8 : Introduction to Testing in SCRUM

4406 – Agile Software Developmet

Level II - Semester 4

Overview

- Agile software development methods are designed to several advantages such as reduce time taken for development, to improve software quality and increasing the satisfaction of customer needs.

- In this slide set, the agile development process is described from the viewpoint of testing and QA.

- It shows how agile testing works and explain where traditional testing techniques are still necessary within the agile environment.

Intended Learning Outcomes

- At the end of this lesson, you will be able to;
  - Understand the differences of testing in between waterfall method and Agile methodology.
  - Identify Agile quality assurance tools.
  - Analyze various testing approaches.
  - Comparing the testing methods

List of sub topics

8.1.1. Testing in waterfall method vs Agile Testing

8.1.2. Agile Quality Assurance vs Traditional Quality Assurance
- 8.1.2.1 Traditional Quality Assurance Tools
- 8.1.2.2 Agile Quality Assurance Tools
- 8.1.2.3 Test Planning in Scrum

8.1.3. Unit testing
- 8.1.3.1 The Test First approach
- 8.1.3.2 Unit Testing Frameworks
- 8.1.3.3 Unit Test Management

List of sub topics

8.1.4. Integration Testing
- 8.1.4.1 Designing Integration Test Cases
- 8.1.4.2 Dependencies and Interfaces
- 8.1.4.3 Integration Levels
- 8.1.4.4 Continuous Integration
- 8.1.4.5 Implementing CI
- 8.1.4.6 Integration Test Management

8.1.5. System Testing and Testing Nonstop
- 8.1.5.1 Manual System Testing
- 8.1.5.2 Automated System Testing
- 8.1.5.3 Using Test First for System Testing
- 8.1.5.4 Non-functional Testing
- 8.1.5.5 Automated Acceptance Testing
- 8.1.5.6 System Test Management

1.1 Testing in waterfall method vs Agile Testing

- Traditional Test Management
  - In traditional process models, the project is divided into distinct phases which has the objective of achieving predefined milestones.
  - If we consider the waterfall model, testing is a separate phase whereas in Agile testing, testing is performed alongside the development.

1.1 Testing in waterfall method vs Agile Testing

- Agile Testing
  - Agile testing follows the principles in the Agile Manifesto and applies the principles of agile methodology to software testing.
  - In Scrum, quality isn't something a testing team "tests in" at the end; it is something that a cross-functional Scrum team owns and continuously builds in and verifies every sprint.
  - As a result, the need for any significant late testing to track on quality is substantially reduced.
  - Therefore, an Agile tester, performs continuous, parallel testing at each sprint.

1.1 Testing in waterfall method vs Agile Testing

- Agile Testing
  - Critical factors in Agile Testing:
  - Test Automation

    When the test cases are automated sufficiently, feedback can be obtained without delay. This is a very important prerequisite for a reliable application and a clean code.

  - Exploratory Testing

    Since automation of every case immediately is not possible, manual testing is needed. Testers can use “Exploratory Testing” technique which is conducted without a lot of preparation. With this, short term testing n new features can be achieved.

1.1 Testing in waterfall method vs Agile Testing

- Agile Testing
  - Critical factors in Agile Testing:
    - Test Expertise within the team

      Testing tasks are handle in the same way as all other activities with in the sprint. Each developer (team member) should contribute in testing. There are some situations where the external testers are being incorporated into the teams. Any way these external testers must learn to test as a part of the agile team; independent testing is not allowed.

1.1 Testing in waterfall method vs Agile Testing

- Agile Testing
  - Critical factors in Agile Testing:
    - Multiple Teams

      If multiple Scrum teams are working on a single project, all the features developing should continue to work together. The system-wide testing should not be neglected. Test Manager and the full time testers assigned to the teams should exchange their views similar to Scrum of Scrums.

1.2 Agile Quality Assurance vs Traditional Quality Assurance

- Traditional Quality Assurance Tools
  - The quality assurance measures that we use in the traditional software development can be either constructive or analytical.
  - Example for a constructive measure is “ A documents is created in a way accordance with a template”.
  - Testing, checking, reviewing are examples of analytical measures.
  - A QA plan is drafted by the QA manager in traditionally managing projects.
  - There is a separate QA group to focus exclusively on QA tasks and identify defects prior to delivery.
  - This is an important advantage of traditional project organization and management.

