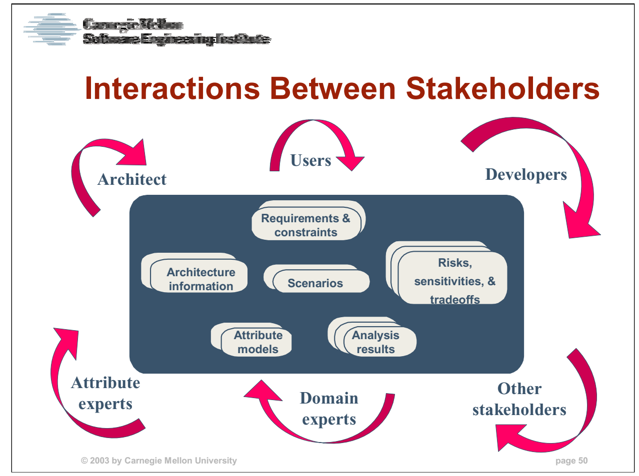
There are multiple activities in the architecture business cycle:

- creating the business case for the system
- understanding the requirements
- creating or selecting the architecture
- representing and communicating the architecture
- analyzing or evaluating the architecture
- implementing the system based on the architecture
- ensuring that the implementation conforms to the architecture

These activities do not take place in a strict sequence. There are many feedback loops as the multiple stakeholders negotiate among themselves, striving for agreement.



This slide illustrates some of the many stakeholders of a system. Each type of stakeholder has a different set of requirements.
In some cases, the requirements overlap (low cost is a recurring theme!), but in other cases, the requirements may be mutually exclusive. It is the architect's job to juggle and balance the conflicting requirements of the various stakeholders.
The result is that the architect may feel overwhelmed by the volume and conflicting nature of all the various requirements, which can affect the decisions made in choosing or developing an architecture.
As we will see later in this course, it is important for the architect to seek and encourage the active engagement of all stakeholders early in the project. This places the architect in a better position to make adjustments and tradeoffs when conflicting stakeholder requirements are identified.



Imagine the stakeholders sharing a blackboard:

- participants can provide or obtain information at any time
- participant can use information from any other participant

Stakeholders must identify the quality attribute requirements and constraints.

The architect provides architectural information including the components and connections between components, showing the flow of data, and the the behavior — underlying semantics of the system and the components, showing the flow of control.

Stakeholders propose scenarios describing an operational situation, a modification to the system, a change in the environment, etc.

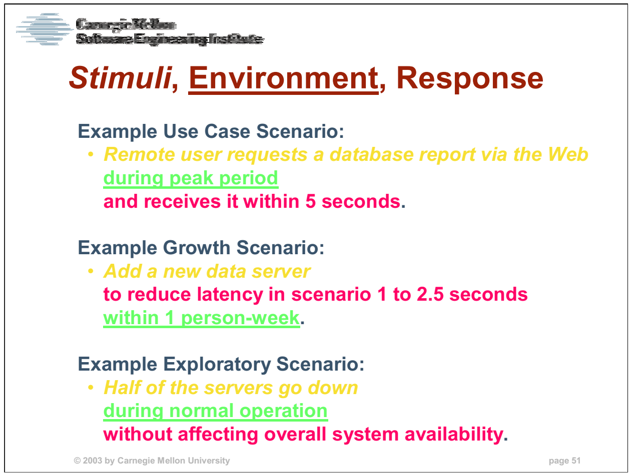
- Scenarios are used to explore the space defined by the requirements, constraints, and architectural decisions. Scenarios define tests to be conducted through architecture analysis

Some stakeholders (e.g., domain experts) identify models for evaluating quality attributes. Some models are specific to certain quality attributes, other models are applicable to multiple attributes.

Depending on the attributes of interest, there are different qualitative and quantitative techniques to conduct the analysis: focus on system activities (e.g., latency, availability), focus on user activities (e.g., time to complete a task), focus on the system (e.g., modifiability, interoperability).

Depending on the attribute models and the architectural approaches, various risks, sensitivities and tradeoffs can be discovered during the analysis:

- risks — alternatives that might create future problems in some quality attribute
- sensitivity points — alternatives for which a slight change makes a significant difference in some quality attribute
- tradeoffs — decisions affecting more than one quality attribute



Scenarios are used to exercise the architecture against current and future situations:

•Use case scenarios reflect the normal state or operation of the system.

