|  |
| --- |
| ***Manual Testing Interview Questions*** |
| **1. 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](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-1-354.asp) 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 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 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.   **2.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. (See the 'Tools' section for web resources with listings of configuration management tools. Also see the Bookstore section's 'Configuration Management' category for useful [books](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-1-354.asp) with more information.)   **3.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 severely affect [schedules](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-1-354.asp), and indicates deeper problems in the software [development http://kona.kontera.com/javascript/lib/imgs/grey_loader.gif](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-1-354.asp" \t "undefined)  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.   **4.How can it be known when to stop testing?**  This can be difficult to determine. Many 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](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-1-354.asp) or alpha testing period ends  **5.What if there isn't enough time for thorough testing?**  Use risk analysis 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?  **6.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 testing, or write up a limited test plan based on the risk analysis.   **7.What can be done if requirements are changing continuously?**  A common problem and a major headache.  • 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 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 re-do 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).   **8.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 would 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](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-1-354.asp) by the designer or the customer. If not removed, design information will be needed to determine added testing needs or regression testing needs. Management should be made aware of any significant added risks as a result of the unexpected functionality. If the functionality only effects areas such as minor improvements in the user interface, for example, it may not be a significant risk. |

|  |
| --- |
| **Manual Testing Interview Questions** |
| **1.What makes a good 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 tool programming. Judgement skills are needed to assess high-risk areas of an application on which to focus testing efforts when time is limited.   **2.What makes a good** [**Software**](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-355.asp) **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.   **3.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 say 'no' to other managers when quality is insufficient or QA processes are not being adhered to  • 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  **4.What's the role of documentation in QA?** Critical. (Note that documentation can be electronic, not necessarily paper.) 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. There should ideally be a system for easily finding and obtaining documents and determining what documentation will have a particular piece of information. Change management for documentation should be used if possible.   **5.What's the big deal about 'requirements'?** One of the most reliable methods of insuring problems, or failure, in a complex software project is to have poorly documented requirements specifications. Requirements are the details describing an application's externally-perceived functionality and properties. Requirements should be clear, complete, reasonably detailed, cohesive, attainable, and testable. A non-testable requirement would be, for example, 'user-friendly' (too subjective). A 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 are available depending on the particular project. Many books are available that describe various approaches to this task. (See the Bookstore section's 'Software Requirements Engineering' category for books on Software Requirements.)  Care should be taken to involve ALL of a project's significant 'customers' in the requirements process. 'Customers' could be in-house personnel or out, 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 project if their expectations aren't met should be included if possible.  Organizations vary considerably in their handling of requirements specifications. Ideally, the requirements are spelled out in a document with statements such as 'The product shall.....'. 'Design' specifications should not be confused with 'requirements'; design specifications should be 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 detailed requirements will be needed by testers in order to properly plan and execute tests. Without such documentation, there will be no clear-cut way to determine if a software application is performing correctly.  'Agile' methods such as XP use methods requiring close interaction and cooperation between [programmers](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-355.asp) and customers/end-users to iteratively develop requirements. The programmer uses 'Test first' [development](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-355.asp) to first create automated unit testing code, which essentially embodies the requirements.   **6.What steps are needed to develop and run software tests?** The following are some of the steps to consider:  • Obtain requirements, functional design, and internal design specifications and other necessary [documents](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-355.asp)  • 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.)  • Identify application's higher-risk aspects, set priorities, and determine scope and limitations of tests  • Determine test approaches and methods - unit, integration, functional, system, load, usability tests, etc.  • Determine test environment requirements (hardware, software, communications, etc.)  • Determine testware requirements (record/playback tools, coverage analyzers, test tracking, problem/bug tracking, etc.)  • Determine test input data requirements  • Identify tasks, those responsible for tasks, and labor requirements  • Set schedule estimates, timelines, milestones  • Determine input equivalence classes, boundary value analyses, error classes  • Prepare test plan document and have needed reviews/approvals  • Write test cases  • Have needed reviews/inspections/approvals of test cases  • 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 and install 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  **7.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 thorough 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/responsibilties  • 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](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-355.asp) 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](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-355.asp), and licensing issues.  • Open issues  • Appendix - glossary, acronyms, etc.  (See the Bookstore section's 'Software Testing' and 'Software QA' categories for useful books with more information.)   **8.What's a 'test case'?** • 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. A test case should contain particulars such as test case identifier, test case name, objective, test conditions/setup, input data requirements, steps, and expected results.  • 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  **1.How can Software QA processes be implemented without stifling productivity?**  By implementing QA processes slowly over time, using consensus to reach agreement on processes, and adjusting and experimenting as an organization grows and matures, productivity will 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, 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 bureacracy, and in the short run things may slow down a bit. A typical scenario would be that more days of planning and development will be needed, but less time will be required for late-night bug-fixing and calming of irate customers.   **2.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 no easy solution in this situation, other than:  • 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   **3.How does a client/server** [**environment**](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-2-353.asp) **affect testing?**  Client/server applications can be quite complex due to the multiple dependencies among clients, data communications, hardware, and servers. Thus testing requirements can be extensive. When time is limited (as it usually is) the focus should be on integration and system testing. Additionally, load/[stress](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-2-353.asp)/performance testing may be useful in determining client/server application limitations and capabilities. There are [commercial](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-2-353.asp) tools to assist with such testing. (See the 'Tools' section for web resources with listings that include these kinds of test tools.)   **4.How can World Wide Web sites be tested?**  Web sites are essentially client/server applications - with web servers and 'browser' clients. Consideration should be given to the interactions between html pages, TCP/IP communications, Internet connections, firewalls, applications that run in web pages (such as applets, javascript, plug-in applications), and applications that run on the server side (such as cgi scripts, database interfaces, logging applications, dynamic page generators, asp, etc.). Additionally, there are a wide variety of servers and browsers, various versions of each, small but sometimes significant differences between them, variations in connection speeds, rapidly changing technologies, and multiple standards and protocols. The end result is that testing for web sites can become a major ongoing effort. Other considerations might include:  • What are the expected loads on the server (e.g., number of hits per unit time?), and what kind of [performance](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-2-353.asp) is required under such loads (such as web server response time, database query response times). What kinds of tools will be needed for performance testing (such as web load testing tools, other tools already in house that can be adapted, web robot downloading tools, etc.)?  • Who is the target audience? What kind of browsers will they be using? What kind of connection speeds will they by using? Are they intra- organization (thus with likely high connection speeds and similar browsers) or Internet-wide (thus with a wide variety of connection speeds and browser types)?  • What kind of performance is expected on the client side (e.g., how fast should pages appear, how fast should animations, applets, etc. load and run)?  • Will down time for server and content maintenance/upgrades be allowed? how much?  • What kinds of security (firewalls, encryptions, passwords, etc.) will be required and what is it expected to do? How can it be tested?  • How reliable are the site's Internet connections required to be? And how does that affect backup system or redundant connection requirements and testing?  • What processes will be required to manage updates to the web site's content, and what are the requirements for maintaining, tracking, and controlling page content, graphics, links, etc.?  • Which HTML specification will be adhered to? How strictly? What variations will be allowed for targeted browsers?  • Will there be any standards or requirements for page appearance and/or graphics throughout a site or parts of a site??  • How will internal and external links be validated and updated? how often?  • Can testing be done on the production system, or will a separate test system be required? How are browser caching, variations in browser option settings, dial-up connection variabilities, and real-world internet 'traffic congestion' problems to be accounted for in testing?  • How extensive or customized are the server logging and reporting requirements; are they considered an integral part of the system and do they require testing?  • How are cgi programs, applets, javascripts, ActiveX components, etc. to be maintained, tracked, controlled, and tested?  Some sources of site security information include the Usenet newsgroup 'comp.security.announce' and links concerning web site security in the 'Other Resources' section.  Some usability guidelines to consider - these are subjective and may or may not apply to a given situation (Note: more information on usability testing issues can be found in articles about web site usability in the 'Other Resources' section):  • Pages should be 3-5 screens max unless content is tightly focused on a single topic. If larger, provide internal links within the page.  • The page layouts and design elements should be consistent throughout a site, so that it's clear to the user that they're still within a site.  • Pages should be as browser-independent as possible, or pages should be provided or generated based on the browser-type.  • All pages should have links external to the page; there should be no dead-end pages.  • The page owner, revision date, and a link to a contact person or organization should be included on each page.  Many new web site test tools have appeared in the recent years and more than 280 of them are listed in the 'Web Test Tools' section. **5.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 affect 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. If the application was well-designed this can simplify test design.   **6.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. It was created by [Kent](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-2-353.asp) Beck who described the approach in his book 'Extreme Programming Explained' (See the Softwareqatest.com Books page.). Testing ('extreme testing') is a core aspect of Extreme Programming. [Programmers](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-2-353.asp) are expected to write unit and functional test code first - before the application is developed. Test code is under source control along with the rest of the code. Customers are expected to be an integral part of the project team and to help develope scenarios for acceptance/black box testing. Acceptance tests are preferably automated, and are modified and rerun for each of the frequent development iterations. QA and test personnel are also required to be an integral part of the project team. Detailed requirements documentation is not used, and frequent re-scheduling, re-estimating, and re-prioritizing is expected. For more info see the XP-related listings in the Softwareqatest.com 'Other Resources' section.   **7.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. It is oriented to 'prevention'. (See the Bookstore section's 'Software QA' category for a list of useful books on Software Quality Assurance.)   **8.What is 'Software Testing'?** Testing involves operation of a system or application under controlled conditions and evaluating the results (eg, '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'. (See the Bookstore section's 'Software Testing' category for a list of useful books on Software Testing.)  • 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 QA processes monitored by project managers. It will depend on what best fits an organization's size and business structure.  **4.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 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 be in (a) requirements management processes, with a goal of clear, complete, testable requirement specifications embodied in requirements or design documentation and (b) design inspections and code inspections.   **5.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.   **6.What is a 'walkthrough'?** A 'walkthrough' is an informal meeting for evaluation or informational purposes. Little or no preparation is usually required.   **7.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 management 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.  **8.What kinds of testing should be considered?** • [Black box](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-3-352.asp) 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.  • 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, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to client/server 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 tools 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](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-3-352.asp) 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.  • security testing - testing how well the system protects against unauthorized internal or external access, willful damage, etc; may require sophisticated testing techniques.  • compatability 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.  • 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.  **1. What are 5 common problems in the software development process?** • Poor requirements - if requirements are unclear, incomplete, too general, or not testable, there will 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 program is any good until the customer complaints or systems crash.  • Futuristic - requests to pile on new features after development is underway; extremely common.  • Miscommunication - if developers don't know what's needed or customer's have erroneous expectations, problems are guaranteed.   **2. 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. Use prototypes to help nail down requirements.  • 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.  • stick to initial requirements as much as possible - be prepared to defend against 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, use rapid prototyping during the design phase so that customers can see what to expect. This will provide them a higher comfort level with their requirements decisions and minimize changes later on.  • communication - require walkthroughs and inspections when appropriate; make extensive use of group communication tools - e-mail, groupware, networked bug-tracking tools and change management tools, intranet capabilities, etc.; insure that documentation is available and up-to-date - preferably electronic, not paper; promote teamwork and cooperation; use prototypes early on so that customers' expectations are clarified.   **3.What is** [**software**](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-4-351.asp) **'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.  **6.What is SEI? CMM? ISO? IEEE? ANSI? Will it help?** • SEI = 'Software Engineering Institute' at Carnegie-Mellon [University](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-4-351.asp" \t "_top)  [http://kona.kontera.com/javascript/lib/imgs/grey_loader.gif](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-4-351.asp" \t "_top)  ; initiated by the U.S. Defense Department to help improve software development processes.  • CMM = 'Capability Maturity Model', developed by the SEI. It's a model of 5 levels of organizational 'maturity' that determine effectiveness in delivering quality software. It is geared to large 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 CMM ratings by undergoing assessments by qualified auditors.   Level 1 - 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 - software project tracking, requirements management, realistic planning, and configuration management processes are in place; successful practices can be repeated.  Level 3 - standard software development and maintenance processes are integrated throughout an organization; a Software Engineering Process Group is is in place to oversee software processes, and training programs are used to ensure understanding and compliance.  Level 4 - metrics are used to track productivity, processes, and products. Project performance is predictable, and quality is consistently high.  Level 5 - the focus is on continuous process improvement. The impact of new processes and technologies can be predicted and effectively implemented when required.  Perspective on CMM ratings: During 1997-2001, 1018 organizations were assessed. Of those, 27% were rated at Level 1, 39% at 2, 23% at 3, 6% at 4, and 5% at 5. (For ratings during the period 1992-96, 62% were at Level 1, 23% at 2, 13% at 3, 2% at 4, and  0.4% at 5.) The median size of organizations was 100 [software engineering](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-4-351.asp)/maintenance personnel; 32% of organizations were U.S. federal contractors or agencies. For those rated at  Level 1, the most problematical key process area was in Software Quality Assurance.  • ISO = 'International Organization for Standardization' - The ISO 9001:2000 standard (which replaces the previous standard of 1994) concerns quality systems that are assessed by outside auditors, and it applies to many kinds of production and manufacturing organizations, not just software. It covers documentation, design, development, production, testing, installation, servicing, and other processes. The full set of standards consists of: (a)Q9001-2000 - Quality Management Systems: Requirements; (b)Q9000-2000 - Quality Management Systems: Fundamentals and Vocabulary; (c)Q9004-2000 - Quality Management Systems: Guidelines for Performance Improvements. To be ISO 9001 certified, a third-party auditor assesses an organization, and certification is typically good for about 3 years, after which a complete reassessment is required. Note that ISO certification does not necessarily indicate quality products - it indicates only that documented processes are followed. Also see http://www.iso.ch/ for the latest information. In the U.S. the standards can be purchased via the ASQ web site at http://e-standards.asq.org/   • IEEE = 'Institute of Electrical and Electronics Engineers' - among other things, creates standards such as 'IEEE Standard for Software Test Documentation' (IEEE/ANSI Standard 829), 'IEEE Standard of [Software Unit Testing](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-4-351.asp) (IEEE/ANSI Standard 1008), 'IEEE Standard for Software Quality Assurance Plans' (IEEE/ANSI Standard 730), and others.   • ANSI = 'American National Standards Institute', the primary industrial standards body in the U.S.; publishes some software-related standards in conjunction with the IEEE and ASQ (American Society for Quality).   • Other software development process assessment methods besides CMM and ISO 9000 include SPICE, Trillium, TickIT. and Bootstrap.   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.   **7.Will automated testing tools make testing easier?** • Possibly. For small projects, the time needed to learn and implement them may not be worth it. For larger projects, or on-going long-term projects they can be valuable.  • A common type of automated tool is the 'record/playback' type. For example, a tester could click through all combinations of [menu](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-4-351.asp) choices, dialog box choices, buttons, etc. in an application GUI and have them 'recorded' and the results logged by a tool. The 'recording' is typically in the form of text based on a scripting language that is interpretable by the testing tool. If new buttons are added, or some underlying code in the application is changed, etc. the application might then be retested by just 'playing back' the 'recorded' actions, and comparing the logging results to check effects of the changes. The problem with such tools is that if there are continual changes to the system being tested, the 'recordings' may have to be changed so much that it becomes very time-consuming to continuously update the scripts. Additionally, interpretation and analysis of results (screens, data, logs, etc.) can be a difficult task. Note that there are record/playback tools for text-based interfaces also, and for all types of platforms.  • 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](http://www.onestoptesting.com/testing-interview-questions/manual-testing/details/manual-testing-interview-questions-4-351.asp) 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. |