1.2 Agile Quality Assurance vs Traditional Quality Assurance

- Agile Quality Assurance Tools
    - The entire team is responsible for the product and its quality.
    - Therefore, we have no dedicated QA group which has the responsibility of QA and testing tasks.
    - Each and every member is performs his/her own special skill; but not bound to a particular job.
    - Quality assurance in an agile team is based on the principle of “Inspection and Adaptation”
    - Inspection
        - Scrum artifacts needs to be inspected frequently so that the team can progress towards sprint goals.

1.2 Agile Quality Assurance vs Traditional Quality Assurance

- Agile Quality Assurance Tools Cont.
  - Adaptation
    - An adjustment must be made as soon as possible to minimize further deviation for the expected limits.
  - Not like in traditionally managed projects, we do not decide when, how the checks are performed in advance.
  - Instead, the decisions are taking at the sprint's meetings such as sprint plan, daily scrum, review and retrospective.
  - Even though there is no specific person assigned for testing in Agile, the QA issues monitored thoroughly.
  - In daily scrum, the QA issues are discussed since Scrum ensures the quality by identifying deficits at early stage.

1.2 Agile Quality Assurance vs Traditional Quality Assurance

- Agile Quality Assurance Tools Cont.

  - The advantages of Agile QA measures are as follows:

1. Every development artifact is tested
   - Checking the program codes, diagrams, design documents through task board and backlog
2. Corrective measure takes place as early as possible
   - In a scenario where the automated test case fails, if possible the responsible programmer rectify the code immediately. Otherwise a solution is given in the current sprint. If the problem exists, creating a solution will be added to the product backlog.

1.2 Agile Quality Assurance vs Traditional Quality Assurance

- Agile Quality Assurance Tools Cont.

3. Constructive QA tools
   - Clean Code, test automation, test first are some techniques used with in the development process. Members of the team practise own standards to increase the quality of the product.
4. Process improvements are made from the bottom up
   - According to the needs of the team, immediate actions are taken.

1.2 Agile Quality Assurance vs Traditional Quality Assurance

- Test planning in Scrum
  - Similar to the other activities, testing activities are also planned and controlled within a Sprint.
    (Product backlog → Sprint backlog → Task board)
  - We need to pay attention to the below facts when Scrum-based tests are planned:
    - Definition of Read (DOR):

DOR is a checklist used to assure the quality of a User Story. From a tester's view, if a given User Story is not clear enough to draft a test case (if the tester cannot figure out when to pass or fail a test case) we can say the story is not sufficiently refined.

1.2 Agile Quality Assurance vs Traditional Quality Assurance

- Test planning in Scrum Cont.

    - Definition if Done (DoD):

DOD is another checklist used by the team to decide whether a given feature is ready to be included in the Sprint Review. DOD has a test types, test coverage and criteria to assure the quality of the product and user satisfaction.

1.3 Unit Testing

- Unit Testing
    - All the tests that a developer applies on his/her code is known as “developer testng” or “unit testing”.
    - In traditional approaches, there are separate integration testing teams / system testing teams. But in Agile all the tasks are handled by the team it self.
    - Unit testing is performed in individual components so when a defect is identified, it is easy to rectify.
    - For object -oriented languages, classes, and associated methods can be taken as sole elements and for non-object oriented languages, we can use functions, modules as individual components.

1.3 Unit Testing

- Test First Approach
    - In traditional software development, we follow “Program first, then test” method.
    - But Test First is an effective method that is used in the Agile development.
    - Test First means considering which tests will be necessary to show that the software actually fulfills any new specifications before any changes in the code itself are performed.
    - Such tests are designed and automated before they are run.
    - Because the new (or altered) product code doesn’t yet exist, the tests will of course fail.

