What is 'Software Quality Assurance'?
Software QA involves the entire software development PROCESS - monitoring and improving the process, making sure that any agreed-upon standards and procedures are followed, and ensuring that problems are found and dealt with.

What is 'Software Testing'?
Testing involves operation of a system or application under controlled conditions and evaluating the results. Testing should intentionally attempt to make things go wrong to determine if things happen when they shouldn't or things don't happen when they should.

Does every software project need testers?
It depends on the size and context of the project, the risks, the development methodology, the skill and experience of the developers. If the project is a short-term, small, low risk project, with highly experienced programmers utilizing thorough unit testing or test-first development, then test engineers may not be required for the project to succeed. For non-trivial-size projects or projects with non-trivial risks, a testing staff is usually necessary. The use of personnel with specialized skills enhances an organization's ability to be successful in large, complex, or difficult tasks. It allows for both a) deeper and stronger skills and b) the contribution of differing perspectives.

What is Regression testing?
Retesting of a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made.

Why does software have bugs?
- Some of the reasons are:
- Miscommunication or no communication.
- Programming errors
- Changing requirements
- Time pressures

How can new Software QA processes be introduced in an existing Organization?
- It depends on the size of the organization and the risks involved.
- For small groups or projects, a more ad-hoc process may be appropriate, depending on the type of customers and projects.
- By incremental self managed team approaches.

What is verification and Validation?
Verification typically involves reviews and meetings to evaluate documents, plans, code, requirements, and specifications. This can be done with checklists, issues lists, walkthroughs, and inspection meetings. Validation typically involves actual testing and takes place after verifications are completed.

What is a 'walkthrough'? What's an 'inspection'?
A 'walkthrough' is an informal meeting for evaluation or informational purposes. Little or no preparation is usually required. An inspection is more formalized than a 'walkthrough', typically with 3-8 people including a moderator, reader, and a recorder to take notes. The subject of the inspection is typically a document such as a requirements spec or a test plan, and the purpose is to find problems and see what's missing, not to fix anything.

What kinds of testing should be considered?
Some of the basic kinds of testing involve:
Blackbox testing, Whitebox testing, Integration testing, Functional testing, smoke testing, Acceptance testing, Load testing, Performance testing, User acceptance testing.

What are 5 common problems in the software development process?
- Poor requirements
- Unrealistic Schedule
- Inadequate testing
- Changing requirements
- Miscommunication

What are 5 common solutions to software development problems?
- Solid requirements
- Realistic Schedule
- Adequate testing
- Clarity of requirements
- Good communication among the Project team

What is software 'quality'?
Quality software is reasonably bug-free, delivered on time and within budget, meets requirements and/or expectations, and is maintainable

What are some recent major computer system failures caused by software bugs?
Trading on a major Asian stock exchange was brought to a halt in November of 2005, reportedly due to an error in a system software upgrade. A May 2005 newspaper article reported that a major hybrid car manufacturer had to install a software fix on 20,000 vehicles due to problems with invalid engine warning lights and occasional stalling. Media reports in January of 2005 detailed severe problems with a $170 million high-profile U.S. government IT systems project. Software testing was one of the five major problem areas according to a report of the commission reviewing the project.

What is 'good code'? What is 'good design'?
'Good code' is code that works, is bug free, and is readable and maintainable. Good internal design is indicated by software code whose overall structure is clear, understandable, easily modifiable, and maintainable; is robust with sufficient error-handling and status logging capability; and works correctly when implemented. Good functional design is indicated by an application whose functionality can be traced back to customer and end-user requirements.

What is SEI? CMM? CMMI? ISO? Will it help?
These are all standards that determine effectiveness in delivering quality software. It helps organizations to identify best practices useful in helping them increase the maturity of their processes.

What steps are needed to develop and run software tests?
- Obtain requirements, functional design, and internal design specifications and other necessary documents
- Obtain budget and schedule requirements.
- Determine Project context.

- Identify risks.
- Determine testing approaches, methods, test environment, test data.
- Set Schedules, testing documents.
- Perform tests.
- Perform reviews and evaluations
- Maintain and update documents

What's a 'test plan'? What's a 'test case'?
A software project test plan is a document that describes the objectives, scope, approach, and focus of a software testing effort. A test case is a document that describes an input, action, or event and an expected response, to determine if a feature of an application is working correctly.

What should be done after a bug is found?
The bug needs to be communicated and assigned to developers that can fix it. After the problem is resolved, fixes should be re-tested, and determinations made regarding requirements for regression testing to check that fixes didn't create problems elsewhere

Will automated testing tools make testing easier?
It depends on the Project size. For small projects, the time needed to learn and implement them may not be worth it unless personnel are already familiar with the tools. For larger projects, or on-going long-term projects they can be valuable.

What's the best way to choose a test automation tool? Some of the points that can be noted before choosing a tool would be:

- Analyze the non-automated testing situation to determine the testing activity that is being performed.
- Testing procedures that are time consuming and repetition.
- Cost/Budget of tool, Training and implementation factors.
- Evaluation of the chosen tool to explore the benefits.

How can it be determined if a test environment is appropriate?
Test environment should match exactly all possible hardware, software, network, data, and usage characteristics of the expected live environments in which the software will be used.

What's the best approach to software test estimation?

- The 'best approach' is highly dependent on the particular organization and project and the experience of the personnel involved. Some of the following approaches to be considered are:
- Implicit Risk Context Approach
- Metrics-Based Approach
- Test Work Breakdown Approach
- Iterative Approach
- Percentage-of-Development Approach

What if the software is so buggy it can't really be tested at all?
The best bet in this situation is for the testers to go through the process of reporting whatever bugs or blocking-type problems initially show up, with the focus being on critical bugs.

How can it be known when to stop testing?
Common factors in deciding when to stop are:

- Deadlines (release deadlines, testing deadlines, etc.)
- Test cases completed with certain percentage passed
- Test budget depleted
- Coverage of code/functionality/requirements reaches a specified point
- Bug rate falls below a certain level
- Beta or alpha testing period ends

What if there isn't enough time for thorough testing?

- Use risk analysis to determine where testing should be focused.
- Determine the important functionalities to be tested.
- Determine the high risk aspects of the project.
- Prioritize the kinds of testing that need to be performed.
- Determine the tests that will have the best high-risk-coverage to time-required ratio.

What if the project isn't big enough to justify extensive testing?
Consider the impact of project errors, not the size of the project. The tester might then do ad hoc testing, or write up a limited test plan based on the risk analysis.

How does a client/server environment affect testing?
Client/server applications can be quite complex due to the multiple dependencies among clients, data communications, hardware, and servers, especially in multi-tier systems. Load/stress/performance testing may be useful in determining client/server application limitations and capabilities.

How can World Wide Web sites be tested?
Some of the considerations might include:

- Testing the expected loads on the server
- Performance expected on the client side
- Testing the required securities to be implemented and verified.
- Testing the HTML specification, external and internal links
- cgi programs, applets, javascripts, ActiveX components, etc. to be maintained, tracked, controlled

How is testing affected by object-oriented designs?
Well-engineered object-oriented design can make it easier to trace from code to internal design to functional design to requirements. If the application was well-designed this can simplify test design.

What is Extreme Programming and what's it got to do with testing?
Extreme Programming (XP) is a software development approach for small teams on risk-prone projects with unstable requirements. For testing ('extreme testing', programmers are expected to write unit and functional test code first - before writing the application code. Customers are expected to be an integral part of the project team and to help develop scenarios for acceptance/black box testing.

What makes a good Software Test engineer?
A good test engineer has a 'test to break' attitude, an ability to take the point of view of the customer, a strong desire for quality, and an attention to detail. Tact and diplomacy are useful in maintaining a cooperative relationship with developers, and an ability to communicate with both technical (developers) and non-technical (customers, management) people is useful.

What makes a good Software QA engineer?
They must be able to understand the entire software development process and how it can fit into the business approach and goals of the organization. Communication skills and the ability to understand various sides of issues are important. In organizations in the early stages of implementing QA processes, patience and diplomacy are especially needed. An ability to find problems as well as to see 'what's missing' is important for inspections and reviews

What's the role of documentation in QA?
QA practices should be documented such that they are repeatable. Specifications, designs, business rules, inspection reports, configurations, code changes, test plans, test cases, bug reports, user manuals, etc. should all be documented. Change management for documentation should be used.