**Testing Models**

There are various models which have been presented in the past 20 years in the field of Software Engineering for Development and Testing. Let us discuss and explore into few of the famous models.   
  
The following models are addressed:   
Waterfall Model.  
Spiral Model.   
'V' Model.   
'W' Model, and   
Butterfly Model.

**The Waterfall Model**   
This is one of the first models of software development, presented by B.W.Boehm. The Waterfall model is a step-by-step method of achieving tasks. Using this model, one can get on to the next phase of development activity only after completing the current phase. Also one can go back only to the immediate previous phase.  
In Waterfall Model each phase of the development activity is followed by the Verification and Validation activities. One phase is completed with the testing activities, then the team proceeds to the next phase. At any point of time, we can move only one step to the immediate previous phase. For example, one cannot move from the Testing phase to the Design phase.   
  
**Spiral Model**   
In the Spiral Model, a cyclical and prototyping view of software development is shown. Test are explicitly mentioned (risk analysis, validation of requirements and of the development) and the test phase is divided into stages. The test activities include module, integration and acceptance tests. However, in this model the testing also follows the coding. The exception to this is that the test plan should be constructed after the design of the system. The spiral model also identifies no activities associated with the removal of defects.   
  
**'V' Model**  
  
Many of the process models currently used can be more generally connected by the 'V' model where the 'V' describes the graphical arrangement of the individual phases. The 'V' is also a synonym for Verification and Validation.  
By the ordering of activities in time sequence and with abstraction levels the connection between development and test activities becomes clear. Oppositely laying activities complement one another (i.e.) server as a base for test activities. For example, the system test is carried out on the basis of the results specification phase.   
  