1.3 Unit Testing

- Test First Approach Cont.
    - Developer starts to write the product code only after completing the automated test.
    - When all the tests are passed, coding is completed.
    - If any test is failed, developer needs to work on the product code until all the test cases are passed.
    - In this approach, tests drives the developer. Therefore we call it “Test-driven development”.
    - The developer applies the cycle “write test → run test → change code”.
    - If a team is using strict Test First methodology, it will dramatically increase the effectiveness of its unit testing as a quality management tool.

1.3 Unit Testing

- Test First Approach Cont.
    - Scrum does not demand Test first approach and work with traditional unit testing.
    - However, when a scrum team uses test first techniques they can get high benefits due to the acceleration of the feedback loop it provides.
    - Every developer gets extra feedback loop for every programming task.
    - Normal unit tests that are written after coding can only check whether the new changes has damaged existing functionality.
    - But, in Test First method, the programmer can check the new functionality.

1.3 Unit Testing

- Test First Approach Cont.
    - Implementing tests for the objects that does not exist is not easy.
    - When writing Test First code, it forces the developer to think in a more abstract way than conventional unit testing,
    - In a practical scenario, when the team is switching to Test First, it is easier if the team uses pair programming.
    - Having code reviews on the team level is also important to improve the Test first approach.

1.3 Unit Testing

- Unit Testing Frameworks
  - Unit test automation is not a hard task.
  - Let's look at an example,

- Suppose there is a class 'abc' that needs to be tested. The test class is 'abcTest'. Each test case is implemented by one unique method.
- When name the test method, use meaningful names which reflects the content.
- Each test case is written to check only one aspect of the test object.
- There are 4 sections in each test case. 'setup', 'test procedure', 'check', and 'teardown'.
- In 'check' section, the actual output is compared with the expected output.
- If actual and expected output are the same, the test is 'passed' otherwise 'failed'.
- These methods can run individually or as a batch using the 'run' method given in framework.

1.3 Unit Testing

- Unit Testing Frameworks Cont.
  - If the developer wants to include additional functionality, it can be done using pre-coded unit test frameworks.
  - Using frameworks can simplify the structure of the test code.
  - These frameworks are freely available online for majority of popular languages.
  - JUnit for Java, CppUnit for C++, NUnit for .Net are some examples created based on the SUnit framework.
  - Practically, the objects which the unit tests are run on will be components of larger systems. There can be dependencies for other components.
  - Therefore, to test an object, all dependencies must be installed in the environment.
  - But some dependent components may not be available at the testing time since they are to be implemented in coming Sprints.

1.3 Unit Testing

- Unit Testing Frameworks Cont.
  - As a solution for the unavailability of some components at the time of testing, we can replace dependencies with placeholders, called ‘test doubles’.
  - Following are different types of placeholders

    Stub: Replaces a depended-on component with a component that has an identical interface that produces specific reactions (returns certain values according to a predefined pattern).

    Spy: Astub with additional technique of recording calls and data handed over by the test object. Recorded data can use to debugging.

1.3 Unit Testing

- Unit Testing Frameworks Cont.

  Mock: A stub that analyzes call and data received from the test object.Return results to the object.

  Fake: Simplified implementation to replace a depended-on component that is required for testing. It does not influence on the test results.

  Dummy: An empty object or null pointer which replace a data object. A dummy is not interpreted and therefore forms no part of the test data

1.3 Unit Testing

- Unit Testing Frameworks Cont.
    - In Agile development, these placeholders play an important role compared to traditionally managing projects.
    - When the components are developed incrementally, there can be situations where some components not yet exist.
    - To prevent the block of receiving feedback that can be resulted from missing test components, it is essential to have place holders.
    - At each time the code modifies, Test First will automatically run existing unit tests. Therefore, it is important to keep the run time of tests minimum.
    - Creating placeholders involves an effort and this needs to be considered when planning a Sprint.

1.3 Unit Testing

- Unit Test Management
  - The following aspects need to be agreed in test management:
    Unit test framework: It is the responsibility of the Srcum master to make sure that the whole team write similarly structured tests.Tests need to be stored centrally with an agreed structure. It is advised to keep the test code separately but parallely to the program code. Team has to agree on a naming convention for test classes and test methods.