What is a test strategy? What is the purpose of a test strategy?
It is a plan for conducting the test effort against one or more aspects of the target system. A test strategy needs to be able to convince management and other stakeholders that the approach is sound and achievable, and it also needs to be appropriate both in terms of the software product to be tested and the skills of the test team.

What information does a test strategy captures?
It captures an explanation of the general approach that will be used and the specific types, techniques, styles of testing

What is test data?
It is a collection of test input values that are consumed during the execution of a test, and expected results referenced for comparative purposes during the execution of a test

What is Unit testing?

It is implemented against the smallest testable element (units) of the software, and involves testing the internal structure such as logic and dataflow, and the unit's function and observable behaviors

How can the test results be used in testing?
Test Results are used to record the detailed findings of the test effort and to subsequently calculate the different key measures of testing

What is Developer testing?
Developer testing denotes the aspects of test design and implementation most appropriate for the team of developers to undertake.

What is independent testing?
Independent testing denotes the test design and implementation most appropriately performed by someone who is independent from the team of developers.

What is Integration testing?
Integration testing is performed to ensure that the components in the implementation model operate properly when combined to execute a use case

What is System testing?
A series of tests designed to ensure that the modified program interacts correctly with other system components. These test procedures typically are performed by the system maintenance staff in their development library.

What is Acceptance testing?
User acceptance testing is the final test action taken before deploying the software. The goal of acceptance testing is to verify that the software is ready, and that it can be used by end users to perform those functions and tasks for which the software was built

What is the role of a Test Manager?
The Test Manager role is tasked with the overall responsibility for the test effort's success. The role involves quality and test advocacy, resource planning and management, and resolution of issues that impede the test effort

What is the role of a Test Analyst?
The Test Analyst role is responsible for identifying and defining the required tests, monitoring detailed testing progress and results in each test cycle and evaluating the overall quality experienced as a result of testing activities. The role typically carries the responsibility for appropriately representing the needs of stakeholders that do not have direct or regular representation on the project.

What is the role of a Test Designer?
The Test Designer role is responsible for defining the test approach and ensuring its successful implementation. The role involves identifying the appropriate techniques, tools and guidelines to implement the required tests, and to give guidance on the corresponding resources requirements for the test effort

What are the roles and responsibilities of a Tester?
The Tester role is responsible for the core activities of the test effort, which involves conducting the necessary tests and logging the outcomes of that testing. The tester is responsible for identifying the most appropriate implementation approach for a given test, implementing individual tests, setting up and executing the tests, logging outcomes and verifying test execution, analyzing and recovering from execution errors.

What are the skills required to be a good tester?
A tester should have knowledge of testing approaches and techniques, diagnostic and problem-solving skills, knowledge of the system or application being tested, and knowledge of networking and system architecture

What is test coverage?
Test coverage is the measurement of testing completeness, and it's based on the coverage of testing expressed by the coverage of test requirements and test cases or by the coverage of executed code.

What is a test script?
The step-by-step instructions that realize a test, enabling its execution. Test Scripts may take the form of either documented textual instructions that are executed manually or computer readable instructions that enable automated test execution.

What is 'Software Quality Assurance'?
Software QA involves the entire software development PROCESS - monitoring and improving the process, making sure that any agreed-upon processes, standards and procedures are followed, and ensuring that problems are found and dealt with. It is oriented to 'prevention'.

What is 'Software Testing'?
Testing involves operation of a system or application under controlled conditions and evaluating the results (e.g., 'if the user is in interface A of the application while using hardware B, and does C, then D should happen'). The controlled conditions should include both normal and abnormal conditions. Testing should intentionally attempt to make things go wrong to determine if things happen when they shouldn't or things don't happen when they should. It is oriented to 'detection'.

- Organizations vary considerably in how they assign responsibility for QA and testing. Sometimes they're the combined responsibility of one group or individual. Also common are project teams that include a mix of testers and developers who work closely together, with overall testing and QA processes monitored by project managers. It will depend on what best fits an organization's size and business structure.
- Note that testing can be done by machines or people. When done by machines (computers usually) it's often called 'automated testing

Does every software project need testers?
While all projects will benefit from testing, some projects may not require independent test staff to succeed.

Which projects may not need independent test staff? The answer depends on the size and context of the project, the risks, the development methodology, the skill and experience of the developers, and other factors. For instance, if the project is a short-term, small, low risk project, with highly experienced programmers utilizing thorough unit testing or test-first development, then test engineers may not be required for the project to succeed.

In some cases an IT organization may be too small or new to have a testing staff even if the situation calls for it. In these circumstances it may be appropriate to instead use contractors or outsourcing, or adjust the project management and development approach (by switching to more senior developers and agile test-first development, for example). Inexperienced managers sometimes gamble on the success of a project by skipping thorough testing or having programmers do post-development functional testing of their own work, a decidedly high risk gamble.

For non-trivial-size projects or projects with non-trivial risks, a testing staff is usually necessary. As in any business, the use of personnel with specialized skills enhances an organization's ability to be successful in large, complex, or difficult tasks. It allows for both a) deeper and stronger skills and b) the contribution of differing perspectives. For example, programmers typically have the perspective of 'what are the technical issues in making this functionality work?'. A test engineer typically has the perspective of 'what might go wrong with this functionality, and how can we ensure it meets expectations?'. A technical person who can be highly effective in approaching tasks from both of those perspectives is rare, which is why, sooner or later, organizations bring in test specialists.

Why does software have bugs?
Miscommunication or no communication - as to specifics of what an application should or shouldn't do (the application's requirements).

Software complexity - the complexity of current software applications can be difficult to comprehend for anyone without experience in modern-day software development. Multi-tier distributed systems, applications utilizing multiple local and remote web services applications, use of cloud infrastructure, data communications, enormous databases, security complexities, and sheer size of applications have all contributed to the exponential growth in software/system complexity.

Programming errors - programmers, like anyone else, can make mistakes.

Changing requirements (whether documented or undocumented) - the end-user may not understand the effects of changes, or may understand and request them anyway - redesign, rescheduling of engineers, effects on other projects, work already completed that may have to be redone or thrown out, hardware requirements that may be affected, etc. If there are many minor changes or any major changes, known and unknown dependencies among parts of the project are likely to interact and cause problems, and the complexity of coordinating changes may result in errors. Enthusiasm of engineering staff may be affected. In some fast-changing business environments, continuously modified requirements may be a fact of life. In this case, management must understand the resulting risks, and QA and test engineers must adapt and plan for continuous extensive testing to keep the inevitable bugs from running out of control

Time pressures - scheduling of software projects is difficult at best, often requiring a lot of guesswork. When deadlines loom and the crunch comes, mistakes will be made

Poorly designed/documented code - it's tough to maintain and modify code that is badly written or poorly documented; the result is bugs. In many organizations management provides no incentive for programmers to write clear, understandable, maintainable code. In fact, it's usually the opposite: they get points mostly for quickly turning out code, and there's job security if nobody else can understand it ('if it was hard to write, it should be hard to read').

Software development tools - visual tools, class libraries, compilers, scripting tools, etc. often introduce their own bugs or are poorly documented, resulting in added bugs.

How can new Software QA processes be introduced in an existing organization?
A lot depends on the size of the organization and the risks involved. For large organizations with high-risk (in terms of lives or property) projects, serious management buy-in is required and a more formalized QA process is necessary.

Where the risk is lower, management and organizational buy-in and QA implementation may be a slower, step-at-a-time process. QA processes should be balanced with productivity so as to keep bureaucracy from getting out of hand.

For small groups or projects, a more ad-hoc process may be appropriate, depending on the type of customers and projects. A lot will depend on team leads or managers, feedback to developers, and ensuring adequate communications among customers, managers, developers, and testers.

The most value for effort will often be in (a) requirements management processes, with a goal of clear, complete, testable requirement specifications embodied in requirements or design documentation, or in agile-type environments extensive continuous coordination with end-users, (b) design reviews and code reviews, and (c) post-mortems/retrospectives. Agile approaches utilizing extensive regular communication can coordinate well with improved QA processes.