**The 'W' Model**  
From the testing point of view, all of the models are deficient in various ways:  
The Test activities first start after the implementation. The connection between the various test stages and the basis for the test is not clear.  
The tight link between test, debug and change tasks during the test phase is not clear.  
  
**Why 'W' Model?**   
In the models presented above, there usually appears an unattractive task to be carried out after coding. In order to place testing on an equal footing, a second 'V' dedicated to testing is integrated into the model. Both 'V's put together give the 'W' of the 'W-Model'.   
  
**Butterfly Model of Test Development**  
Butterflies are composed of three pieces – two wings and a body. Each part represents a piece of software testing, as described hereafter.  
**Test Analysis**  
The left wing of the butterfly represents test analysis – the investigation, quantization, and/or re-expression of a facet of the software to be tested. Analysis is both the byproduct and foundation of successful test design. In its earliest form, analysis represents the thorough pre-examination of design and test artifacts to ensure the existence of adequate testability, including checking for ambiguities, inconsistencies, and omissions.   
Test analysis must be distinguished from software design analysis. Software design analysis is constituted by efforts to define the problem to be solved, break it down into manageable and cohesive chunks, create software that fulfills the needs of each chunk, and finally integrate the various software components into an overall program that solves the original problem. Test analysis, on the other hand, is concerned with validating the outputs of each software development stage or micro-iteration, as well as verifying compliance of those outputs to the (separately validated) products of previous stages.  
Test analysis mechanisms vary according to the design artifact being examined. For an aerospace software requirement specification, the test engineer would do all of the following, as a minimum:  
Verify that each requirement is tagged in a manner that allows correlation of the tests for that requirement to the requirement itself. (Establish Test Traceability)   
Verify traceability of the software requirements to system requirements.   
Inspect for contradictory requirements.  
Inspect for ambiguous requirements.   
Inspect for missing requirements.   
Check to make sure that each requirement, as well as the specification as a whole, is understandable.  
Identify one or more measurement, demonstration, or analysis method that may be used to verify the requirement’s implementation (during formal testing).   
Create a test “sketch” that includes the tentative approach and indicates the test’s objectives.  
Out of the items listed above, only the last two are specifically aimed at the act of creating test cases. The other items are almost mechanical in nature, where the test design engineer is simply checking the software engineer’s work. But all of the items are germane to test analysis, where any error can manifest itself as a bug in the implemented application.   
Test analysis also serves a valid and valuable purpose within the context of software development. By digesting and restating the contents of a design artifact (whether it be requirements or design), testing analysis offers a second look – from another viewpoint – at the developer’s work. This is particularly true with regard to lower-level design artifacts like detailed design and source code. This kind of feedback has a counterpart in human conversation. To verify one’s understanding of another person’s statements, it is useful to rephrase the statement in question using the phrase “So, what you’re saying is…”. This powerful method of confirming comprehension and eliminating miscommunication is just as important for software development – it helps to weed out misconceptions on the part of both the developer and tester, and in the process identifies potential problems in the software itself.  
It should be clear from the above discussion that the tester’s analysis is both formal and informal. Formal analysis becomes the basis for documentary artifacts of the test side of the V. Informal analysis is used for immediate feedback to the designer in order to both verify that the artifact captures the intent of the designer and give the tester a starting point for understanding the software to be tested.   
In the bulleted list shown above, the first two analyses are formal in nature (for an aerospace application). The verification of system requirement tags is a necessary step in the creation of a test traceability matrix. The software to system requirements traceability matrix similarly depends on the second analysis.  
The three inspection analyses listed are more informal, aimed at ensuring that the specification being examined is of sufficient quality to drive the development of a quality implementation. The difference is in how the analytical outputs are used, not in the level of effort or attention that go into the analysis.