    Coverage measurement: The Scrum master needs to ensure the test coverage is measured in a reliable way. It can be achieved by integrating a coverage measurement tool. measurements such as class coverage, method coverage within a class, line coverage has to be taken.

1.3 Unit Testing

- Unit Test Management Cont.

  - The following aspects need to be agreed in test management:
    Static code analysis: In addition to the static code analysis, dynamic unit tests need to be implemented. Developers can use different tools based on the programming language they use.

    Test code reviews: Regular test code reviews has to carried out to check the test design, boundary values, effectiveness etc. If this process found any deficiencies, the team should start appropriate training. A review also helps in identifying which test cases still need to be written to increase coverage.

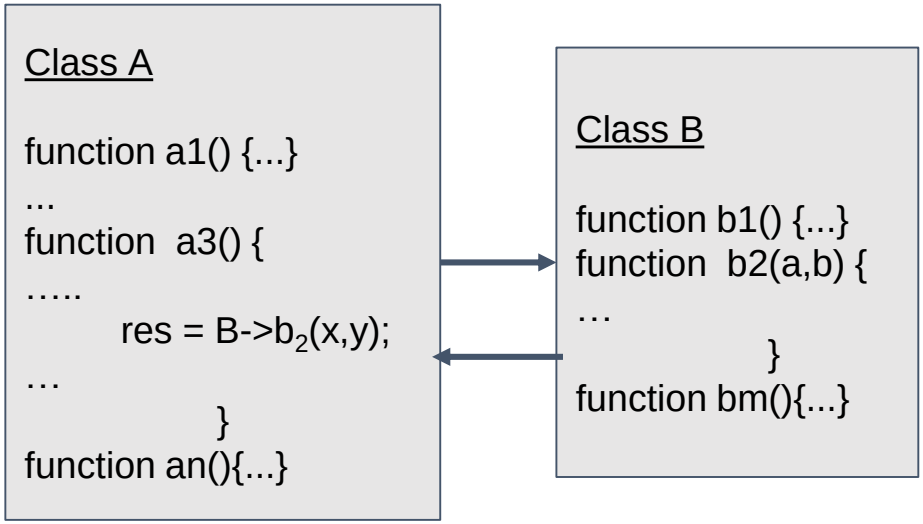
1.3 Unit Testing

- Unit Test Management Cont.

  - After several Sprints of successful quality automated tests, the team can extend unit tests to check non-functionalities such as performance, security etc.
  - Anyway these tests does not guarantee the non-functional aspects of the final product. But it does help to identify weaknesses at the early stage.

1.4 Integration Testing

- Integration Testing
  - Ensures that all the independently designed components of a system perform together as expected.
  - Integration tests identify potential defects in interactions between individual components and user interfaces.
  - Example: An integration test can be performed to test if the classes A and B are working well together. The interaction is between the methods b2 and a3, the test should ensure if b2 is called correctly and if it returns the correct values for a3.



1.4 Integration Testing

- Designing Integration Test Cases
  - Given below are the steps to follow to derive integration test cases:

1. Analyse interaction

   Identify and list inter-component interactions using the architecture diagram.

2. Equivalence partitioning

   Identify the parameters or messages (inputs) that are exchanged in the interactions identified in step 1. Then, partition these parameters into sets of equal size and determine which of them cause the behaviour of the affected component.

1.4 Integration Testing

- Designing Integration Test Cases Contd.

3. Establish test case inputs
   Using all the equivalences determined in step 2 identify the calls or calling sequences in a component's API that are required to call the other component.

3. Define an expected behaviour
   Define the expected behaviours of both observed components for each test case.

3. Test robustness of asynchronous communication
   Asynchronous calls from one component to another require additional testing to check and fix timing, throughput, capacity and performance defects.
   This step is only necessary when the components run parallelly and are connected asynchronously.

1.4 Integration Testing

- Dependencies and Interfaces
  - Explicit (Direct) Dependency
    - One component calls another directly via its API and this type of dependency can clearly be observed from the code.
    - Identified by the compiler or static code analysis.
    - Integration tests for this type of components require to cover all relevant variations on their method calls or the data transfer between the components in case of asynchronous communication.