Other possibilities include incremental self-managed team approaches such as 'Kaizen' methods of continuous process improvement, the Deming-Shewhart Plan-Do-Check-Act cycle, and others.

What is verification & validation?
Verification typically involves reviews and meetings to evaluate documents, plans, code, requirements, and specifications. This can be done with checklists, issues lists, walkthroughs, and inspection meetings. Validation typically involves actual testing and takes place after verifications are completed. The term 'IV & V' refers to Independent Verification and Validation.

What is a 'walkthrough'?
A 'walkthrough' is an informal meeting for evaluation or informational purposes. Little or no preparation is usually required.

What's an 'inspection'?
An inspection is more formalized than a 'walkthrough', typically with 3-8 people including a moderator, reader, and a recorder to take notes. The subject of the inspection is typically a document such as a requirements spec or a test plan, and the purpose is to find problems and see what's missing, not to fix anything. Attendees should prepare for this type of meeting by reading thru the document; most problems will be found during this preparation. The result of the inspection meeting should be a written report. Thorough preparation for inspections is difficult, painstaking work, but is one of the most cost effective methods of ensuring quality. Employees who are most skilled at inspections are like the 'eldest brother' in the parable in 'Why is it often hard for organizations to get serious about quality assurance?'. Their skill may have low visibility but they are extremely valuable to any software development organization, since bug prevention is far more cost-effective than bug detection.

What kinds of testing should be considered?

- black box testing - not based on any knowledge of internal design or code. Tests are based on requirements and functionality.
- white box testing - based on knowledge of the internal logic of an application's code. Tests are based on coverage of code statements, branches, paths, conditions.
- unit testing - the most 'micro' scale of testing; to test particular functions or code modules. Typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code. Not always easily done unless the application has a well-designed architecture with tight code; may require developing test driver modules or test harnesses.
- API testing - testing of messaging/data exchange among systems or components of systems. Such testing usually does not involve GUI's (graphical user interfaces). It is often considered a type of 'mid-level' testing.
- incremental integration testing - continuous testing of an application as new functionality is added; requires that various aspects of an application's functionality be independent enough to work separately before all parts of the program are completed, or that test drivers be developed as needed; done by programmers or by testers.
- integration testing - testing of combined parts of an application to determine if they function together correctly. The 'parts' can be code modules, services, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to multi-tier and distributed systems.
- functional testing - black-box type testing geared to functional requirements of an application; this type of testing should be done by testers. This doesn't mean that the programmers shouldn't check that their code works before releasing it (which of course applies to any stage of testing.)
- system testing - black-box type testing that is based on overall requirements specifications; covers all combined parts of a system.
- end-to-end testing - similar to system testing; the 'macro' end of the test scale; involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.
- sanity testing or smoke testing - typically an initial testing effort to determine if a new software version is performing well enough to accept it for a major testing effort. For example, if the new software is crashing systems every 5 minutes, bogging down systems to a crawl, or corrupting databases, the software may not be in a 'sane' enough condition to warrant further testing in its current state.
- regression testing - re-testing after fixes or modifications of the software or its environment. It can be difficult to determine how much re-testing is needed, especially near the end of the development cycle. Automated testing approaches can be especially useful for this type of testing.
- acceptance testing - final testing based on specifications of the end-user or customer, or based on use by end-users/customers over some limited period of time.
- load testing - testing an application under heavy loads, such as testing of a web site under a range of loads to determine at what point the system's response time degrades or fails.
- stress testing - term often used interchangeably with 'load' and 'performance' testing. Also used to describe such tests as system functional testing while under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, large complex queries to a database system, etc.

- performance testing - term often used interchangeably with 'stress' and 'load' testing. Ideally 'performance' testing (and any other 'type' of testing) is defined in requirements documentation or QA or Test Plans.
- usability testing - testing for 'user-friendliness'. Clearly this is subjective, and will depend on the targeted end-user or customer. User interviews, surveys, video recording of user sessions, and other techniques can be used. Programmers and testers are usually not appropriate as usability testers.
- install/uninstall testing - testing of full, partial, or upgrade install/uninstall processes.
- recovery testing - testing how well a system recovers from crashes, hardware failures, or other catastrophic problems.
- failover testing - typically used interchangeably with 'recovery testing'
- security testing - testing how well the system protects against unauthorized internal or external access, willful damage, etc; may require sophisticated testing techniques.
- compatibility testing - testing how well software performs in a particular hardware/software/operating system/network/etc. environment.
- exploratory testing - often taken to mean a creative, informal software test that is not based on formal test plans or test cases; testers may be learning the software as they test it.
- ad-hoc testing - similar to exploratory testing, but often taken to mean that the testers have significant understanding of the software before testing it.
- context-driven testing - testing driven by an understanding of the environment, culture, and intended use of software. For example, the testing approach for life-critical medical equipment software would be completely different than that for a low-cost computer game.
- user acceptance testing - determining if software is satisfactory to an end-user or customer.
- comparison testing - comparing software weaknesses and strengths to competing products.
- alpha testing - testing of an application when development is nearing completion; minor design changes may still be made as a result of such testing. Typically done by end-users or others, not by programmers or testers.
- beta testing - testing when development and testing are essentially completed and final bugs and problems need to be found before final release. Typically done by end-users or others, not by programmers or testers.
- mutation testing - a method for determining if a set of test data or test cases is useful, by deliberately introducing various code changes ('bugs') and retesting with the original test data/cases to determine if the 'bugs' are detected. Proper implementation requires large computational resources.

What are 5 common problems in the software development process?

- poor requirements or user stories - if these are unclear, incomplete, too general, or not testable, there may be problems.
- unrealistic schedule - if too much work is crammed in too little time, problems are inevitable.
- inadequate testing - no one will know whether or not the software is any good until customers complain or systems crash.
- featuritis - requests to add on new features after development goals are agreed on.
- miscommunication - if developers don't know what's needed or customers have erroneous expectations, problems can be expected.

In agile projects, problems often occur when the project diverges from agile principles (such as forgetting that 'Business people and developers must work together daily throughout the project.'

What are 5 common solutions to software development problems?

- solid requirements - clear, complete, detailed, cohesive, attainable, testable requirements that are agreed to by all players. In 'agile'-type environments, continuous close coordination with customers/end-users is necessary to ensure that changing/emerging requirements are understood.
- realistic schedules - allow adequate time for planning, design, testing, bug fixing, re-testing, changes, and documentation; personnel should be able to complete the project without burning out.
- adequate testing - start testing early on, re-test after fixes or changes, plan for adequate time for testing and bug-fixing. 'Early' testing could include static code analysis/testing, test-first development, unit testing by developers, built-in testing and diagnostic capabilities, automated post-build testing, etc.
- stick to initial requirements where feasible - be prepared to defend against excessive changes and additions once development has begun, and be prepared to explain consequences. If changes are necessary, they should be adequately reflected in related schedule changes. If possible, work closely with customers/end-users to manage expectations. In 'agile'-type environments, initial requirements may be expected to change significantly, requiring that true agile processes be in place and followed.
- communication - require walkthroughs and inspections when appropriate; make extensive use of group communication tools - groupware, wiki's, bug-tracking tools and change management tools, etc.; ensure that information/documentation is available and up-to-date; promote teamwork and cooperation; use prototypes, frequent deliveries, and/or continuous communication with end-users if possible to clarify expectations.

What is software 'quality'?

Quality software is reasonably bug-free, delivered on time and within budget, meets requirements and/or expectations, and is maintainable. However, quality is obviously a subjective term. It will depend on who the 'customer' is and their overall influence in the scheme of things. A wide-angle view of the 'customers' of a software development project might include end-users, customer acceptance testers, customer contract officers, customer management, the development organization's management/accountants/testers/salespeople, future software maintenance engineers, stockholders, magazine columnists, etc. Each type of 'customer' will have their own slant on 'quality' - the accounting department might define quality in terms of profits while an end-user might define quality as user-friendly and bug-free.

What is 'good code'?

