MODULE I

Chapter 1

**Software delivery**

For successful software, the ﬁrst release is just the beginning of the delivery process .

In order to achieve ways to deliver high-quality, valuable software in an efﬁcient, fast, and reliable manner, We need to make **frequent**, **automated** releases of our software.

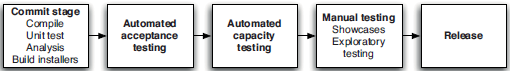
**Frequent-**If releases are frequent, the delta between releases will be small. This signiﬁcantly reduces the risk associated with releasing and makes it much easier to roll back. Frequent releases also lead to faster feedback

**Automated-**If the build, deploy, test, and release process is not automated, it is not repeatable. Every time it is done, it will be different, because of the conﬁguration of the system, the environments, and the release process.

**Principles of Software Delivery**

A **deployment pipeline** is an automated implementation of your application’s build, deploy, test, and release process.

•Every organization will have differences in the implementation of their deployment pipelines, depending on their value stream for releasing software, but the principles that govern them do not vary.



The deployment pipeline

•One of the first steps in the pipeline is to create binaries and installers.

•The rest of the pipeline runs a series of tests on the binaries to prove that they can be released.

•Each test that the release candidate passes gives us more confidence that this particular combination of binary code, configuration information, environment, and data will work.

•If the release candidate passes all the tests, it can be released.

**The Release Candidate**

•A *release candidate*(*RC*), also known as "going silver", is a beta version with potential to be a final product, which is ready to release unless significant bugs emerge.

•The process build internally to check if any critical problems have gone undetected into the code during the previous development period.

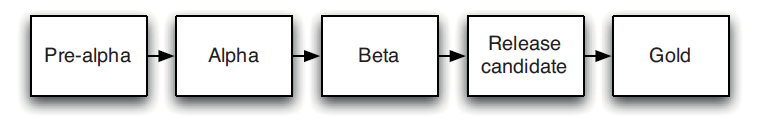
•It is the build, deployment, and test process that we apply to that change that validates whether the change can be released.

•This process gives us increasing confidence that the change is safe to release.

•Release candidate and Beta testing are different.

•If defects are found, then a round of testing is performed to ensure there are no further issues.

•If the resulting product is found to be free of defects, and it meets the acceptance criteria set out by the customer, then it can be released.



Traditional view of release candidate

•Every change is, in effect, a release candidate. Every time a change is committed to version control, the expectation is that it will pass all of its tests, produce working code, and can be released into production.

•Release candidates are NOT for production deployment, but they are for **testing** purposes only.

**Principles of Software Delivery**

1. Create a Repeatable, Reliable Process for Releasing Software.

2 .Automate Almost Everything.

3. Keep Everything in Version Control.

4. If It Hurts, Do It More Frequently, and Bring the Pain Forward.

5. Build Quality In.

6. Done Means Released.

7. Everybody Is Responsible for the Delivery Process.

8. Continuous Improvement.

**1. Create a Repeatable, Reliable Process for Releasing Software**

•Releasing software should be easy. It should be easy because you have tested every single part of the release process hundreds of times before.

•The repeatability and reliability derive from two principles: automate almost everything, and keep everything you need to build, deploy, test, and release your application in version control.

Deploying software ultimately involves three things:

• Provisioning and managing the environment in which your application will run (hardware configuration, software, infrastructure, and external services).

• Installing the correct version of your application into it.

• Configuring your application, including any data or state it requires.

The deployment of your application can be implemented using a fully automated process from version control.

Application configuration can also be a fully automated process, with the necessary scripts and state kept in version control or databases.

2. **Automate Almost Everything**

•Your build process should be automated up to the point where it needs specific human direction or decision making.

•Acceptance tests can be automated.

•Database upgrades and downgrades can be automated too.

•Even network and firewall configuration can be automated.

•Automation is a prerequisite for the deployment pipeline, because it is only through automation that we can guarantee that people will get what they need at the push of a button.

3.**Keep Everything in Version Control**

•Everything you need to build, deploy, test, and release your application should be kept in some form of versioned storage.

•This includes requirement documents, test scripts, automated test cases, network configuration scripts, deployment scripts, database creation, upgrade, downgrade, and initialization scripts, application stack configuration scripts, libraries, tool chains, technical documentation, and so on.