1.4 Integration Testing

- Dependencies and Interfaces Contd.
  - Implicit (Indirect) Dependency
    - Multiple components share a single resource such as a global variable, a file or a database.
    - These dependencies are caused by the changes in behaviour of one component due to content of a shared resource altering as a result of a write by another component.
    - The code itself will only reveal that each component accesses the same resource but not how or when.
    - Integration tests should either cover all read/write sequences that affect the shared resource and/or the relevant contents of the resource.

1.4 Integration Testing

- Integration Levels

  - Components to be integrated may be of different levels of granularity depending on the level of abstraction on which the integration takes place.

  1. Class Integration:
     - This is the lowest level of integration.
     - In object-oriented programming, smaller units are encapsulated within a class and are tested as part of a unit test.
     - Types of class integration are as follows:

1.4 Integration Testing

- Integration Levels Contd

1. Class Integration Contd:

    1. Vertical Integration

    - Classes are integrated in order of their inheritance hierarchies.
    - Example: If class B inherits its attributes from class A, then if class A is altered the calls from B would fail or behave differently. Therefore, an integration test of the interaction between A and B has to be repeated after each change to A.
    - In the case that methods in B can overwrite the methods in A, then the changes in B should be followed by tests too.
    - If there is only one runtime object that runs the inherited code in its method then the test will follow the inheritance hierarchy and can be viewed as a unit test.

1.4 Integration Testing

- Integration Levels Contd

1. Class Integration Contd:

2. Horizontal Integration

   - Describes an interaction between two classes with no inherited attributes and with at least one object of each existing at runtime.
   - One class uses the other.
   - The interaction between the two classes require to be integration tested.

1.4 Integration Testing

- Integration Levels Contd

1. Class Integration Contd:

3. Compound Classes and Objects

- One class contains the other in the form of a data structure.
- Class A accesses B by using the B's methods and/ or variables.
- Class B can be seen as part of A and even though unit testing A would test B indirectly, B's behaviour cannot be observed.
- Dependency injections can be used in testing where the dependent object is not embedded as a variable of A but is instead given as a parameter, allowing to observe B's behaviour from outside.

1.4 Integration Testing

- Integration Levels Contd

2. Subsystem Integration:
   - Once paired classes have been tested and works as expected, they can be compounded together to create a new component.
   - Repeated integration of such classes creates a cluster of classes referred to as a package or subsystem.
   - The interface used by the cluster varies based on the programming language whereas some languages allow specifying an interface for the package/ subsystem. While other use the some of the APIs within the cluster.

1.4 Integration Testing

- Integration Levels Contd

3. System Integration:

    - Once all the subsystems have been tested, they’re integrated to form a complete system.
    - Includes testing if the system uses the intended interfaces for communicating, often as part of system testing.
    - Testing at this level ensures that the customer receives a defect-free software and eliminates the costs that could be incurred should the product be revealed to be faulty once deployed/ installed on the customers’ devices.

1.4 Integration Testing

- Continuous Integration
  - Continuous integration (CI) is a step that facilitates incremental integration.
    - Incremental integration is when every piece of code is installed in the integration environment and integrated as soon as it is finished.
  - CI is fully automated with the followings steps:

1.4 Integration Testing

- Continuous Integration Contd
  - Steps in the CI Process

    1. The team maintains a version-managed code in a **central repository** with frequent updates.
    2. Automatic code integration in the CI server includes the sequential following of the steps below:
       1. Code is compiled and the warnings and errors are logged.
       2. Logging the results of the static code analysis in terms of quality metrics and coding guidelines.
       3. The code is deployed and installed in the test environment once 2.1 and 2.2 are successful.
       4. Initialization
       5. Unit testing and logging the results to the CI server
       6. Integration testing and logging the results to the CI server
       7. System testing and logging the results to the CI server
       8. CI server displays all results on a dashboard in real-time

1.4 Integration Testing

- Implementing CI
  - Preparing the CI environment should be done before the first sprint as it may include a complete reorganization of the team. The process may take up to several weeks.
  - The steps in setting up a CI environment are as follows:

