Software Quality Assurance

Lecture 4

Created By: M.Imran Javed

Imranjaved49@gmail.com

Causes of Poor Software Quality

Poor quality is not an inevitable attribute of software. It results from known causes. It can be predicted and controlled, but only if its causes are understood and addressed.

With more critical business processes being implemented in software, quality problems are a primary business risk. I'll discuss five primary causes of poor software quality and how to mitigate their damaging effects using methods other than brute testing.

1. Lack of domain knowledge:

Perhaps the greatest contributor to poor software quality is the unfortunate fact that most developers are not experts in the business domain served by their applications, be it telecommunications, banking, energy, supply chain, retail, or others.

Over time they will learn more about the domain, but much of this learning will come through correcting defects caused by their mistaken understanding of the functional requirements. The best way to mitigate this cause is to provide access to domain experts from the business, proactively train developers in the business domain, and conduct peer reviews with those possessing more domain experience.

2. Lack of technology knowledge:

Most developers are proficient in several computer languages and technologies. However, modern multi-tier business applications are a complex tangle of many computer languages and different software platforms.

These tiers include user interface, business logic, and data management, and they may interact through middleware with enterprise resource systems and legacy applications written in archaic languages. Few developers know all of these languages and technologies, and their incorrect assumptions about how other technologies work is a prime source of the non-functional defects that cause damaging outages, data corruption, and security breaches during operation.

The best way to mitigate this cause is to cross-train developers in different application technologies, conduct peer reviews with developers working in other application tiers, and perform static and dynamic analyses of the code.

3. Unrealistic schedules:

When developers are forced to sacrifice sound software development practices to ridiculous schedules the results are rarely good.

The few successful outcomes are based on heroics that are rarely repeated on future death marches. When working at breakneck pace, stressed developers make more mistakes and have less time to find them. The only way to mitigate these death march travesties is through enforcing strong project management practices. Controlling commitments through planning, tracking progress to identify problems, and controlling endless requirements changes are critical practices for providing a professional environment for software development.

4. Badly engineered software:

Two-thirds or more of most software development activity involves changing or enhancing existing code. Studies have shown that half of the time spent modifying existing software is expended trying to figure out what is going on in the code.

Unnecessarily complex code is often impenetrable and modifying it leads to numerous mistakes and unanticipated negative side effects. These newly injected defects cause expensive rework and delayed releases. The best way to mitigate this cause is to re-factor critical portions of the code guided by information from architectural and static code analyses.

5. Poor acquisition practices:

Most large multi-tier applications are built and maintained by distributed teams, some or all of whom may be outsourced from other companies. Consequently, the acquiring organization often has little visibility into or control over the quality of the software they are receiving.

For various reasons, CMMI levels have not always guaranteed high quality software deliveries. To mitigate the risks of quality problems in externally supplied software, acquiring managers should implement quality targets in their contracts and a strong quality assurance gate for delivered software.

Thoughts on the List

The first two causes distinguish between functional and non-functional quality problems, a critical distinction since non-functional defects are not detected as readily during test and their effects are frequently more devastating during operations.

The third and fourth causes have been perennial, although the fourth problem is exacerbated by the increase in technologies integrated into modern applications.

The final problem is not entirely new, but has grown in effect with growth in outsourcing and packaged software. Just missing this list but deserving of attention are breakdowns in coordination among distributed software teams, a cause that would make the top five in some environments.

In well run software organizations testing is not a defect detection activity. Rather, testing should merely verify that the software performs correctly under a wide range of operational conditions. By understanding and addressing the top five causes of defects, quality can be designed in from the start, substantially reducing both the 40% of project effort typically spent on rework and the risks to which software exposes business.

Quality Assurance Organizations

- No quality assurance 60%
- Token quality assurance 20%
- Passive quality assurance 15%
- Active quality assurance 5%

The focus on software quality naturally is dependent on the industry, as well as the importance of the software application. More critical applications, naturally, need to have higher software quality than others.

