1: How to test website functionality? Design various test cases for website functionality and web applications.

TEST STRATEGIES FOR WEBAPPS:

The strategy for WebApp testing adopts the basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems. The following steps summarize the approach:

1. The content model for the WebApp is reviewed to uncover errors.
2. The interface model is reviewed to ensure that all use cases can be accommodated.
3. The design model for the WebApp is reviewed to uncover navigation errors.
4. user interface is tested to uncover errors in presentation and/or navigation mechanics.
5. Each functional component is unit tested.
6. Navigation throughout the architecture is tested.
7. The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
9. Performance tests are conducted.
10. The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated for errors.

TEST STRATEGIES FOR MOBILEAPPS:

The strategy for testing mobile applications adopts the basic principles for all software testing. However, the unique nature of MobilePASS demands the consideration of a number of specialized testing approaches:

user-experience testing: Users are involved early in the development process to ensure that the Mobile App lives up to the usability and accessibility expectations of the stakeholders on all supported devices.

Device compatibility testing. Testers verify that the Mobile App works correctly on all required hardware and software combinations.

Performance testing. Testers check nonfunctional requirements unique to mobile devices (e.g., download times, processor speed, storage capacity, power availability).

Connectivity testing. Testers ensure that the Mobile App can access any needed networks or Web services and can tolerate weak or interrupted network access.

Security testing. Testers ensure that the Mobile App does not compromise the privacy or security requirements of its users.

Testing-in-the-wild. The app is tested under realistic conditions on actual user devices in a variety of networking environments around the globe.

Certification testing. Testers ensure that the Mobile App meets the standards established by the app stores that will distribute it.

SOFTWARE TESTING F UNDAMENTALS:

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. Therefore, you should design and implement a computer-based system or a product with "testability" in mind. At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

Testability. James Bach 1 provides the following definition for testability: "Software testability is simply how easily [a computer program] can be tested." The following characteristics lead to testable software.

Operability. "The better it works, the more efficiently it can be tested." If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.

Observability. "What you see is what you test." Inputs provided as part of testing produce distinct outputs. System states and variables are visible or quarriable during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.

Controllability. "The better we can control the software, the more the testing can be automated and optimized." All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured. All code is executable through some combination of input. Software and hardware states and variables can be controlled directly by the test engineer. Tests can be conveniently specified, automated, and reproduced.

Decomposability. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting." The software system is built from independent modules that can be tested independently.

Simplicity. "The less there is to test, the more quickly we can test it." The program should exhibit functional simplicity (e.g., the feature set is the minimum necessary to meet requirements); structural simplicity (e.g., architecture is modularized to limit the propagation of faults), and code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability. "The fewer the changes, the fewer the disruptions to testing." Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failures

Understandability. "The more information we have, the smarter we will test." The architectural design and the dependencies between internal, external, and shared components are well understood. Technical documentation is instantly accessible, well organized, specifi c and detailed, and accurate. Changes to the design are communicated to testers.

Test Characteristics. And what about the tests themselves? Kaner, Falk, and Nguyen [Kan93] suggest the following attributes of a "good" test:

A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.

A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).

A good test should be "best of breed" [Kan93]. In a group of tests that have a similar intent, time and resource limitations may dictate the execution of only those tests that has the highest likelihood of uncovering a whole class of errors.

A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

Exhaustive Testing:

Consider a 100-line program in the language C. After some basic data declaration, the program contains two nested loops that execute from 1 to 20 times each, depending on conditions specified at input. Inside the interior loop, four if-then-else constructs are required. There are approximately 10 power 14 possible paths that may be executed in this program! To put this number in perspective, we assume that a magic test processor ("magic" because no such processor exists) has been developed for exhaustive testing. The processor can develop a test case, execute it, and evaluate the results in one millisecond. Working 24 hours a day, 365 days a year, the processor would work for 3170 years to test the program. This would, undeniably, cause havoc in most development schedules. Therefore, it is reasonable to assert that exhaustive testing is impossible for large software systems.

Will exhaustive testing guarantee that the program is 100 percent correct? Illustrate with suitable examples.

No, even exhaustive testing will not guarantee that the program is 100 percent correct. There are too many variables to consider.

Consider this...

Installation testing - did the program install according to the instructions?

Integration testing - did the program work with all of the other programs on the system without interference, and did the installed modules of the program integrate and work with other installed modules? Function testing - did each of the program functions work properly? Unit testing - did the unit work as a standalone as designed, and did the unit work when placed in the overall process? User Acceptance Testing - did the program fulfil all of the user requirements and work per the user design? Performance testing - did the

program perform to a level that was satisfactory and could it carry the volume load placed upon it? While these are just the basic tests for an exhaustive testing scenario, you could keep testing beyond these tests using destructive methods, white box internal program testing, establish program exercises using automated scripts, etc. The bottom line is... testing has to stop at some point in time. Either the time runs out that was allotted for testing, or you gain a confidence level that the program is going to work. (Of course, the more you test, the higher your confidence level). I don't know anyone that would give a 100% confidence level that the program is 100% correct, (to do so is to invite people to prove you wrong and they will come back with all kinds of bugs you never even considered). However, you may be 95% confident that you found most all of the major bugs. Based upon this level of confidence, you would then place the program into production use - always expecting some unknown bug to be found.