**Test Design**  
Thus far, the tester has produced a lot of analytical output, some semi-formalized documentary artifacts, and several tentative approaches to testing the software. At this point, the tester is ready for the next step: test design.  
The right wing of the butterfly represents the act of designing and implementing the test cases needed to verify the design artifact as replicated in the implementation. Like test analysis, it is a relatively large piece of work. Unlike test analysis, however, the focus of test design is not to assimilate information created by others, but rather to implement procedures, techniques, and data sets that achieve the test’s objective(s).   
The outputs of the test analysis phase are the foundation for test design. Each requirement or design construct has had at least one technique (a measurement, demonstration, or analysis) identified during test analysis that will validate or verify that requirement. The tester must now put on his or her development hat and implement the intended technique.   
Software test design, as a discipline, is an exercise in the prevention, detection, and elimination of bugs in software. Preventing bugs is the primary goal of software testing [BEIZ90]. Diligent and competent test design prevents bugs from ever reaching the implementation stage. Test design, with its attendant test analysis foundation, is therefore the premiere weapon in the arsenal of developers and testers for limiting the cost associated with finding and fixing bugs.   
Before moving further ahead, it is necessary to comment on the continued analytical work performed during test design. As previously noted, tentative approaches are mapped out in the test analysis phase. During the test design phase of test development, those tentatively selected techniques and approaches must be evaluated more fully, until it is “proven” that the test’s objectives are met by the selected technique. If all tentatively selected approaches fail to satisfy the test’s objectives, then the tester must put his test analysis hat back on and start looking for more alternatives.   
  
**Test Execution**  
In the butterfly model of software test development, test execution is a separate piece of the overall approach. In fact, it is the smallest piece – the slender insect’s body – but it also provides the muscle that makes the wings work. It is important to note, however, that test execution (as defined for this model) includes only the formal running of the designed tests. Informal test execution is a normal part of test design, and in fact is also a normal part of software design and development.  
Formal test execution marks the moment in the software development process where the developer and the tester join forces. In a way, formal execution is the moment when the developer gets to take credit for the tester’s work – by demonstrating that the software works as advertised. The tester, on the other hand, should already have proactively identified bugs (in both the software and the tests) and helped to eliminate them – well before the commencement of formal test execution!   
Formal test execution should (almost) never reveal bugs. I hope this plain statement raises some eyebrows – although it is very much true. The only reasonable cause of unexpected failure in a formal test execution is hardware failure. The software, along with the test itself, should have been through the wringer enough to be bone-dry. Note, however, that unexpected failure is singled out in the above paragraph. That implies that some software tests will have expected failures, doesn’t it? Yes, it surely does!  
The reasons behind expected failure vary, but allow me to relate a case in point:   
In the commercial jet engine control business, systems engineers prepare a wide variety of tests against the system (being the FADEC – or Full Authority Digital Engine Control) requirements. One such commonly employed test is the “flight envelope” test. The flight envelope test essentially begins with the simulated engine either off or at idle with the real controller (both hardware and software) commanding the situation. Then the engine is spooled up and taken for a simulated ride throughout its defined operational domain – varying altitude, speed, thrust, temperature, etc. in accordance with real world recorded profiles. The expected results for this test are produced by running a simulation (created and maintained independently from the application software itself) with the same input data sets.   
Minor failures in the formal execution of this test are fairly common. Some are hard failures – repeatable on every single run of the test. Others are soft – only intermittently reaching out to bite the tester. Each and every failure is investigated, naturally – and the vast majority of flight envelope failures are caused by test stand problems. These can include issues like a voltage source being one twentieth of a volt low, or slight timing mismatches caused by the less exact timekeeping of the test stand workstation as compared to the FADEC itself.   
Some flight envelope failures are attributed to the model used to provide expected results. In such cases, hours and days of gut-wrenching analytical work go into identifying the miniscule difference between the model and the actual software.  
A handful of flight envelope test failures are caused by the test parameters themselves. Tolerances may be set at unrealistically tight levels, for example. Or slight operating mode mismatches between the air speed and engine fan speed may cause a fault to be intermittently annunciated.   
In very few cases have I seen the software being tested lay at the root of the failure. (I did witness the bugs being fixed, by the way!)   
The point is this – complex and complicated tests can fail due to a variety of reasons, from hardware failure, through test stand problems, to application error. Intermittent failures may even jump into the formal run, just to make life interesting.  
But the test engineer understands the complexity of the test being run, and anticipates potential issues that may cause failures. In fact, the test is expected to fail once in a while. If it doesn’t, then it isn’t doing its job – which is to exercise the control software throughout its valid operational envelope. As in all applications, the FADEC’s boundaries of valid operation are dark corners in which bugs (or at least potential bugs) congregate. It was mentioned during our initial discussion of the V development model that the model is sufficient, from a software development point of view, to express the lineage of test artifacts. This is because testing, again from the development viewpoint, is composed of only the body of the butterfly – formal test execution. We testers, having learned the hard way, know better.