Software Quality in Six Sub- Industries

- Systems software that controls physical devices
- Information systems that companies build for their own use
- Outsource or contract software built for clients
- Commercial software built by vendors for lease or sale
- Military software built following various military standards
- End-user software built for private use by computer literate workers or managers

Quality Laggards

- No software quality measurement program of any kind
- No usage of formal design and code inspections
- No knowledge of the concepts of defect potentials and defect removal efficiency
- Either no quality assurance group or a group that is severely understaffed
- No trained testing specialists available
- Few or no automated quality assurance tools
- No quality and reliability estimation capability
- Minimal or no software project management tools available
- No automated risk assessment or avoidance capability
- From a low of one to a high of perhaps four distinct testing stages
- No test library or test-case management tools available
- No complexity analysis tools utilized
- Defect potentials averaging more than 6 defects per function point
- Defect removal efficiency averaging less than 80%
- Executive and managerial indifference (and ignorance) of quality matters

Related Issues

- Staff morale and voluntary attrition
- Market shares and competitive positioning
- Litigation and product recalls

Quality laggards are the very companies with highest probability of cancelled projects, several schedule overruns, and severe overruns. It is no coincidence that software groups among the quality laggards also tend to be candidates for immediate replacement by outsource organizations.

Quality Attributes

Quality attributes are the overall factors that affect run-time behavior, system design, and user experience. They represent areas of concern that have the potential for application wide impact across layers and tiers. Some of these attributes are related to the overall system design, while others are specific to run time, design time, or user centric issues. The extent to which the application possesses a desired combination of quality attributes such as usability, performance, reliability, and security indicates the success of the design and the overall quality of the software application.

When designing applications to meet any of the quality attributes requirements, it is necessary to consider the potential impact on other requirements. You must analyze the tradeoffs between multiple quality attributes. The importance or priority of each quality

attribute differs from system to system; for example, interoperability will often be less important in a single use packaged retail application than in a line of business (LOB) system.

This chapter lists and describes the quality attributes that you should consider when designing your application. To get the most out of this chapter, use the table below to gain an understanding of how quality attributes map to system and application quality factors, and read the description of each of the quality attributes. Then use the sections containing key guidelines for each of the quality attributes to understand how that attribute has an impact on your design, and to determine the decisions you must make to addresses these issues. Keep in mind that the list of quality attributes in this chapter is not exhaustive, but provides a good starting point for asking appropriate questions about your architecture.

Common Quality Attributes

The following table describes the quality attributes covered in this chapter. It categorizes the attributes in four specific areas linked to design, runtime, system, and user qualities. Use this table to understand what each of the quality attributes means in terms of your application design.

Category	Quality attribute	Description
Design Qualities	*Conceptual Integrity*	Conceptual integrity defines the consistency and coherence of the overall design. This includes the way that components or modules are designed, as well as factors such as coding style and variable naming.
	Maintainability	Maintainability is the ability of the system to undergo changes with a degree of ease. These changes could impact components, services, features, and interfaces when adding or changing the functionality, fixing errors, and meeting new business requirements.
	Reusability	Reusability defines the capability for components and subsystems to be suitable for use in other applications and in other scenarios. Reusability minimizes the duplication of components and also the implementation time.
Run-time Qualities	*Availability*	Availability defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period. Availability will be affected by system errors, infrastructure problems, malicious attacks, and system load.
	Interoperability	Interoperability is the ability of a system or different systems to operate successfully by communicating and exchanging information with other external systems written and run by external parties. An interoperable system makes it easier to