•All of this stuff should be version-controlled, and the relevant version should be identifiable for any given build. That is, these change sets should have a single identifier, such as a build number or a version control change set number, that references every piece.

• It should be possible for a new team member to sit down at a new workstation, check out the project’s revision control repository, and run a single command to build and deploy the application to any accessible environment, including the local development workstation.

4. **If It Hurts, Do It More Frequently, and Bring the Pain Forward**

•If testing is a painful process that occurs just before release, don’t do it at the end. Instead, do it continually from the beginning of the project.

•If releasing software is painful, aim to release it every time somebody checks in a change that passes all the automated tests. If you can’t release it to real users upon every change, release it to a production-like environment upon every check-in.

•If creating application documentation is painful, do it as you develop new features instead of leaving it to the end. Make documentation for a feature part of the definition of done, and automate the process as far as possible.

•If creating application documentation is painful, do it as you develop new features instead of leaving it to the end. Make documentation for a feature part of the definition of done, and automate the process as far as possible.

5.**Build Quality In**

•The earlier you catch defects, the cheaper they are to fix. Defects are fixed most cheaply if they are never checked in to version control in the first place.

•The techniques such as continuous integration, comprehensive automated testing, and automated deployment, are designed to catch defects as early in the delivery process as possible.

•Delivery teams must be disciplined about fixing defects as soon as they are found.

•Everybody on the delivery team is responsible for the quality of the application all the time.

6. **Done Means Released**

•For some agile delivery teams, “done” means released into production. This is the ideal situation for a software development project. However, it is not always practical to use this as a measure of done.

•The initial release of a software system can take a while before it is in a state where real external users are getting benefit from it.

•A functionality is “done” once it has been successfully showcased, that is, demonstrated to, representatives of the user community, from a production-like environment.

7. **Everybody Is Responsible for the Delivery Process**

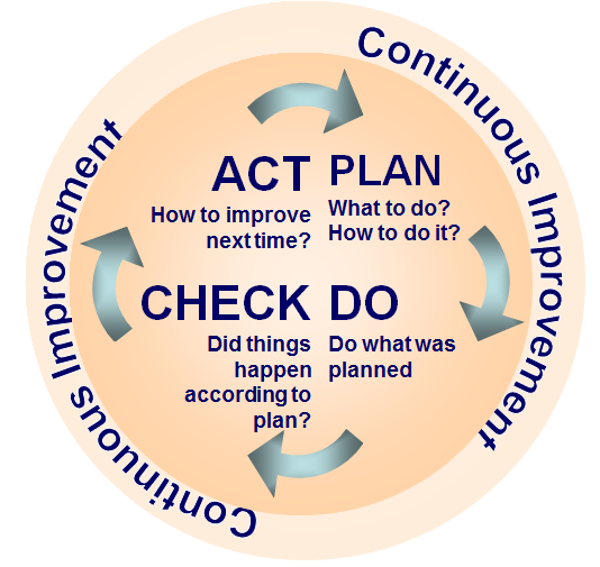
•Ideally, everybody within an organization is aligned with its goals, and people work together to help each to meet them. Ultimately the team succeeds or fails as a team, not as individuals. However, in too many projects the reality is that developers throw their work over the wall to testers.

•Then testers throw work over the wall to the operations team at release time. When something goes wrong, people spend as much time blaming one another as they do fixing the defects that inevitably arise from such a siloed approach.

8. **Continuous Improvement**

•The first release of an application is just the first stage in its life. All applications evolve, and more releases will follow. It is important that your delivery process also evolves with it.

•The whole team should regularly gather together and hold a retrospective on the delivery process. Somebody should be nominated to own each idea and ensure that it is acted upon. Then, the next time that the team gathers, they should report back on what happened. This is known as the Deming cycle: plan, do, check(study), act.



Deming cycle

Chapter 2

“**Conﬁguration Management,”** sets out how to manage everything required to build, deploy, test, and release your application, from source code and build scripts to your environment and application conﬁguration.

**Configuration Management**

•Configuration management is a term that is widely used, often as a synonym for version control.

•Configuration management refers to the process by which all artifacts relevant to your project, and the relationships between them, are stored, retrieved, uniquely identified, and modified.

•Configuration management strategy will determine how you manage all of the changes that happen within your project.

•Divided into:

1. Getting everything into version control.

2. Managing dependencies.

3. Managing software configuration.

4. Configuration management of whole environments.

**Version Control**

•A component of software Configuration management, Version control system is the management system that manages the changes that you make in your project till the end.