What are the signs that a software project is in jeopardy? Suggest a five- part commonsense approach to software project.

In order to manage a successful software project, you have to understand what can go wrong so that problems can be avoided. In an excellent paper on software projects, John Reel [Ree99] defines signs that indicate that an information systems project is in jeopardy.

In some cases,

software people don’t understand their customer’s needs. This leads to a project with a poorly defined scope.

In other projects, changes are managed poorly.

Sometimes, the chosen technology changes or business needs shift and management sponsorship is lost. Management can set unrealistic deadlines or end users can be resistant to the new system.

There are cases in which the project team simply does not have the requisite skills. And finally, there are developers who never seem to learn from their mistakes.

Five-part commonsense approach to software projects;

1.Start on the right foot. This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objectives and expectations for everyone who will be involved in the project. It is reinforced by building the right team (Section 31.2.3) and giving the team the autonomy, authority, and technology needed to do the job.

2. Maintain momentum. Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team’s way.

3. Track progress. For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using technical reviews) as part of a quality assurance activity. In addition, software process and project measures (Chapter 32) can be collected and used to assess progress against averages developed for the software development organization.

4. Make smart decisions. In essence, the decisions of the project manager and the software team should be to "keep it simple." Whenever possible, decide to use commercial off-the-shelf software or existing software components or patterns, decide to avoid custom interfaces when standard approaches are available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks (you'll need every minute).

5. Conduct a postmortem analysis. Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyse software project metrics, get feedback from team members and customers, and record findings in written form.

Briefly explain any five-software metrics used for software measurement.

Source code metrics:

These are measurements of the source code that make up all your software. Source code is the fundamental building block of which your software is made, so measuring it is key to making sure your code is high-calibre. (Not to mention there is almost always room for improvement.) Look closely enough at even your best source code, and you might spot a few areas that you can optimize for even better performance.

When measuring source code quality make sure you're looking at the number of lines of code you have, which will ensure that you have the appropriate amount of code and it's no more complex than it needs to be. Another thing to track is how compliant each line of code is with the programming languages' standard usage rules. Equally important is to track the percentage of comments within the code, which will tell you how much maintenance the program will require. The less comments, the more problems when you decide to change or upgrade. Other things to include in your measurements is code duplications and unit test coverage, which will tell you how smoothly your product will run (and at when are you likely to encounter issues).

Development metrics:

These metrics measure the custom software development process itself. Gather development metrics to look for ways to make your operations more efficient and reduce incidents of software errors.

Measuring number of defects within the code and time to fix them tells you a lot about the development process itself. Start by tallying up the number of defects that appear in the code and note the time it takes to fix them. If any defects have to be fixed multiple time, then there might be a misunderstanding of requirements or a skills gap – which is important to address as soon as possible.

Project Metrics:

Unlike software process metrics that are used for strategic purposes, software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work fl ow and technical activities. The fi rst application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

Size-Oriented Metrics:

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure 32.2 , can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project.

Function-Oriented Metrics:

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used functionoriented metric is the function point (FP). Computation of the function point is based on characteristics of the software's information domain and complexity.

Reconciling LOC and FP Metrics:

The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures. The following table4 [QSM02] provides rough estimates of the average number of lines of code required to build one function point in various programming languages:

Object-Oriented Metrics:

Conventional software project metrics (LOC or FP) can be used to estimate object-oriented software projects. However, these metrics do not provide enough granularity for the schedule and effort adjustments that are required as you iterate through an evolutionary or incremental process. Lorenz and Kidd [Lor94] suggest the following set of metrics for OO projects:

Use Case-Oriented Metrics:

Use cases are used widely as a method for describing customer-level or business domain requirements that imply software features and functions. It would seem reasonable to use the use case as a normalization measure

similar to LOC or FP. Like FP, the use case is defined early in the software process, allowing it to be used for estimation before significant modelling and construction activities are initiated. Use cases describe (indirectly, at least) user-visible functions and features that are basic requirements for a system. The use case is independent of programming language. In addition, the number of use cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

Testing metrics:

These metrics help you evaluate how functional your product is. (And we're assuming you want it very functional for your customers.) There are two major testing metrics. One of them is "test coverage" that collects data about which parts of the software program are executed when it runs a test. The second part is a test of the testing itself. It's called "defect removal efficiency," and it checks your success rate for spotting and removing defects. The more you measure, the more you know about your software product, the more likely you are able to improve it. Automating the measurement process is the best way to measure software quality – it's not the easiest thing, or the cheapest, but it'll save you tons of cost down the line.