		exchange and reuse information internally as well as externally.
	Manageability	Manageability defines how easy it is for system administrators to manage the application, usually through sufficient and useful instrumentation exposed for use in monitoring systems and for debugging and performance tuning.
	Performance	Performance is an indication of the responsiveness of a system to execute any action within a given time interval. It can be measured in terms of latency or throughput. Latency is the time taken to respond to any event. Throughput is the number of events that take place within a given amount of time.
	Reliability	Reliability is the ability of a system to remain operational over time. Reliability is measured as the probability that a system will not fail to perform its intended functions over a specified time interval.
	Scalability	Scalability is ability of a system to either handle increases in load without impact on the performance of the system, or the ability to be readily enlarged.
	Security	Security is the capability of a system to prevent malicious or accidental actions outside of the designed usage, and to prevent disclosure or loss of information. A secure system aims to protect assets and prevent unauthorized modification of information.
	Supportability	Supportability is the ability of the system to provide information helpful for identifying and resolving issues when it fails to work correctly.
System Qualities	*Testability*	Testability is a measure of how easy it is to create test criteria for the system and its components, and to execute these tests in order to determine if the criteria are met. Good testability makes it more likely that faults in a system can be isolated in a timely and effective manner.
User Qualities	*Usability*	Usability defines how well the application meets the requirements of the user and consumer by being intuitive, easy to localize and globalize, providing good access for disabled users, and resulting in a good overall user experience.

The following sections describe each of the quality attributes in more detail, and provide guidance on the key issues and the decisions you must make for each one:

- Availability
- Conceptual Integrity
- Interoperability
- Maintainability
- Manageability
- Performance
- Reliability
- Reusability
- Scalability
- Security
- Supportability
- Testability
- User Experience / Usability

Availability

Availability defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period. Availability will be affected by system errors, infrastructure problems, malicious attacks, and system load. The key issues for availability are:

- A physical tier such as the database server or application server can fail or become unresponsive, causing the entire system to fail. Consider how to design failover support for the tiers in the system. For example, use Network Load Balancing for Web servers to distribute the load and prevent requests being directed to a server that is down. Also, consider using a RAID mechanism to mitigate system failure in the event of a disk failure. Consider if there is a need for a geographically separate redundant site to failover to in case of natural disasters such as earthquakes or tornados.
- Denial of Service (DoS) attacks, which prevent authorized users from accessing the system, can interrupt operations if the system cannot handle massive loads in a timely manner, often due to the processing time required, or network configuration and congestion. To minimize interruption from DoS attacks, reduce the attack surface area, identify malicious behavior, use application instrumentation to expose unintended behavior, and implement comprehensive data validation. Consider using the Circuit Breaker or Bulkhead patterns to increase system resiliency.
- Inappropriate use of resources can reduce availability. For example, resources acquired too early and held for too long cause resource starvation and an inability to handle additional concurrent user requests.
- Bugs or faults in the application can cause a system wide failure. Design for proper exception handling in order to reduce application failures from which it is difficult to recover.
- Frequent updates, such as security patches and user application upgrades, can reduce the availability of the system. Identify how you will design for run-time upgrades.

- A network fault can cause the application to be unavailable. Consider how you will handle unreliable network connections; for example, by designing clients with occasionally-connected capabilities.
- Consider the trust boundaries within your application and ensure that subsystems employ some form of access control or firewall, as well as extensive data validation, to increase resiliency and availability.

Conceptual Integrity

Conceptual integrity defines the consistency and coherence of the overall design. This includes the way that components or modules are designed, as well as factors such as coding style and variable naming. A coherent system is easier to maintain because you will know what is consistent with the overall design. Conversely, a system without conceptual integrity will constantly be affected by changing interfaces, frequently deprecating modules, and lack of consistency in how tasks are performed. The key issues for conceptual integrity are:

- Mixing different areas of concern within your design. Consider identifying areas of concern and grouping them into logical presentation, business, data, and service layers as appropriate.
- Inconsistent or poorly managed development processes. Consider performing an Application Lifecycle Management (ALM) assessment, and make use of tried and tested development tools and methodologies.
- Lack of collaboration and communication between different groups involved in the application lifecycle. Consider establishing a development process integrated with tools to facilitate process workflow, communication, and collaboration.
- Lack of design and coding standards. Consider establishing published guidelines for design and coding standards, and incorporating code reviews into your development process to ensure guidelines are followed.
- Existing (legacy) system demands can prevent both refactoring and progression toward a new platform or paradigm. Consider how you can create a migration path away from legacy technologies, and how to isolate applications from external dependencies. For example, implement the Gateway design pattern for integration with legacy systems.