•The changes that you make might be some kind of adding some new files or modifying the older files by changing the source code.

•Every time you make a change in your project it creates a snapshot of your entire project & save it. These snapshots are actually known as the different versions.

•Examples:

•SCCS (Source Code Control System)

•CVS (Concurrent Versions System)

•GIT

**How to make the most effective use of version control.**

*Keep Absolutely Everything in Version Control*

•Every single artifact related to the creation of your software should be under version control.

•Every member of the team should store any document or file related to the project in version control (Analysts should store their requirement documents , Testers their test scripts and procedures , Project managers their release plans , progress charts etc) .

*Check In Regularly to Trunk*

•Once you check your changes into version control, they become public, instantly available to everybody else on the team.

•Since checking in is a form of publication, it is important to be sure that your work, whatever it may be, is ready for the level of publicity that a check-in implies.

*Use Meaningful Commit Messages*

•The benefits of version control are enhanced when you commit regularly.

•Every version control system has the facility to add a description to your commit.

•The most important reason to write descriptive commit messages is so that, when the build breaks, you know who broke the build and why.

•In a multi paragraph commit message the first paragraph is a summary and the following paragraphs add more detail.

**Managing Dependencies** •The most common external dependencies within your application are the third party libraries it uses and the relationships between components or modules under development by other teams within your organization. •Libraries are typically deployed in the form of binary files, are never changed by your application’s development team, and are updated very infrequently.*Managing External Libraries* •External libraries usually come in binary form, unless you’re using an interpreted language. Even with interpreted languages, external libraries are normally installed globally on your system by a package management system. •Keep copies of external libraries somewhere locally. This is essential if you have to follow compliance regulations, and it also makes getting started on a project faster. *Managing Components* •It’s good practice to split all but the smallest applications into components. Doing so limits the scope of changes to your application, reducing regression bugs. •It also encourages reuse and enables a much more efficient development process on large projects.**Managing Software Configuration**

•Configuration is one of the three key parts that comprise an application, along with its binaries and its data.

•Configuration information can be used to change the behaviour of software at build time, deploy time, and run time.

*Configuration and Flexibility*

•Configuration information is somehow risky to change than source code.

•Configuration information can be injected into your application at several points in your build, deploy, test, and release process, and it’s usual for it to be included at more than one point.

*Types of Configuration*

O Your build scripts can pull configuration in and incorporate it into your binaries at **build time**.

O Your packaging software can inject configuration at **packaging time**, such as when creating assemblies.

O Your deployment scripts or installers can fetch the necessary information or ask the user for it and pass it to your application at **deployment time** as part of the installation process.

O Your application itself can fetch configuration at **startup time** or **run time**.

**Accessing Configuration**

•The easiest way for an application to access its configuration is via the file system. This has the advantage of being cross-platform and supported in every language.

**Modeling Configuration**

•Each configuration setting can be modeled as a tuple, so the configuration for an application consists of a set of tuples.

•The set of the tuples available and their values typically depend on three things:

O The application , The version of the application , The environment it runs in (for example, development, UAT, performance, staging, or production)

**Testing System Configuration**

•There are two parts to testing configuration.

•The first stage is to ensure that references to external services in your configuration settings are good.

•The second stage is to actually run some smoke tests once your application is installed to make sure it is operating as expected.

•These tests should stop the application and fail the installation or deployment process if the results are not as expected.

*Managing Configuration across Applications*

•The problem of managing configuration is particularly complex in medium and large organizations where many applications have to be managed together.

•The goal is to be able to see each application’s configuration through your operation team’s production monitoring system, which should also display which version of each application is deployed in each environment.

•Tools such as Nagios also provide services like periodic checks on critical parameters of application, network and server resources.

*Principles of Managing Application Configuration*

•Consider where in your application’s lifecycle it makes sense to inject a particular piece of configuration.

•Keep the available configuration options for your application in the same repository as its source code, but keep the values somewhere else.

•Configuration should always be performed by automated processes using values taken from your configuration repository.

•Use clear naming conventions for your configuration options. Avoid obscure or cryptic names.

•Ensure that your configuration information is modular and encapsulated.

•Use the DRY (don’t repeat yourself) principle. Define the elements of your configuration so that each concept has only one representation in the set of configuration information.