    1. Check the current state of the configuration management system
       What's the CM tool used? How widely is it used? How does the team use the tool? How long does it take to compile a build? etc.
       Questions such as the above should be answered and addressed appropriately

    2. Explore available CI server software with the team

1.4 Integration Testing

- Implementing CI Contd.
  - Having followed the first two steps to set up the CI environment, then move on to the following steps to implement the CI system:

    1. Selecting CM and CI tools
       A CI server has to reliably identify changes in code within the CM system.
    2. Installing CM and CI tools
       Installation of the selected tools on a separate computer.
    3. Migrate CM
       Importing the old code repository to the new CM.
    4. CI scripting
       The scripts received with the software is adapted to the project environment.

1.4 Integration Testing

- Implementing CI Contd.
  - At this point, the system is similar to a conventional CM system as it does not include automated testing yet.
  - Therefore, appropriate automated test packages are gradually adapted and installed.
  - The following outline few points to strategise the degree of automation required:
    - Every build should automatically be unit tested.
    - Integration tests should be added.
      - And if the team deems it necessary then, the unit and integration test environments can be separated.
    - System tests should be automated and included in the CI system.

1.4 Integration Testing

- Integration Test Management
  - Should ensure that sufficient numbers of tests are written and run.
  - Integration tests should reflect the planned architecture of the system.
  - Integration tests should be designed using the Test First principle.
  - Results of the integration tests can be used to update the system's architecture in terms of architecture diagrams or automated Test First test cases.
  - Integration strategy is dictated by the Story Map.
  - Individual sprint integration strategy is dictated by the Sprint Backlog.

1.4 Integration Testing

- Integration Test Management Contd.
  - Due to continuous integration, sprint planning determines the integration test sequence in advance.
  - Integration testing effort should be taken into consideration in Sprint Planning.
    - The effort required depends on the number of dependencies between the components and not on the number of components itself.
  - Additional integration-related code analyses should be carried out when possible.
  - Integration tests should be sorted into batched.
  - The speed of the CI process should continuously be optimised.

1.5. System Testing and Testing Nonstop

- System Testing
  - Scrum produces shippable products at the end of every sprint that require to have a user interface and the capability interoperating with the customer's existing systems.
  - And system tests check that the product works from the user's perspective with the user's own interfaces.
  - These elements are not covered by unit or integration testing.
  - System test cases trigger a flow of data that passes through the entire system.
  - System test cases are derived directly from the requirements and the acceptance criteria in the Product Backlog or use cases.

1.5. System Testing and Testing Nonstop

- System Testing Contd
  - System test environments are more complex than other test environments
  - System test environments replicate the production environment as accurately as possible by representing the external factors the system will work with in the user's .
  - The environment would have as many real-world components (Hardware, software, networks, other systems as possible) as possible resulting in a large number of ways that these components can be configured in.
  - The system testing effort depends on the number of different test environments that need to be checked and not the number of test cases itself.

1.5. System Testing and Testing Nonstop

- Manual System Testing

1. Exploratory Testing
   - These tests begin by only defining the objectives of the test (eg: a feature or a user story to be tested) and the test will primarily focus on this.
   - The structure and the individual steps are not predefined and are decided depending on the tester's observations during the test.
   - Components that behave normally will be lightly tested or skipped altogether, focusing on components that display unusual behaviour to find the cause.
   - These tests are ideal for checking new features quickly.
   - The quality of these tests depend heavily on the tester's competence therefore, are difficult to be reproduced.

1.5. System Testing and Testing Nonstop

- Manual System Testing Contd

  2. Session-Based Testing
     - The tester briefly describes the objectives and the strategy for the test in 2-3 lines.
     - Test setup, design and execution, defect localization and defect reporting are all limited to a maximum of 90 minutes.
     - The objectives, procedures, coverage, tested elements, defects discovered, etc. are noted in a session sheet.
       - Session sheets based on keywords facilitate long-term test automation.
     - Session-based testing overcomes the disadvantages of exploratory testing such as the inability to reproduce the tests, by electronically logging the session sheets.