Interoperability

Interoperability is the ability of a system or different systems to operate successfully by communicating and exchanging information with other external systems written and run by external parties. An interoperable system makes it easier to exchange and reuse information internally as well as externally. Communication protocols, interfaces, and data formats are the key considerations for interoperability. Standardization is also an important aspect to be considered when designing an interoperable system. The key issues for interoperability are:

- Interaction with external or legacy systems that use different data formats. Consider how you can enable systems to interoperate, while evolving separately or even being replaced. For example, use orchestration with adaptors to connect with external or legacy systems and translate data between systems; or use a canonical data model to handle interaction with a large number of different data formats.
- Boundary blurring, which allows artifacts from one system to defuse into another. Consider how you can isolate systems by using service interfaces and/or mapping layers. For example, expose services using interfaces based on XML or standard types in order to support interoperability with other systems. Design components to be cohesive and have low coupling in order to maximize flexibility and facilitate replacement and reusability.
- Lack of adherence to standards. Be aware of the formal and de facto standards for the domain you are working within, and consider using one of them rather than creating something new and proprietary.

Maintainability

Maintainability is the ability of the system to undergo changes with a degree of ease. These changes could impact components, services, features, and interfaces when adding or changing the application's functionality in order to fix errors, or to meet new business requirements. Maintainability can also affect the time it takes to restore the system to its operational status following a failure or removal from operation for an upgrade. Improving system maintainability can increase availability and reduce the effects of run-time defects. An application's maintainability is often a function of its overall quality attributes but there a number of key issues that can directly affect maintainability:

- Excessive dependencies between components and layers, and inappropriate coupling to concrete classes, prevents easy replacement, updates, and changes; and can cause changes to concrete classes to ripple through the entire system. Consider designing systems as well-defined layers, or areas of concern, that clearly delineate the system's UI, business processes, and data access functionality. Consider implementing cross-layer dependencies by using abstractions (such as abstract classes or interfaces) rather than concrete classes, and minimize dependencies between components and layers.
- The use of direct communication prevents changes to the physical deployment of components and layers. Choose an appropriate communication model, format, and protocol. Consider designing a pluggable architecture that allows easy upgrades and maintenance, and improves testing opportunities, by designing interfaces that allow the use of plug-in modules or adapters to maximize flexibility and extensibility.
- Reliance on custom implementations of features such as authentication and authorization prevents reuse and hampers maintenance. To avoid this, use the built-in platform functions and features wherever possible.

- The logic code of components and segments is not cohesive, which makes them difficult to maintain and replace, and causes unnecessary dependencies on other components. Design components to be cohesive and have low coupling in order to maximize flexibility and facilitate replacement and reusability.
- The code base is large, unmanageable, fragile, or over complex; and refactoring is burdensome due to regression requirements. Consider designing systems as well defined layers, or areas of concern, that clearly delineate the system's UI, business processes, and data access functionality. Consider how you will manage changes to business processes and dynamic business rules, perhaps by using a business workflow engine if the business process tends to change. Consider using business components to implement the rules if only the business rule values tend to change; or an external source such as a business rules engine if the business decision rules do tend to change.
- The existing code does not have an automated regression test suite. Invest in test automation as you build the system. This will pay off as a validation of the system's functionality, and as documentation on what the various parts of the system do and how they work together.
- Lack of documentation may hinder usage, management, and future upgrades. Ensure that you provide documentation that, at minimum, explains the overall structure of the application.