'Good code' is code that works, is reasonably bug free, secure, and is readable and maintainable. Some organizations have coding 'standards' that all developers are supposed to adhere to, but everyone has different ideas about what's best, or what is too many or too few rules. There are also various theories and metrics, such as McCabe Complexity metrics. It should be kept in mind that excessive use of standards and rules can stifle productivity and creativity. 'Peer reviews', 'buddy checks' pair programming, code analysis tools, etc. can be used to check for problems and enforce standards. For example, in C/C++ coding, here are some typical ideas to consider in setting rules/standards; these may or may not apply to a particular situation:

- minimize or eliminate use of global variables.
- use descriptive function and method names - use both upper and lower case, avoid abbreviations, use as many characters as necessary to be adequately descriptive (use of more than 20 characters is not out of line); be consistent in naming conventions.
- use descriptive variable names - use both upper and lower case, avoid abbreviations, use as many characters as necessary to be adequately descriptive (use of more than 20 characters is not out of line); be consistent in naming conventions.
- function and method sizes should be minimized; less than 100 lines of code is good, less than 50 lines is preferable.
- function descriptions should be clearly spelled out in comments preceding a function's code.
- organize code for readability.
- use whitespace generously - vertically and horizontally
- each line of code should contain 70 characters max.
- one code statement per line.
- coding style should be consistent throughout a program (e.g., use of brackets, indentations, naming conventions, etc.)
- in adding comments, err on the side of too many rather than too few comments; a common rule of thumb is that there should be at least as many lines of comments (including header blocks) as lines of code.
- no matter how small, an application should include documentation of the overall program function and flow (even a few paragraphs is better than nothing); or if possible a separate flow chart and detailed program documentation.
- make extensive use of error handling procedures and status and error logging.
- for C++, to minimize complexity and increase maintainability, avoid too many levels of inheritance in class hierarchies (relative to the size and complexity of the application). Minimize use of multiple inheritance, and minimize use of operator overloading (note that the Java programming language eliminates multiple inheritance and operator overloading.)
- for C++, keep class methods small, less than 50 lines of code per method is preferable.
- for C++, make liberal use of exception handlers

What is 'good design'?

'Design' could refer to many things, but often refers to 'functional design' or 'internal design'. Good internal design is indicated by software code whose overall structure is clear, understandable, easily modifiable, and maintainable; is robust with sufficient error-handling and status logging capability; and works as expected when implemented. Good functional design is indicated by an application whose functionality can be traced back to customer and end-user requirements or user stories. (See further discussion of functional and internal design in FAQ 'What's the big deal about requirements?'). For programs that have a user interface, it's often a good idea to assume that the end user will have little computer knowledge and may not read a user manual or even the on-line help; some common rules-of-thumb include:

- the program should act in a way that least surprises the user
- it should always be evident to the user what can be done next and how to exit
- the program shouldn't let the users do something stupid without warning them

What is SEI? CMM? CMMI? ISO? Will it help?

SEI = 'Software Engineering Institute' at Carnegie-Mellon University; initiated by the U.S. Defense Department to help improve software development processes.

CMM = 'Capability Maturity Model', now called the CMMI ('Capability Maturity Model Integration'), developed by the SEI and as of January 2013 overseen by the CMMI Institute at Carnegie Mellon University. In the 'staged' version, it's a model of 5 levels of process 'maturity' that help determine effectiveness in delivering quality software. CMMI models are "collections of best practices that help organizations to improve their processes." It is geared to larger organizations such as large U.S. Defense Department contractors. However, many of the QA processes involved are appropriate to any organization, and if reasonably applied can be helpful. Organizations can receive CMMI ratings by undergoing assessments by qualified auditors. CMMI V1.3 (2010) supports Agile development processes.

- Level 1 - 'Initial': characterized by chaos, periodic panics, and heroic efforts required by individuals to successfully complete projects. Few if any processes in place; successes may not be repeatable.
- Level 2 - 'Managed': projects carried out in accordance with policies and employ skilled personnel with sufficient resources. Project tracking and reporting is in place. Schedules and budgets are set and revised as needed. Work products are appropriately controlled.
- Level 3 - 'Defined': standard development and maintenance processes are established, integrated consistently throughout an organization,
- Level 4 - 'Quantitatively Managed': metrics are used to track process performance. Project performance is controlled and predictable.
- Level 5 - 'Optimizing': the focus is on continuous process improvement. The impact of new processes and technologies can be predicted and effectively implemented when required. Quality and process objectives are established and regularly revised to reflect changing objectives and organizational performance, and used as criteria in managing process improvement.

What is the 'software life cycle'?

The life cycle begins when an application is first conceived and ends when it is no longer in use. It includes aspects such as initial concept, requirements analysis, functional design, internal design, documentation planning, test planning, coding, document preparation, integration, testing, maintenance, updates, retesting, phase-out, and other aspects.

What makes a good Software Test engineer?

A good test engineer has a 'test to break' attitude, an ability to take the point of view of the customer, a strong desire for quality, and an attention to detail. Tact and diplomacy are useful in maintaining a cooperative relationship with developers, and an ability to communicate with both technical (developers) and non-technical (customers, management) people is useful. Previous software development experience can be helpful as it provides a deeper understanding of the software development process, gives the tester an appreciation for the developers' point of view, and reduce the learning curve in automated test programming. Judgement skills are needed to assess high-risk or critical areas of an application on which to focus testing efforts when time is limited.

What makes a good Software QA engineer?

The same qualities a good tester has are useful for a QA engineer. Additionally, they must be able to understand the entire software development process and how it can fit into the business approach and goals of the organization. Communication skills and the ability to understand various sides of issues are important. In organizations in the early stages of implementing QA processes, patience and diplomacy are especially needed. An ability to find problems as well as to see 'what's missing' is important for inspections and reviews.

What makes a good QA or Test manager?

A good QA, test, or QA/Test(combined) manager should:

- be familiar with the software development process
- be able to maintain enthusiasm of their team and promote a positive atmosphere, despite what is a somewhat 'negative' process (e.g., looking for or preventing problems)
- be able to promote teamwork to increase productivity
- be able to promote cooperation between software, test, and QA engineers
- have the diplomatic skills needed to promote improvements in QA processes
- have the ability to withstand pressures and provide appropriate feedback to other managers when there are issues with quality/processes/schedules/risk
- have people judgement skills for hiring and keeping skilled personnel
- be able to communicate with technical and non-technical people, engineers, managers, and customers.
- be able to run meetings and keep them focused

What's the role of documentation in QA?

Generally, the larger the team/organization, the more useful it will be to stress documentation, in order to manage and communicate more efficiently. (Note that documentation may be electronic, not necessarily in printable form, and may be embedded in code comments, may be embodied in well-written test cases, user stories, etc.) QA practices may be documented to enhance their repeatability. Specifications, designs, business rules, configurations, code changes, test plans, test cases, bug reports, user manuals, etc. may be documented in some form. There would ideally be a system for easily finding and obtaining information and determining what documentation will have a particular piece of information. Change management for documentation can be used where appropriate. For agile software projects, it should be kept in mind that one of the agile values is "Working software over comprehensive documentation", which does not mean 'no' documentation. Agile projects tend to stress the short term view of project needs; documentation often becomes more important in a project's long-term context.

What's the big deal about 'requirements'?

Depending on the project, it may or may not be a 'big deal'. For agile projects, requirements are expected to change and evolve, and detailed documented requirements may not be needed. However some requirements, in the form of user stories or something similar, are useful. For non-agile types of projects detailed documented requirements are usually needed. (Note that requirements documentation can be electronic, not necessarily in the form of printable documents, and may be embedded in code comments, or may be embodied in well-written test

cases, wiki's, user stories, etc.) Requirements are the details describing an application's externally-perceived functionality and properties. Requirements are ideally clear, complete, reasonably detailed, cohesive, attainable, and testable. A non-testable requirement would be, for example, 'user-friendly' (too subjective). A more testable requirement would be something like 'the user must enter their previously-assigned password to access the application'. Determining and organizing requirements details in a useful and efficient way can be a difficult effort; different methods and software tools are available depending on the particular project. Many books are available that describe various approaches to this task.

Care should be taken to involve ALL of a project's significant 'customers' in the requirements process. 'Customers' could be in-house personnel or outside personnel, and could include end-users, customer acceptance testers, customer contract officers, customer management, future software maintenance engineers, salespeople, etc. Anyone who could later derail the success of the project if their expectations aren't met should be included if possible.