1.5. System Testing and Testing Nonstop

- Manual System Testing Contd

  2. Session-Based Testing Contd
     - The tests provide an insight into the current state of the product and how users interact with it.

  3. Acceptance Testing
     - Is an approval test selected by the customer.
     - Helps determine if the system is acceptable.
     - Identifies if the system does what was intended with the correct functionality.
       - *Did we build the right system?*
     - A sprint's acceptance tests will focus primarily on newly added features.
     - Acceptance tests alone are insufficient to identify possible unwanted side effects of new changes to a system.

1.5. System Testing and Testing Nonstop

- Automated System Testing
  - System testing is difficult to be automated due to the following reasons:
    - Complex environments
    - Requires dedicated GUI tools as the main testing interface is the GUI and as GUI tests require reactions from other components, they're relatively much slower.
    - A clearly defined and reproducible original state of the system configuration is required to base the test scripts on
    - It's difficult to automatically compare the expected behaviour with actual behaviour.
    - There are many interfaces involved and therefore additional tests and tools are required.
    - Lack of team members who are experienced in automating system tests.

1.5. System Testing and Testing Nonstop

- Automated System Testing

  1. Record/ Playback Testing
     - A record/ playback tools are used to record all user commands (Eg: manual keyboard and mouse commands) given during the test as a script.
     - Running the saved script reproduces the test.
     - Drawbacks -
       - It is recommended that new test cases are written during each Sprint for new functions rather than updating existing tests because the GUI changes significantly in each Sprint.
       - Does not cater to alternative GUIs or GUI layouts.

1.5. System Testing and Testing Nonstop

- Automated System Testing Contd.

  2. Keyword-Driven Testing
     - Uses domain-specific vocabulary to describe the test procedure instead of general navigation commands.
     - The test cases represent what the system should be capable of doing but not how it works.
     - These are easily understood by technical and non-technical members.
     - A change in implementation (such as an interface change) will not affect the test case's validity.
     - Drawbacks -
       - The team has to agree on a stable vocabulary and assign an interpreter to convert the commands.
       - Requires a sequence control mechanism and an adapter to connect the mechanism to the test objects.
       - Each test object needs to be a unique ID that will remain the same throughout the Sprints.
       - Time and resource consuming to set up a keyword-based system test automation.

1.5. System Testing and Testing Nonstop

- Automated System Testing Contd.

  3. Behaviour-Driven Testing (BDT)
     - Uses behaviour-driven techniques and natural language.
     - The tester can use central keywords to define BDT scripts.
     - A feature is tested using various scenarios.
     - A scenario corresponds to a test case and is divided into sections based on the *Given-When-Then* principle.
     - BDT uses the test object's API as the test automation interface.

1.5. System Testing and Testing Nonstop

- Using Test First for System Testing
  - The Test First principle cannot be applied when GUI test tools are being used as they require direct access to the system's user interface where they already have to be in existence.
  - But, if test cases are keyword-based or BDT based then because these types of tests are independent of the system's technical implementations and user interfaces, can be created before the test object actually exists. And thereby allow Test First based system test automation.

1.5. System Testing and Testing Nonstop

- Using Test First for System Testing Contd.
  - System Test Repository
    - A centralised repository enables collecting and managing a team's standardised set of notations and keywords for BDT or Keyword-Based test case creation.

1.5. System Testing and Testing Nonstop

- Using Test First for System Testing Contd.
  - Pair Programming
    - Pair Programming enables the following in terms of implementing Test First for system testing:
      - A programmer and a tester should together design the test cases, where the programmer will ensure that the correct feature is being focused on and the tester will ensure that the test cases represent the user's point of view.
      - A pair of testers produce better results when drafting and maintaining the standard set of keywords.
      - Teams of testers and programmers together write the relevant programs in GUI test tool's scripting language and perform xUnit programming to enable the implementation of keywords on all tiers of the test sequence controller.

1.5. System Testing and Testing Nonstop

- Non-Functional Testing
  - Checks how well and at which level of quality the system performs its intended tasks.
  - The system is tested for efficiency, compatibility, usability, reliability, security, maintainability and portability.
  - Given below are types of non-functional tests:

    1. Load Testing
       Checks system behaviour with increasing load (eg: parallel user, database transactions)

    2. Performance Testing
       Measures the processing speed and response time for specific use cases.

1.5. System Testing and Testing Nonstop

- Non-Functional Testing Contd.