Manageability

Manageability defines how easy it is for system administrators to manage the application, usually through sufficient and useful instrumentation exposed for use in monitoring systems and for debugging and performance tuning. Design your application to be easy to manage, by exposing sufficient and useful instrumentation for use in monitoring systems and for debugging and performance tuning. The key issues for manageability are:

- Lack of health monitoring, tracing, and diagnostic information. Consider creating a health model that defines the significant state changes that can affect application performance, and use this model to specify management instrumentation requirements. Implement instrumentation, such as events and performance counters, that detects state changes, and expose these changes through standard systems such as Event Logs, Trace files, or Windows Management Instrumentation (WMI). Capture and report sufficient information about errors and state changes in order to enable accurate monitoring, debugging, and management. Also, consider creating management packs that administrators can use in their monitoring environments to manage the application.
- Lack of runtime configurability. Consider how you can enable the system behavior to change based on operational environment requirements, such as infrastructure or deployment changes.
- Lack of troubleshooting tools. Consider including code to create a snapshot of the system's state to use for troubleshooting, and including custom

instrumentation that can be enabled to provide detailed operational and functional reports. Consider logging and auditing information that may be useful for maintenance and debugging, such as request details or module outputs and calls to other systems and services.

Performance

Performance is an indication of the responsiveness of a system to execute specific actions in a given time interval. It can be measured in terms of latency or throughput. Latency is the time taken to respond to any event. Throughput is the number of events that take place in a given amount of time. An application's performance can directly affect its scalability, and lack of scalability can affect performance. Improving an application's performance often improves its scalability by reducing the likelihood of contention for shared resources. Factors affecting system performance include the demand for a specific action and the system's response to the demand. The key issues for performance are:

- Increased client response time, reduced throughput, and server resource over utilization. Ensure that you structure the application in an appropriate way and deploy it onto a system or systems that provide sufficient resources. When communication must cross process or tier boundaries, consider using coarse-grained interfaces that require the minimum number of calls (preferably just one) to execute a specific task, and consider using asynchronous communication.
- Increased memory consumption, resulting in reduced performance, excessive cache misses (the inability to find the required data in the cache), and increased data store access. Ensure that you design an efficient and appropriate caching strategy.
- Increased database server processing, resulting in reduced throughput. Ensure that you choose effective types of transactions, locks, threading, and queuing approaches. Use efficient queries to minimize performance impact, and avoid fetching all of the data when only a portion is displayed. Failure to design for efficient database processing may incur unnecessary load on the database server, failure to meet performance objectives, and costs in excess of budget allocations.
- Increased network bandwidth consumption, resulting in delayed response times and increased load for client and server systems. Design high performance communication between tiers using the appropriate remote communication mechanism. Try to reduce the number of transitions across boundaries, and minimize the amount of data sent over the network. Batch work to reduce calls over the network.

Reliability

Reliability is the ability of a system to continue operating in the expected way over time. Reliability is measured as the probability that a system will not fail and that it will

perform its intended function for a specified time interval. The key issues for reliability are:

- The system crashes or becomes unresponsive. Identify ways to detect failures and automatically initiate a failover, or redirect load to a spare or backup system. Also, consider implementing code that uses alternative systems when it detects a specific number of failed requests to an existing system.
- Output is inconsistent. Implement instrumentation, such as events and performance counters, that detects poor performance or failures of requests sent to external systems, and expose information through standard systems such as Event Logs, Trace files, or WMI. Log performance and auditing information about calls made to other systems and services.
- The system fails due to unavailability of other externalities such as systems, networks, and databases. Identify ways to handle unreliable external systems, failed communications, and failed transactions. Consider how you can take the system offline but still queue pending requests. Implement store and forward or cached message-based communication systems that allow requests to be stored when the target system is unavailable, and replayed when it is online. Consider using Windows Message Queuing or BizTalk Server to provide a reliable once-only delivery mechanism for asynchronous requests.

Reusability

Reusability is the probability that a component will be used in other components or scenarios to add new functionality with little or no change. Reusability minimizes the duplication of components and the implementation time. Identifying the common attributes between various components is the first step in building small reusable components for use in a larger system. The key issues for reusability are:

- The use of different code or components to achieve the same result in different places; for example, duplication of similar logic in multiple components, and duplication of similar logic in multiple layers or subsystems. Examine the application design to identify common functionality, and implement this functionality in separate components that you can reuse. Examine the application design to identify crosscutting concerns such as validation, logging, and authentication, and implement these functions as separate components.
- The use of multiple similar methods to implement tasks that have only slight variation. Instead, use parameters to vary the behavior of a single method.
- Using several systems to implement the same feature or function instead of sharing or reusing functionality in another system, across multiple systems, or across different subsystems within an application. Consider exposing functionality from components, layers, and subsystems through service interfaces that other layers and systems can use. Use platform agnostic data types and structures that can be accessed and understood on different platforms.