Organizations vary considerably in their handling of requirements specifications. Often the requirements are spelled out in a document with statements such as 'The product shall.....'. 'Design' specifications should not be confused with 'requirements'. It can be helpful to have design specifications traceable back to the requirements.

In some organizations requirements may end up in high level project plans, functional specification documents, in design documents, or in other documents at various levels of detail. No matter what they are called, some type of documentation with requirements, user stories, and related information will be useful to testers in order to properly plan and execute tests (manual or automated). Without such documentation, there will be no clear-cut way to determine if software is performing correctly.

If testable requirements are not available or are only partially available, useful testing can still be performed. In this situation test results may be more oriented to providing information about the state of the software and risk levels, rather than providing pass/fail results. A relevant testing approach in this situation may include approaches such as 'exploratory testing'. Many software projects have a mix of documented testable requirements, poorly documented requirements, undocumented requirements, and changing requirements. In such projects a mix of scripted and exploratory testing approaches may be useful.

'Agile' approaches use methods requiring close interaction and cooperation between programmers and stakeholders/customers/end-users to iteratively develop requirements, user stories, etc. In the XP 'test first' approach developers create automated unit testing code before the application code, and these automated unit tests can essentially embody the requirements.

What steps are needed to develop and run software tests?

The following are some of the steps to consider:

- Obtain requirements, functional design, internal design specifications, user stories, or other available/necessary information
- Obtain budget and schedule requirements
- Determine project-related personnel and their responsibilities, reporting requirements, required standards and processes (such as release processes, change processes, etc.)

- Determine project context, relative to the existing quality culture of the product/organization/business, and how it might impact testing scope, approaches, and methods.
- Identify the application's higher-risk and more important aspects, set priorities, and determine scope and limitations of tests.
- Determine test approaches and methods - unit, integration, functional, system, security, load, usability tests, whichever are in scope.
- Determine test environment requirements (hardware, software, configuration, versions, communications, etc.)
- Determine testware requirements (automation tools, coverage analyzers, test tracking, problem/bug tracking, etc.)
- Determine test input data requirements
- Identify tasks, those responsible for tasks, and labor requirements
- Set initial schedule estimates, timelines, milestones where feasible.
- Determine, where appropriate, input equivalence classes, boundary value analyses, error classes
- Prepare test plan document(s) and have needed reviews/approvals
- Write test cases or test scenarios as needed.
- Have needed reviews/inspections/approvals of test cases/scenarios/approaches.
- Prepare test environment and testware, obtain needed user manuals/reference documents/configuration guides/installation guides, set up test tracking processes, set up logging and archiving processes, set up or obtain test input data
- Obtain/install/configure software releases
- Perform tests
- Evaluate and report results
- Track problems/bugs and fixes
- Retest as needed
- Maintain and update test plans, test cases, test environment, and testware through life cycle

What's a 'test plan'?

A software project test plan is a document that describes the objectives, scope, approach, and focus of a software testing effort. The process of preparing a test plan is a useful way to think through the efforts needed to validate the acceptability of a software product. The completed document will help people outside the test group understand the 'why' and 'how' of product validation. It should be thorough enough to be useful but not so overly detailed that no one outside the test group will read it. The following are some of the items that might be included in a test plan, depending on the particular project:

- Title
- Identification of software including version/release numbers
- Revision history of document including authors, dates, approvals
- Table of Contents
- Purpose of document, intended audience
- Objective of testing effort
- Software product overview
- Relevant related document list, such as requirements, design documents, other test plans, etc.
- Relevant standards or legal requirements
- Traceability requirements

- Relevant naming conventions and identifier conventions
- Overall software project organization and personnel/contact-info/responsibilities
- Test organization and personnel/contact-info/responsibilities
- Assumptions and dependencies
- Project risk analysis
- Testing priorities and focus
- Scope and limitations of testing
- Test outline - a decomposition of the test approach by test type, feature, functionality, process, system, module, etc. as applicable
- Outline of data input equivalence classes, boundary value analysis, error classes
- Test environment - hardware, operating systems, other required software, data configurations, interfaces to other systems
- Test environment validity analysis - differences between the test and production systems and their impact on test validity.
- Test environment setup and configuration issues
- Software migration processes
- Software CM processes
- Test data setup requirements
- Database setup requirements
- Outline of system-logging/error-logging/other capabilities, and tools such as screen capture software, that will be used to help describe and report bugs
- Discussion of any specialized software or hardware tools that will be used by testers to help track the cause or source of bugs
- Test automation - justification and overview
- Test tools to be used, including versions, patches, etc.
- Test script/test code maintenance processes and version control
- Problem tracking and resolution - tools and processes
- Project test metrics to be used
- Reporting requirements and testing deliverables
- Software entrance and exit criteria
- Initial sanity testing period and criteria
- Test suspension and restart criteria
- Personnel allocation
- Personnel pre-training needs
- Test site/location
- Outside test organizations to be utilized and their purpose, responsibilties, deliverables, contact persons, and coordination issues
- Relevant proprietary, classified, security, and licensing issues.
- Open issues
- Appendix - glossary, acronyms, etc.

What's a 'test case'?

A test case describes an input, action, or event and an expected response, to determine if a feature of a software application is working correctly. A test case may contain particulars such as test case identifier, test case name, objective, test conditions/setup, input data requirements, steps, and expected results. The level of detail may vary significantly depending on the organization and project context. Note that organizations vary considerably in their handling of test cases; many utilize less-detailed 'test scenarios' that allow for simpler and more adaptable/maintainable test documentation.

Note that the process of developing test cases can help find problems in the requirements or design of an application, since it requires completely thinking through the operation of the application. For this reason, it's useful to prepare test cases early in the development cycle if possible.

What should be done after a bug is found?

The bug needs to be communicated and assigned to developers that can fix it. After the problem is resolved, fixes should be re-tested, and determinations made regarding requirements for regression testing to check that fixes didn't create problems elsewhere. If a problem-tracking system is in place, it should encapsulate these processes. A variety of commercial problem-tracking/management software tools are available (see the 'Tools' section for web resources with listings of such tools). The following are items to consider in the tracking process:

- Complete information such that developers can understand the bug, get an idea of it's severity, and reproduce it if necessary.
- Bug identifier (number, ID, etc.)
- Current bug status (e.g., 'Released for Retest', 'New', etc.)
- The application name or identifier and version
- The function, module, feature, object, screen, etc. where the bug occurred
- Environment specifics, system, platform, relevant hardware specifics
- Test case or scenario information/name/number/identifier
- One-line bug description
- Full bug description
- Description of steps needed to reproduce the bug if not covered by a test case or automated test or if the developer doesn't have easy access to the test case/test script/test tool
- Names and/or descriptions of file/data/messages/etc. used in test
- File excerpts/error messages/log file excerpts/screen shots/test tool logs that would be helpful in finding the cause of the problem
- Severity estimate (a 5-level range such as 1-5 or 'critical'-to-'low' is common)
- Was the bug reproducible?
- Tester name
- Test date
- Bug reporting date
- Name of developer/group/organization the problem is assigned to
- Description of problem cause
- Description of fix
- Code section/file/module/class/method that was fixed
- Date of fix
- Application version that contains the fix
- Tester responsible for retest
- Retest date
- Retest results
- Regression testing requirements
- Tester responsible for regression tests
- Regression testing results

A reporting or tracking process should enable notification of appropriate personnel at various stages. For instance, testers need to know when retesting is needed, developers need to know

when bugs are found and how to get the needed information, and reporting/summary capabilities are needed for managers.

What is 'configuration management'?
Configuration management covers the processes used to control, coordinate, and track: code, requirements, documentation, problems, change requests, designs, tools/compilers / libraries / patches, changes made to them, and who makes the changes.

What if the software is so buggy it can't really be tested at all?
The best bet in this situation is for the testers to go through the process of reporting whatever bugs or blocking-type problems initially show up, with the focus being on critical bugs. Since this type of problem can significantly affect schedules, and indicates deeper problems in the software development process (such as insufficient unit testing or insufficient integration testing, poor design, improper build or release procedures, etc.) managers should be notified, and provided with some documentation as evidence of the problem.