Effectiveness of soft-ware engineering Team:

To effectively manage a software engineering team, it is crucial to understand the roles of stakeholders, the skills of team leaders, the structure of the team, the dynamics that foster cohesiveness, and the coordination and communication issues that arise. Here's a detailed explanation based on the provided text:

The Stakeholders

The software process (and every software project) is populated by stakeholders who can be categorized into one of five constituencies:

1. Senior Managers: Define business issues that influence the project.

2. Project (Technical) Managers: Plan, motivate, organize, and control practitioners who do the software work.

3. Practitioners: Deliver the technical skills necessary to engineer a product or application.

4. Customers: Specify the requirements for the software and other stakeholders who have a peripheral interest in the outcome.

5. End Users: Interact with the software once it is released for production use.

Team Leaders

Effective team leaders are essential for project success. They should have the right mix of people skills, and Weinberg's MOI model suggests that successful project leaders possess:

1. Motivation: Encourage technical people to produce to their best ability.

2. Organization: Mold existing processes or invent new ones to translate the initial concept into a final product.

3. Ideas or Innovation: Encourage creativity even within established bounds.

Effective project managers also have key traits:

1. Problem Solving: Diagnose technical and organizational issues and develop solutions.

2. Managerial Identity: Take charge of the project confidently.

3. Achievement: Reward initiative and accomplishment to optimize productivity.

4. Influence and Team Building: Understand and react to the needs of the team, especially in high-stress situations.

The Software Team

The structure of the software team depends on various factors, including the management style of the organization, the number of people, their skill levels, and the problem difficulty. Four organizational paradigms for software engineering teams are:

1. Closed Paradigm: Traditional hierarchy of authority, suitable for projects similar to past efforts but less innovative.

2. Random Paradigm: Loosely structured, depends on individual initiative, excels in innovation but may struggle with orderly performance.

3. Open Paradigm: Combines control with innovation, suitable for complex problems, relies on heavy communication and consensus-based decision-making.

4. Synchronous Paradigm: Relies on natural compartmentalization, with little active communication among team members.

A high-performance team must have trust, appropriate skill distribution, and may need to exclude mavericks to maintain cohesiveness. DeMarco and Lister describe a "jelled" team as one that is strongly knit, with members significantly more productive and motivated than average.

Agile Teams

Agile teams emphasize customer satisfaction, early incremental delivery of software, small highly motivated project teams, informal methods, minimal software engineering work products, and overall development simplicity. Key aspects include:

1. Self-Organization: Teams have significant autonomy to make project management and technical decisions.

2. Collaboration: Continual self-organization and collaboration move the team toward a completed software increment.

3. Daily Meetings: Used to coordinate and synchronize work for the day.

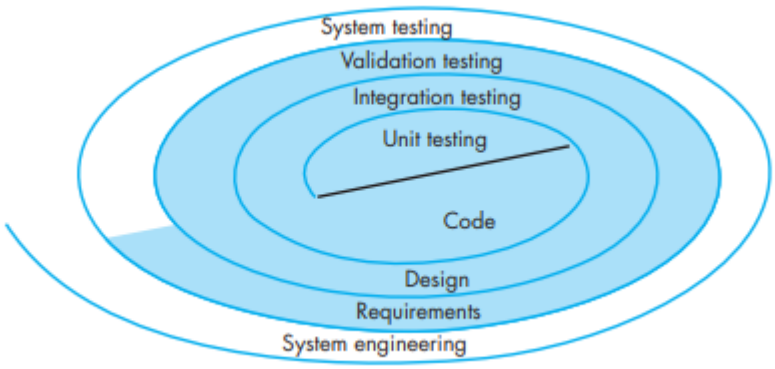
Coordination and Communication Issues

Effective coordination and communication are crucial due to the scale, uncertainty, and interoperability of modern software. Mechanisms for formal and informal communication must be established:

1. Formal Communication: Writing, structured meetings, and other non-interactive channels. **2. Informal Communication:** Personal interactions, ad hoc idea sharing, and daily problem-solving interactions.

By understanding and implementing these principles, a software engineering team can be organized and managed effectively, leading to successful project outcomes.

22.1.3 Software Testing Strategy—The Big Picture:



The software process may be viewed as the spiral illustrated in Figure 22.1 . Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behaviour, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, you come to design and finally to coding. To develop computer software, you spiral inward along streamlines that decrease the level of abstraction on each turn.

A strategy for software testing may also be viewed in the context of the spiral (Figure 22.1). Unit testing begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code. Testing progresses by moving outward along the spiral to integration testing, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, you encounter validation testing, where requirements established as part of requirements modelling are validated against the software that has been constructed. Finally, you arrive at system testing, where the software and other system elements are tested as a whole. To test computer software, you spiral out along streamlines that broaden the scope of testing with each turn. Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are shown in Figure 22.2 . Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name unit testing. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection. Next, components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and program construction. Test-case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of high-order tests is conducted. Validation criteria (established during requirements analysis) must be evaluated. Validation testing provides final assurance that software meets all functional, behavioural, and performance requirements.