Scalability

Scalability is ability of a system to either handle increases in load without impact on the performance of the system, or the ability to be readily enlarged. There are two methods for improving scalability: scaling vertically (scale up), and scaling horizontally (scale out). To scale vertically, you add more resources such as CPU, memory, and disk to a single system. To scale horizontally, you add more machines to a farm that runs the application and shares the load. The key issues for scalability are:

- Applications cannot handle increasing load. Consider how you can design layers and tiers for scalability, and how this affects the capability to scale up or scale out the application and the database when required. You may decide to locate logical layers on the same physical tier to reduce the number of servers required while maximizing load sharing and failover capabilities. Consider partitioning data across more than one database server to maximize scale-up opportunities and allow flexible location of data subsets. Avoid stateful components and subsystems where possible to reduce server affinity.
- Users incur delays in response and longer completion times. Consider how you will handle spikes in traffic and load. Consider implementing code that uses additional or alternative systems when it detects a predefined service load or a number of pending requests to an existing system.
- The system cannot queue excess work and process it during periods of reduced load. Implement store-and-forward or cached message-based communication systems that allow requests to be stored when the target system is unavailable, and replayed when it is online.

Security

Security is the capability of a system to reduce the chance of malicious or accidental actions outside of the designed usage affecting the system, and prevent disclosure or loss of information. Improving security can also increase the reliability of the system by reducing the chances of an attack succeeding and impairing system operation. Securing a system should protect assets and prevent unauthorized access to or modification of information. The factors affecting system security are confidentiality, integrity, and availability. The features used to secure systems are authentication, encryption, auditing, and logging. The key issues for security are:

- Spoofing of user identity. Use authentication and authorization to prevent spoofing of user identity. Identify trust boundaries, and authenticate and authorize users crossing a trust boundary.
- Damage caused by malicious input such as SQL injection and cross-site scripting. Protect against such damage by ensuring that you validate all input for length, range, format, and type using the constrain, reject, and sanitize principles. Encode all output you display to users.
- Data tampering. Partition the site into anonymous, identified, and authenticated users and use application instrumentation to log and expose behavior that can be

monitored. Also use secured transport channels, and encrypt and sign sensitive data sent across the network

- Repudiation of user actions. Use instrumentation to audit and log all user interaction for application critical operations.
- Information disclosure and loss of sensitive data. Design all aspects of the application to prevent access to or exposure of sensitive system and application information.
- Interruption of service due to Denial of service (DoS) attacks. Consider reducing session timeouts and implementing code or hardware to detect and mitigate such attacks.

Supportability

Supportability is the ability of the system to provide information helpful for identifying and resolving issues when it fails to work correctly. The key issues for supportability are:

- Lack of diagnostic information. Identify how you will monitor system activity and performance. Consider a system monitoring application, such as Microsoft System Center.
- Lack of troubleshooting tools. Consider including code to create a snapshot of the system's state to use for troubleshooting, and including custom instrumentation that can be enabled to provide detailed operational and functional reports. Consider logging and auditing information that may be useful for maintenance and debugging, such as request details or module outputs and calls to other systems and services.
- Lack of tracing ability. Use common components to provide tracing support in code, perhaps though Aspect Oriented Programming (AOP) techniques or dependency injection. Enable tracing in Web applications in order to troubleshoot errors.
- Lack of health monitoring. Consider creating a health model that defines the significant state changes that can affect application performance, and use this model to specify management instrumentation requirements. Implement instrumentation, such as events and performance counters, that detects state changes, and expose these changes through standard systems such as Event Logs, Trace files, or Windows Management Instrumentation (WMI). Capture and report sufficient information about errors and state changes in order to enable accurate monitoring, debugging, and management. Also, consider creating management packs that administrators can use in their monitoring environments to manage the application.