How can it be known when to stop testing?
This can be difficult to determine. Most modern software applications are so complex, and run in such an interdependent environment, that complete testing can never be done. Common factors in deciding when to stop are:

- Deadlines (release deadlines, testing deadlines, etc.)
- Test cases completed with certain percentage passed
- Test budget depleted
- Coverage of code/functionality/requirements reaches a specified point
- Bug rate falls below a certain level
- Beta or alpha testing period ends

What if there isn't enough time for thorough testing?
Use risk analysis, along with discussion with project stakeholders, to determine where testing should be focused. Since it's rarely possible to test every possible aspect of an application, every possible combination of events, every dependency, or everything that could go wrong, risk analysis is appropriate to most software development projects. This requires judgement skills, common sense, and experience. (If warranted, formal methods are also available.) Considerations can include:

- Which functionality is most important to the project's intended purpose?
- Which functionality is most visible to the user?
- Which functionality has the largest safety impact?
- Which functionality has the largest financial impact on users?
- Which aspects of the application are most important to the customer?
- Which aspects of the application can be tested early in the development cycle?
- Which parts of the code are most complex, and thus most subject to errors?
- Which parts of the application were developed in rush or panic mode?
- Which aspects of similar/related previous projects caused problems?
- Which aspects of similar/related previous projects had large maintenance expenses?
- Which parts of the requirements and design are unclear or poorly thought out?
- What do the developers think are the highest-risk aspects of the application?
- What kinds of problems would cause the worst publicity?

- What kinds of problems would cause the most customer service complaints?
- What kinds of tests could easily cover multiple functionalities?
- Which tests will have the best high-risk-coverage to time-required ratio?

What if the project isn't big enough to justify extensive testing?
Consider the impact of project errors, not the size of the project. However, if extensive testing is still not justified, risk analysis is again needed and the same considerations as described previously in 'What if there isn't enough time for thorough testing?' apply. The tester might then do ad hoc or exploratory testing, or write up a limited test plan based on the risk analysis.

How do distributed multi-tier environments affect testing?
Most current software being tested involves multi-tier and distributed applications which can be highly complex due to the multiple dependencies among systems, services, data communications, hardware, and servers. Thus testing requirements can be extensive. When time is limited (as it usually is) a focus on integration and system testing can be considered. Additionally, load/stress/performance testing may be useful in determining distributed application limitations and capabilities and where the limitations are. There are commercial and open source tools to assist with such testing

How should Web sites be tested?
Many modern web sites are essentially complex distributed systems with html, css, web services, encrypted communications, browser-side scripts/apps/libraries (such as javascript, flash, etc), the wide variety of applications/libraries/datastores that could run on the server side, load balancers, etc. Additionally, there are a wide variety of servers and browsers, mobile and other platforms, various versions of each, small but sometimes significant differences between them, variations in connection speeds, rapidly changing technologies, and multiple standards and protocols. Although web site testing was initially relatively simple years ago, testing of modern web site front ends, back end systems, mid-level tiers, web services, databases, security, performance, etc, can be as complex as or more complex than any other type of application.

To assist in testing of web sites via their GUI's, many popular web browsers normally include a set of 'Developer Tools' that are helpful in testing and debugging, and in developing test automation scripts:

How is testing affected by object-oriented designs?
Well-engineered object-oriented design can make it easier to trace from code to internal design to functional design to requirements. While there will be little effect on black box testing (where an understanding of the internal design of the application is unnecessary), white-box testing can be oriented to the application's objects, methods, etc. If the application was well-designed this can simplify test design and test automation design.

What is Agile Software Development and how does it impact testing?
Agile Software Development generally refers to incremental, collaborative software development approaches that provide alternatives to 'heavyweight', documentation-driven,

waterfall-type development practices. It grew out of such approaches as Extreme Programming, SCRUM, DSDM, Crystal, and other 'lightweight' methodologies. In 2001 a group of software development and test practitioners gathered to discuss lightweight methods and created the 'Agile Manifesto' which describes the Agile approach values and lists 12 principles that describe Agile software development. In reality many organizations implement these principles to widely varying degrees (and with widely varying degrees of success) and still call their approach 'Agile'. The impact of Agile approaches on software testing can also vary widely but often includes the following:

- Requirements and documentation are often minimal and when present are often in the form of high-level 'user stories' and 'acceptance tests'.
- Requirements can be added or changed often
- Iterative development/test cycles ('sprints') are often in the range of 1-3 weeks. Both new functionality testing and regression testing (preferably automated) may occur within each iterative cycle.
- Close collaboration between testers and developers, product owners and other team members
- Short daily project status 'standup' meetings that include testers.
- Common testing-related practices in agile projects may include test-driven development, extensive unit testing and unit test automation, API-level test automation, exploratory and session-based testing, UI test automation.
- Testers may be heavily involved in fleshing out requirements details, including both functional and non-functional requirements.

Why is it sometimes hard for organizations to get serious about quality assurance? Solving problems is a high-visibility process; preventing problems is low-visibility. This is illustrated by an old parable:

In ancient China there was a family of healers, one of whom was known throughout the land and employed as a physician to a great lord. The physician was asked which of his family was the most skillful healer. He replied,

"I tend to the sick and dying with drastic and dramatic treatments, and on occasion someone is cured and my name gets out among the lords."

"My elder brother cures sickness when it just begins to take root, and his skills are known among the local peasants and neighbors."

"My eldest brother is able to sense the spirit of sickness and eradicate it before it takes form. His name is unknown outside our home."

This is a problem in any business, but it's a particularly difficult problem in the software industry. Software quality problems are often not as readily apparent as they might be in the case of an industry with more physical products, such as auto manufacturing or home construction.

Additionally: Many organizations are able to determine who is skilled at fixing problems, and then reward such people. However, determining who has a talent for preventing problems in the first place, and figuring out how to incentivize such behavior, is a significant challenge.

Who is responsible for risk management?
Risk management means the actions taken to avoid things going wrong on a software development project, things that might negatively impact the scope, quality, timeliness, or cost of a project. This is, of course, a shared responsibility among everyone involved in a project. However, there needs to be a 'buck stops here' person who can consider the relevant tradeoffs when decisions are required, and who can ensure that everyone is handling their risk management responsibilities.

It is not unusual for the term 'risk management' to never come up at all in a software organization or project. If it does come up, it's often assumed to be the responsibility of QA or test personnel. Or there may be a 'risks' or 'issues' section of a project, QA, or test plan, and it's assumed that this means that risk management has taken place.

The issues here are similar to those for the LFAQ question "Who should decide when software is ready to be released?" It's generally NOT a good idea for a test lead, test manager, or QA manager to be the 'buck stops here' person for risk management. Typically QA/Test personnel or managers are not managers of developers, analysts, designers and many other project personnel, and so it would be difficult for them to ensure that everyone on a project is handling their risk management responsibilities. Additionally, knowledge of all the considerations that go into risk management mitigation and tradeoff decisions is rarely the province of QA/Test personnel or managers. Based on these factors, the project manager is usually the most appropriate 'buck stops here' risk management person. QA/Test personnel can, however, provide input to the project manager. Such input could include analysis of quality-related risks, risk monitoring, process adherence reporting, defect reporting, and other information.

Who should decide when software is ready to be released?
In many projects this depends on the release criteria for the software. Such criteria are often in turn based on the decision to end testing, discussed in FAQ #2 item "How can it be known when to stop testing?" Unfortunately, for any but the simplest software projects, it is nearly impossible to adequately specify useful criteria without a significant amount of assumptions and subjectivity. For example, if the release criteria are based on passing a certain set of tests, there is likely an assumption that the tests have adequately addressed all appropriate software risks. For most software projects, this would of course be impossible without enormous expense, so this assumption would be a large leap of faith. Additionally, since most software projects involve a balance of quality, timeliness, and cost, testing alone cannot address how to balance all three of these competing factors when release decisions are needed.

A typical approach is for a lead tester or QA or Test manager to be the release decision maker. This again involves significant assumptions - such as an assumption that the test manager understands the spectrum of considerations that are important in determining whether software quality is 'sufficient' for release, or the assumption that quality does not have to be balanced with timeliness and cost. In many organizations, 'sufficient quality' is not well defined, is extremely subjective, may have never been usefully discussed, or may vary from project to project or even from day to day.