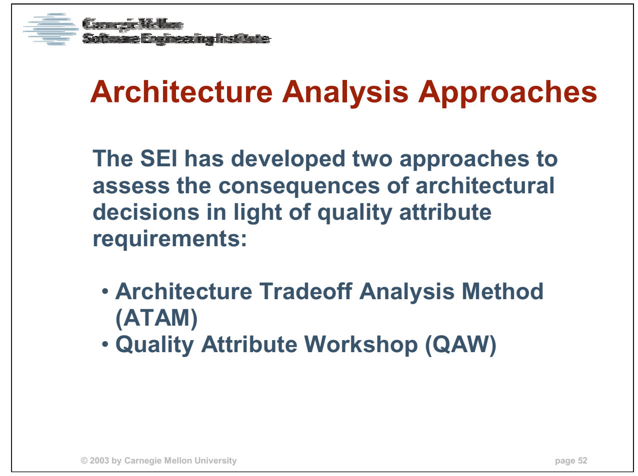
•Growth scenarios are anticipated changes to the system (e.g., double the message traffic, change message format shown on operator console).

•Exploratory scenarios are extreme changes to the system. These changes are not necessarily anticipated or even desirable situations (e.g., message traffic grows 100 times, replace the operating system).

The distinction between growth and exploratory scenarios is system or situation dependent.

•What might be anticipated growth in a business application might be a disaster in a deep space probe (e.g., 20% growth in message storage per year).

•There are no clear rules other than stakeholder consensus that some scenarios are likely (desirable or otherwise) and other scenarios are unlikely (but could happen and, if they do, it would be useful to understand the consequences).



Carnegie Mellon
Software Engineering Institute

Different in Application

ATAM
- requires a software architecture
- stakeholders propose scenarios
- scenario analysis during meetings

QAW
- before software architecture is drafted
- stakeholder scenarios refined into test cases
- scenario analysis completed off-line but results presented during meetings

 page 53

In an ATAM evaluation, an external team facilitates stakeholder meetings during which scenarios are developed representing the quality attributes of the system. These scenarios are then prioritized, and the highest priority scenarios are analyzed against the software architecture.

In a QAW, the highest priority stakeholder-generated scenarios are turned into “test cases” by adding additional details (e.g., context, assets involved, sequence of activities). The architecture team then independently analyzes the “test cases” against the system architecture and documents the results.

The test case creation and analysis phase often takes place over an extended period of time. After completing this phase, the architecture team presents the results to the sponsors and stakeholders.

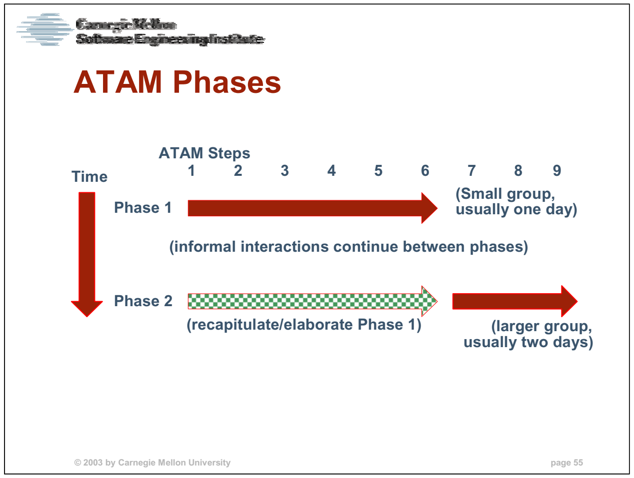
Carnegie Mellon
Software Engineering Institute

The ATAM Process

The ATAM process is a short, facilitated interaction between the stakeholders to conduct the activities outlined in the blackboard, leading to the identification of risks, sensitivities, and tradeoffs:

- risks can be the focus of mitigation activities, e.g. further design, further analysis, prototyping
- sensitivities and tradeoffs can be explicitly documented