Testability

Testability is a measure of how well system or components allow you to create test criteria and execute tests to determine if the criteria are met. Testability allows faults in a system to be isolated in a timely and effective manner. The key issues for testability are:

- Complex applications with many processing permutations are not tested consistently, perhaps because automated or granular testing cannot be performed if the application has a monolithic design. Design systems to be modular to support testing. Provide instrumentation or implement probes for testing, mechanisms to debug output, and ways to specify inputs easily. Design components that have high cohesion and low coupling to allow testability of components in isolation from the rest of the system.
- Lack of test planning. Start testing early during the development life cycle. Use mock objects during testing, and construct simple, structured test solutions.
- Poor test coverage, for both manual and automated tests. Consider how you can automate user interaction tests, and how you can maximize test and code coverage.
- Input and output inconsistencies; for the same input, the output is not the same and the output does not fully cover the output domain even when all known variations of input are provided. Consider how to make it easy to specify and understand system inputs and outputs to facilitate the construction of test cases.

User Experience / Usability

The application interfaces must be designed with the user and consumer in mind so that they are intuitive to use, can be localized and globalized, provide access for disabled users, and provide a good overall user experience. The key issues for user experience and usability are:

- Too much interaction (an excessive number of clicks) required for a task. Ensure you design the screen and input flows and user interaction patterns to maximize ease of use.
- Incorrect flow of steps in multistep interfaces. Consider incorporating workflows where appropriate to simplify multistep operations.
- Data elements and controls are poorly grouped. Choose appropriate control types (such as option groups and check boxes) and lay out controls and content using the accepted UI design patterns.
- Feedback to the user is poor, especially for errors and exceptions, and the application is unresponsive. Consider implementing technologies and techniques that provide maximum user interactivity, such as Asynchronous JavaScript and XML (AJAX) in Web pages and client-side input validation. Use asynchronous techniques for background tasks, and tasks such as populating controls or performing long-running tasks.

Categories of Quality Attributes

- Product-specific quality attributes
- Organization-specific quality attributes

Product-Specific Attributes

- Ease of use
- Documentation
- Defect tolerance
- Defect frequency
- Defect impact
- Packaging
- Price versus reliability
- Performance

Organization-Specific Attributes

- Service and support
- Internal processes

Achieving High Levels of Software Quality

The purpose of Software Quality Management is to develop a quantitative understanding of the quality of the project's software products and achieve specific quality goals.

Software Quality Management involves defining quality goals for the software products, establishing plans to achieve these goals, and monitoring and adjusting the software plans, software work products, activities, and quality goals to satisfy the needs and desires of the customer and end user for high quality products.

The practices of Software Quality Management build on the practices of the Integrated Software Management and Software Product Engineering key process areas, which establish and implement the project's defined software process, and the Quantitative Process Management key process area, which establishes a quantitative understanding of the ability of the project's defined software process to achieve the desired results.

Quantitative goals are established for the software products based on the needs of the organization, the customer, and the end users. So that these goals may be achieved, the organization establishes strategies and plans, and the project specifically adjusts its defined software process, to accomplish the quality goals.

- Enterprise-wide quality programs
- Quality awareness and training methods
- Quality standards and guidelines
- Quality analysis methods
- Quality measurement methods
- Defect prevention methods
- Non-test defect removal methods
- Testing methods
- User-satisfaction methods
- Post-release quality control

Best in Class Quality Results

- Quality measurements
- Defect prevention
- Defect and quality estimation automation
- Defect tracking automation
- Complexity analysis tools
- Test coverage analysis tools
- Formal inspections
- Formal testing by test specialists
- Formal quality assurance group
- Executive and managerial understanding of quality

Two Components of Software Quality Improvement

- Reductions in total defect potentials using methods of defect prevention
- Improvements in cumulative removal efficiency levels

For Help Contact:

https://www.facebook.com/pages/Creatics/490207631116955?ref=aymt_homepage_panel

Skype: creatics1