Release criteria considerations can include deadlines, sales goals, business/market/competitive considerations, business segment quality norms, legal requirements, technical and programming considerations, end-user expectations, internal budgets, impacts on other organization projects or goals, and a variety of other factors. Knowledge of all these factors is

often shared among a number of personnel in a large organization, such as the project manager, director, customer service manager, technical lead or manager, marketing manager, QA manager, etc. In smaller organizations or projects it may be appropriate for one person to be knowledgeable in all these areas, but that person is typically a project manager, not a test lead or QA manager.

For these reasons, it's generally not a good idea for a test lead, test manager, or QA manager to decide when software is ready to be released. Their responsibility should be to provide input to the appropriate person or group that makes a release decision. For small organizations and projects that person could be a product manager, a project manager, or similar manager. For larger organizations and projects, release decisions might be made by a committee of personnel with sufficient collective knowledge of the relevant considerations.

What can be done if requirements are changing continuously?
This is a common problem for organizations where there are expectations that requirements can be pre-determined and remain stable. If these expectations are reasonable, here are some approaches:

- Work with the project's stakeholders early on to understand how requirements might change so that alternate test plans and strategies can be worked out in advance, if possible.
- It's helpful if the application's initial design allows for some adaptability so that later changes do not require redoing the application from scratch.
- If the code is well-commented and well-documented this makes changes easier for the developers.
- Use some type of rapid prototyping whenever possible to help customers feel sure of their requirements and minimize changes.
- The project's initial schedule should allow for some extra time commensurate with the possibility of changes.
- Try to move new requirements to a 'Phase 2' version of an application, while using the original requirements for the 'Phase 1' version.
- Negotiate to allow only easily-implemented new requirements into the project, while moving more difficult new requirements into future versions of the application.
- Be sure that customers and management understand the scheduling impacts, inherent risks, and costs of significant requirements changes. Then let management or the customers (not the developers or testers) decide if the changes are warranted - after all, that's their job.
- Balance the effort put into setting up automated testing with the expected effort required to refactor them to deal with changes.
- Try to design some flexibility into automated test scripts.
- Focus initial automated testing on application aspects that are most likely to remain unchanged.
- Devote appropriate effort to risk analysis of changes to minimize regression testing needs.
- Design some flexibility into test cases (this is not easily done; the best bet might be to minimize the detail in the test cases, or set up only higher-level generic-type test plans)
- Focus less on detailed test plans and test cases and more on ad hoc testing (with an understanding of the added risk that this entails).

If this is a continuing problem, and the expectation that requirements can be pre-determined and remain stable is NOT reasonable, it may be a good idea to figure out why the expectations

are not aligned with reality, and to refactor an organization's or project's software development process to take this into account. It may be appropriate to consider agile development approaches.

What if the application has functionality that wasn't in the requirements?
It may take serious effort to determine if an application has significant unexpected or hidden functionality, and it could indicate deeper problems in the software development process. If the functionality isn't necessary to the purpose of the application, it should be removed, as it may have unknown impacts or dependencies that were not taken into account by the designer or the customer. (If the functionality is minor and low risk then no action may be necessary.) If not removed, information will be needed to determine risks and to determine any added testing needs or regression testing needs. Management should be made aware of any significant added risks as a result of the unexpected functionality.

This problem is a standard aspect of projects that include COTS (Commercial Off-The-Shelf) software or modified COTS software. The COTS part of the project will typically have a large amount of functionality that is not included in project requirements, or may be simply undetermined. Depending on the situation, it may be appropriate to perform in-depth analysis of the COTS software and work closely with the end user to determine which pre-existing COTS functionality is important and which functionality may interact with or be affected by the non-COTS aspects of the project. A significant regression testing effort may be needed (again, depending on the situation), and automated regression testing may be useful.

How can Software QA processes be implemented without reducing productivity?
By implementing QA processes slowly over time, using consensus to reach agreement on processes, focusing on processes that align tightly with organizational goals, and adjusting/experimenting/refactoring as an organization matures, productivity can be improved instead of stifled. Problem prevention will lessen the need for problem detection, panics and burn-out will decrease, and there will be improved focus and less wasted effort. At the same time, attempts should be made to keep processes simple and efficient, avoid a 'Process Police' mentality, minimize paperwork, promote computer-based processes and automated tracking and reporting, minimize time required in meetings, and promote training as part of the QA process. However, no one - especially talented technical types - likes rules or bureaucracy, and in the short run things may slow down a bit. A typical scenario would be that more days of planning, reviews, and inspections will be needed, but less time will be required for late-night bug-fixing and handling of irate customers.

Other possibilities include incremental self-managed team approaches such as 'Kaizen' methods of continuous process improvement, the Deming-Shewhart Plan-Do-Check-Act cycle, and others.

What if an organization is growing so fast that fixed QA processes are impossible?
This is a common problem in the software industry, especially in new technology areas. There is generally no easy solution in this situation. One approach is:

- Hire good people
- Management should 'ruthlessly prioritize' quality issues and maintain focus on the customer
- Everyone in the organization should be clear on what 'quality' means to the customer

Depending on the growth rate, it is possible that incremental self-managed team approaches may be applicable, such as 'Kaizen' methods of continuous process improvement, or the Deming-Shewhart Plan-Do-Check-Act cycle, and others.

Will automated testing tools make testing easier?

- Possibly. For small projects, the time needed to learn and implement them may not be worth it unless personnel are already familiar with the tools. For larger projects, or ongoing long-term projects they can be valuable.
- Most test automation tools utilize a standard coding language such as ruby, python, Java, etc or a proprietary scripting language specific to the tool. Sometimes initial tests can be 'recorded' such that the test scripts are automatically generated. and then modified as needed. One of the challenges of automated testing is that if there are continual changes to the system being tested, the test automation code may have to be changed so often that it becomes very time-consuming (thus expensive) to continuously update the scripts. Additionally, interpretation and analysis of results (screens, reports, data, logs, etc.) can be a difficult task. Note that there are test automation tools and frameworks for web and UI interfaces as well as text-based and back-end interfaces, and for all types of platforms.
- A common type of approach for automation of functional testing is 'data-driven' or 'keyword-driven' automated testing, in which the test drivers are separated from the data and/or actions utilized in testing (an 'action' would be something like 'enter a value in a text box'). Test drivers can be in the form of automated test tools or frameworks or custom-written testing software. The data and actions can be more easily maintained - such as via a spreadsheet - since they are separate from the test drivers. The test drivers 'read' the data/action information to perform specified tests. This approach can enable more efficient control, development, documentation, and maintenance of automated tests/test cases.
- Other automated tools can include:
  - code analyzers - monitor code complexity, adherence to standards, etc.
  - coverage analyzers - these tools check which parts of the code have been exercised by a test, and may be oriented to code statement coverage, condition coverage, path coverage, etc.
  - memory analyzers - such as bounds-checkers and leak detectors.
  - load/performance test tools - for testing client/server and web applications under various load levels.
  - web test tools - to check that links are valid, HTML code usage is correct, client-side and server-side programs work, a web site's interactions are secure.
  - other tools - for test case management, BDT (behavior-driven testing), documentation, management, bug reporting, and configuration management, file and database comparisons, screen captures, security testing, macro recorders, etc.

Test automation is, of course, possible without COTS tools. Many successful automation efforts utilize open source tools, or custom automation software that is targeted for specific projects, specific software applications, or a specific organization's software development environment. In test-driven agile software development environments, automated tests are often built into the software during (or preceding) coding of the application.

What's the best way to choose a test automation tool?
It's easy to get caught up in enthusiasm for the 'silver bullet' of test automation, where the dream is that a single mouse click can initialize thorough unattended testing of an entire software application, bugs will be automatically reported, and easy-to-understand summary reports will be waiting in the manager's in-box in the morning.
Although that may in fact be possible in some situations, it is not the way things generally play out.