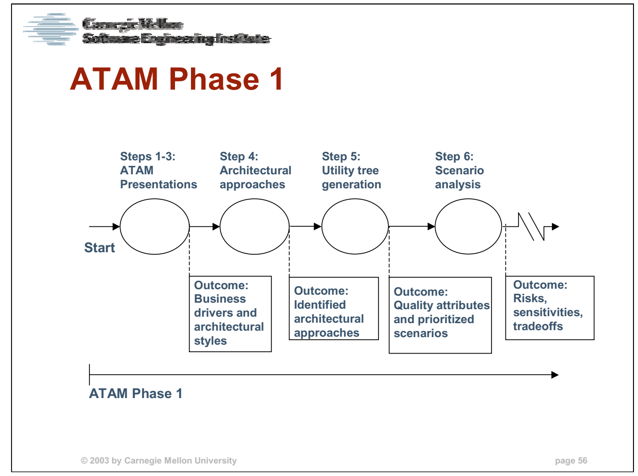
 page 54



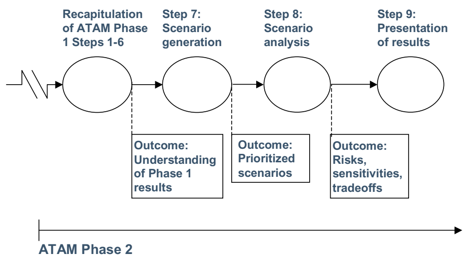
ATAM evaluations are often conducted in two stages or phases:

•during phase 1 the architect describes the quality attribute goals and how the architecture meets these goals

•during phase 2 evaluators determine if the larger group of stakeholders agrees with the goals and the results



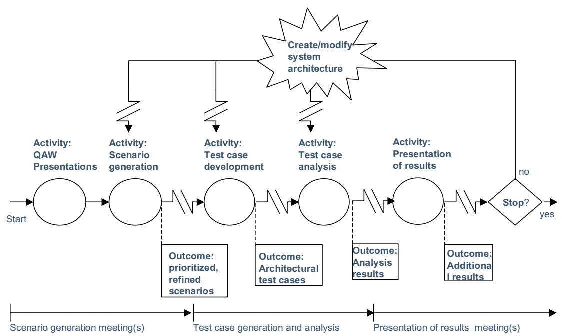
ATAM Phase 2



Quality Attribute Workshops

The Quality Attributes Workshops (QAW) are a variation of ATAM that is applicable earlier in the life-cycle, before a complete software architecture has been defined.

The QAW Process



M.R. Barbacci, et al., *Quality Attribute Workshops, 2nd Edition,* (CMU/SEI-2002-TR-019). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 2002.

The process can be organized into four distinct segments: (1) scenario generation, prioritization, and refinement; (2) test case development; (3) analysis of test cases against the architecture; and (4) presentation of the results. These are the four red ovals in the figure.

The first and last segments of the process occur in facilitated one-day meetings. The middle segments take place off-line and could continue over an extended period of time.

The process is iterative in that the test case analyses might lead to the development of additional test cases or to architectural modifications. Architectural modifications might prompt additional test case analyses, etc.

There is a further iteration, not shown in the figure, in which test cases are developed in batches, sequential analyses are performed, and each time, the architecture is modified accordingly.

Test Case Context

Describes the mission, the assets involved, the geographical region, the operational setting, and the players.

For example in a test case involving a failure, the test case may define:

- **the operation at the time of failure**
- **what happens immediately after, when the system is reacting to the failure,**
- **degraded operation during the interval when repair is underway**
- **restoring the system to normal operation.**

Example Test Case Context

"Humans and robotic missions are present in the Mars surface when one of three stationary-stationary satellites has a power amplifier failure.
The primary communications payload is disabled for long-haul functions but ….. Secondary Telemetry and Tele-Command (TTC) for spacecraft health is …..
The crew on the surface is concentrated in one area and the other missions ….
The event occurs late in the development of the communications network, so the system is well developed."

Test Case Issues and Questions

The test case includes a number of questions about the events, to be answered by the analysis:

- to help focus the analysis, questions are grouped according to a specific issue of concern
- issues are tagged by the quality attributes it addresses, e.g., performance, security

Example Issues and Questions

1. *Issue: Mission safety requires consistent and frequent communications between the crew and earth (P, A)*
   a) *Question: How long does it take to detect the failure?*
   b) *Question: How long does it take to reconfigure the system to minimize the time the crew is without communication?*

2. *Issue: System operation will be degraded (P, A)*
   a) *Question: Is there a way for the customer to simplify their procedures so they can handle a larger number of missions with less trouble than coordinating two as they do now?*
   b) *Question: What redundancy is required?*
   c) *Question: Is there a way to send information about the degraded satellite back to Earth for analysis?*

ATAM and QAW Status

We have experience in using the methods in a wide variety of application areas.

There is an ATAM handbook and a training course to make process repeatable and transitionable. Most of the material is relevant to the QAW process.

Additional information available:

http://www.sei.cmu.edu/activities/ata