3. Volume/ Mass/ Stress Testing
   Checks system behaviour for varying amounts of data and when overloaded.

4. Security Testing
   Observes the system against unauthorised system or data access.

5. Reliability and Stability Testing
   Testing the system under continuous use.

6. Compatibility and Data Conversion Testing
   Observes the level of interoperability of the system with other systems and processes.

1.5. System Testing and Testing Nonstop

- Non-Functional Testing Contd.

7. Robustness Testing
   Tests error handling and recovery with misuse, programming faults , hardware failure, etc.

8. Configuration Testing
   Observes the system against unauthorised system or data access.

9. Usability Testing
   Determines if the system is easy to use and learn.

10. Documentation Checks
    Checks if the system behaviour matches the documentation.

1.5. System Testing and Testing Nonstop

- Non-Functional Testing Contd.

11. Changeability and Maintainability Testing
    Checks how up-to-date and accurate the documentation, system structure, etc. are.

- Challenges:
  - Load, performance, volume and other, similar tests are, extensive and long-term and therefore, would slow down the CI process.
  - Robustness, hardware failure, recovery and other, similar tests are difficult to automate as they require manual intervention during the environment setup.
  - Usability, documentation and code maintainability tests are run manually with extensive reviews.

1.5. System Testing and Testing Nonstop

- Non-Functional Testing Contd.
  - Non-Functional requirements should be addressed early on in the project.
  - If feedback from Non-Functional testing is not available on a timely manner, comprehensive refactoring of the system may be required later on.
  - Solutions for addressing the issue are given below:
    - Designing, automation and performance of tests to check non-functional attributes of features during the Sprint.
    - Performing robustness, hardware failure and other, similar tests as exploratory tests.
    - Performing user-friendliness, documentation, code maintainability and other, similar tests continuously from an early stage using pair programming.
    - Conducting regular reviews of the test results.

1.5. System Testing and Testing Nonstop

- Automated Acceptance Testing
  - When designing a set of acceptance tests (acceptance test suite), the Product Owner can use already automated unit or integration tests that cater to a specific approval test requirement.
  - Product Owner may opt for automated system testing instead of testing manually.
  - When selecting the tests he should focus on the acceptance criteria that need to be covered.
  - And should only perform manual testing for the criteria that are not covered by the existing automated tests.

1.5. System Testing and Testing Nonstop

- System Test Management
  - Address system testing tasks (eg: writing test cases, setting up the environment and automation) in the Sprint Planning and allocate effort for those tasks.
  - Individual tests and test results should be reviewed periodically.
  - Test cases should be updated and replaced continuously.
  - All team members should be aware of the test results.
  - A dedicated test manager can be elected to examine the quality of tests, interpret results, defining the team's testing tasks, etc.

1.5. System Testing and Testing Nonstop

- System Test Management Contd.
  - Daily bugs should be analysed prior to the daily standup so bugs that require dedicated tasks can be decided quickly.
  - If there are manual tests to be performed, the test manager should which are to be run as regression tests and how frequently.
  - If there's a large number of manual tests, they should be selected based on risk assessment.
  - Manual tests should be monitored, and pair programming, predefined tests and checklists will facilitate that.
  - The test framework should be built step by step including only the components that are needed immediately.

Summary

Testing in waterfall method vs Agile Testing

Agile Quality Assurance vs Traditional Quality Assurance

- Traditional Quality Assurance Tools
- Agile Quality Assurance Tools
- Test Planning in Scrum

Unit Testing

- The Test First approach
- Unit Testing Frameworks
- Unit Test Management

Summary

Integration Testing	• Designing Integration Test Cases • Dependencies and Interfaces • Integration Levels • Continuous Integration • Implementing CI • Integration Test Management
System Testing and testing non-stop	• Manual System Testing • Automated System Testing • Using Test First for System Testing • Non-functional Testing • Automated Acceptance Testing • System Test Management