In manual testing, the test engineer exercises software functionality to determine if the software is behaving in an expected way. This means that the tester must be able to judge what the expected outcome of a test should be, such as expected data outputs, screen messages, changes in the appearance of a User Interface, XML files, database changes, etc. In an automated test, the computer does not have human-like 'judgement' capabilities to determine whether or not a test outcome was correct. This means there must be a mechanism by which the computer can do an automatic comparison between actual and expected results for every automated test scenario and unambiguously make a pass or fail determination. This factor may require a significant change in the entire approach to testing, since in manual testing a human is involved and can:

- make mental adjustments to expected test results based on variations in the pre-test state of the software system
- often make on-the-fly adjustments, if needed, to data used in the test
- make pass/fail judgements about results of each test
- make quick judgements and adjustments for changes to requirements.
- make a wide variety of other types of judgements and adjustments as needed.

For those new to test automation, it might be a good idea to do some reading or training first. There are a variety of ways to go about doing this; some example approaches are:

- Read through information on the web about test automation such as general information available on some test tool vendor sites or some of the automated testing articles listed in the 'Other Resources' Automation section.
- Read some books on test automation such as those listed in the Softwaregatest.com Bookstore
- Obtain some test tool trial versions or low cost or open source test tools and experiment with them
- Attend software testing conferences or training courses related to test automation

As in anything else, proper planning and analysis are critical to success in choosing and utilizing an automated test tool. Choosing a test tool just for the purpose of 'automating testing' is not useful; useful purposes might include: testing more thoroughly, testing in ways that were not previously feasible via manual methods (such as load testing), testing faster, enabling continuous integration processes, or reducing excessively tedious manual testing. Automated testing rarely enables savings in the cost of testing, although it may result in software lifecycle savings (or increased sales) just as with any other quality-related initiative.
With the proper background and understanding of test automation, the following considerations can be helpful in choosing a test tool (automated testing will not necessarily resolve them, they are only considerations for automation potential):

- Analyze the current non-automated testing situation to determine where testing is not being done or does not appear to be sufficient
- Where is current testing excessively time-consuming?
- Where is current testing excessively tedious?
- What kinds of problems are repeatedly missed with current testing?
- What testing procedures are carried out repeatedly (such as regression testing or security testing)?
- What testing procedures are not being carried out repeatedly but should be?
- What test tracking and management processes can be implemented or made more effective through the use of an automated test tool?

Taking into account the testing needs determined by analysis of these considerations and other appropriate factors, the types of desired test tools can be determined. For each type of test tool (such as functional test tool, load test tool, etc.) the choices can be further narrowed based on the characteristics of the software application. The relevant characteristics will depend, of course, on the situation and the type of test tool and other factors. Such characteristics could include the operating system, GUI components, development languages, web server type, etc. Other factors affecting a choice could include experience level and capabilities of test personnel, advantages/disadvantages in developing a custom automated test tool, tool costs, tool quality and ease of use, usefulness of the tool on other projects, etc.

Once a short list of potential test tools is selected, several can be utilized on a trial basis for a final determination. Any expensive test tool should be thoroughly analyzed during its trial period to ensure that it is appropriate and that it's capabilities and limitations are well understood. This may require significant time or training, but the alternative is to take a major risk of a mistaken investment (in terms of time, resources, and/or purchase price).

How can it be determined if a test environment is appropriate?
This is a difficult question in that it typically involves tradeoffs between 'better' test environments and cost. The ultimate situation would be a collection of test environments that mimic exactly all possible hardware, software, network, data, and usage characteristics of the expected live environments in which the software will be used. For many software applications, this would involve a nearly infinite number of variations, and would clearly be impossible. And for new software applications, it may also be impossible to predict all the variations in environments in which the application will run. For very large, complex systems, duplication of a 'live' type of environment may be prohibitively expensive.

In reality judgements must be made as to which characteristics of a software application environment are important, and test environments can be selected on that basis after taking into account time, budget, and logistical constraints. Such judgements are preferably made by those who have the most appropriate technical knowledge and experience, along with an understanding of risks and constraints.

For smaller or low risk projects, an informal approach is common, but for larger or higher risk projects (in terms of money, property, or lives) a more formalized process involving multiple personnel and significant effort and expense may be appropriate.

In some situations it may be possible to mitigate the need for maintenance of large numbers of varied test environments. One approach might be to coordinate internal testing with beta testing

efforts. Another possible mitigation approach is to provide built-in automated tests that run automatically upon installation of the application by end-users. These tests might then automatically report back information, via the internet, about the application environment and problems encountered. Another possibility is the use of virtual environments instead of physical test environments, using such tools as VMWare or VirtualBox.

What's the best approach to software test estimation?
There is no simple answer for this. The 'best approach' is highly dependent on the particular organization and project and the experience of the personnel involved.
For example, given two software projects of similar complexity and size, the appropriate test effort for one project might be very large if it was for life-critical medical equipment software, but might be much smaller for the other project if it was for a low-cost computer game. A test estimation approach that only considered size and complexity might be appropriate for one project but not for the other. Following are some approaches to consider:

Implicit Risk Context Approach:
A typical approach to test estimation is for a project manager or QA manager to implicitly use risk context, in combination with past personal experiences in the organization, to choose a level of resources to allocate to testing. In many organizations, the 'risk context' is assumed to be similar from one project to the next, so there is no explicit consideration of risk context. (Risk context might include factors such as the organization's typical software quality levels, the software's intended use, the experience level of developers and testers, etc.) This is essentially an intuitive guess based on experience.

Metrics-Based Approach:
A useful approach is to track past experience of an organization's various projects and the associated test effort that worked well for projects. Once there is a set of data covering characteristics for a reasonable number of projects, then this 'past experience' information can be used for future test project planning. (Determining and collecting useful project metrics over time can be an extremely difficult task.) For each particular new project, the 'expected' required test time can be adjusted based on whatever metrics or other information is available, such as function point count, number of external system interfaces, unit testing done by developers, risk levels of the project, etc. In the end, this is essentially 'judgement based on documented experience', and is not easy to do successfully.

Test Work Breakdown Approach:
Another common approach is to decompose the expected testing tasks into a collection of small tasks for which estimates can, at least in theory, be made with reasonable accuracy. This of course assumes that an accurate and predictable breakdown of testing tasks and their estimated effort is feasible. In many large projects, this is not the case. For example, if a large number of bugs are being found in a project, this will add to the time required for testing, retesting, bug analysis and reporting. It will also add to the time required for development, and if development schedules and efforts do not go as planned, this will further impact testing.

Iterative Approach:
In this approach for large test efforts, an initial rough testing estimate is made. Once testing begins, a more refined estimate is made after a small percentage (e.g., 1%) of the first estimate's work is done. At this point testers have obtained additional test project knowledge and a better understanding of issues, general software quality, and risk. Test plans and

schedules can be refactored if necessary and a new estimate provided. Then a yet-more-refined estimate is made after a somewhat larger percentage (e.g., 2%) of the new work estimate is done. Repeat the cycle as necessary/appropriate.

Percentage-of-Development Approach:
Some organizations utilize a quick estimation method for testing based on the estimated programming effort. For example, if a project is estimated to require 1000 hours of programming effort, and the organization normally finds that a 40% ratio for testing is appropriate, then an estimate of 400 hours for testing would be used. This approach may or may not be useful depending on the project-to-project variations in risk, personnel, types of applications, levels of complexity, etc.

Successful test estimation is a challenge for most organizations, since few can accurately estimate software project development efforts, much less the testing effort of a project. It is also difficult to attempt testing estimates without first having detailed information about a project, including detailed requirements, the organization's experience with similar projects in the past, and an understanding of what should be included in a 'testing' estimation for a project (functional testing? unit testing? reviews? inspections? load testing? security testing?)

With agile software development approaches, test effort estimations may be unnecessary if pure test-driven development is utilized. However, it is not uncommon to have a mix of some automated positive-type unit tests, along with some type of separate manual or automated functional testing. In general, agile-based projects by their nature will not be heavily dependent on large one-shot testing efforts, since they emphasize the construction of releasable software in short iteration cycles. Test estimates are often focused on individual 'stories' and the testing associated with each of these. These smaller multiple test effort estimates may not be as difficult to estimate and the impact of inaccurate estimates will be less severe, and expectations are that estimates will improve with each sprint.