•Be minimalist: Keep the configuration information as simple and as focused as possible.

**Configuration Management of all Environments**

•In order to reduce the cost and risk of managing environments, it is essential to turn our environments into mass-produced objects whose creation is repeatable and takes a predictable amount of time.

•The key to managing environments is to make their creation a fully automated process. It should always be cheaper to create a new environment than to repair an old one.

**Configuration Management of all Environments**

•An environment that is in a properly deployed state is known as a baseline in configuration management terminology. Your automated environment provisioning system should be able to establish, or re-establish, any given baseline that has existed in the recent history of your project.

•Any time you change any aspect of the host environment of your applications, you should store the change, creating a new version of the baseline and associating that version of the application with the new version of the baseline.

*Tools to Manage Environments*

•Puppet and Cf Engine are two examples of tools that make it possible to manage operating system configuration in an automated fashion.

•Using these tools, you can declaratively define things such as which users should have access to your boxes and what software should be installed.

•These definitions can be stored in your version control system.

•Agents running on your systems regularly pull the latest configuration and update the operating system and the software installed on it.

*Managing the Change Process*

•A production environment should be completely locked down. It should not be possible for anybody to make a change to it without going through your organization’s change management process.

•The reason for this is simple: Even a tiny change could break it.

•A change must be tested before it goes into production, and for that it should be scripted and checked into version control.

•Then, once the change has been approved, it can be rolled out to the production environments in an automated fashion.

**CHAPTER 3**

**CONTINOUS INTEGRATION**

**“Continuous Integration,”** covers the practice of building and running automated tests against every change you make to your application so you can ensure that your software is always in a working state.

**Continuous Integration (CI)**

•**Continuous integration** is the practice of merging all developer working copies to a shared mainline.

•Continuous integration requires that every time somebody commits any change, the entire application is built and a comprehensive set of automated tests is run against it.

•If the build or test process fails, the development team stops whatever they are doing and fixes the problem immediately.

•The goal of continuous integration is that the software is in a working state all the time.

•Continuous integration represents a paradigm shift.

•With continuous integration, your software is proven to work with every new change and you know the moment it breaks and can fix it immediately.

•The teams that use continuous integration effectively are able to deliver software much faster, and with fewer bugs, than teams that do not.

•Bugs are caught much earlier in the delivery process when they are cheaper to fix, providing significant cost and time savings. Hence we consider it an essential practice for professional teams, perhaps as important as using version control.

**Implementing Continuous Integration**

•**Three** things that you need before you can start with continuous integration:

1.Version Control

2.An Automated Build

3. Agreement of the Team

•**A Basic Continuous Integration System**

•Don’t need a continuous integration software in order to do continuous integration—as we say, it is a practice, not a tool.

•CI tools these days are extremely simple to install and get running. There are several open source options, such as Hudson and the venerable Cruise Control family (Cruise Control, CruiseControl.NET, and Cruise Control.rb).

•Once you have your CI tool of choice installed, given the preconditions described, it should be possible to get started in just a few minutes by telling your tool where to find your source control repository, what script to run in order to compile, if necessary, and run the automated commit tests for your application, and how to tell you if the last set of changes broke the software.

**Prerequisites for Continuous Integration**

**Check In Regularly**

•The most important practice for continuous integration to work properly is frequent check-ins to trunk or mainline.

•You should be checking in your code at least a couple of times a day.

•It makes your changes smaller and thus less likely to break the build.

•Checking into trunk on purpose.

•Many projects use branches in version control to manage large teams. But it is impossible to truly do continuous integration while using branches because, by definition, if you are working on a branch, your code is not being integrated with that of other developers.

**Create a Comprehensive Automated Test Suite**

•It’s essential to have some level of automated testing to provide confidence that your application is actually working.

•There are three kinds of tests we are interested in running from our continuous integration build: unit tests, component tests, and acceptance tests.

•**Unit tests** are written to test the behaviour of small pieces of your application in isolation (say, a method, or a function, or the interactions between a small group of them).

•They can usually be run without starting the whole application.

•They do not hit the database (if your application has one), the file system, or the network. They don’t require your application to be running in a production-like environment.

•Unit tests should run very fast—your whole suite, even for a large application, should be able to run in under ten minutes.

•**Component tests** test the behavior of several components of your application.

•Like unit tests, they don’t always require starting the whole application. However, they may hit the database, the file system, or other systems (which may be stubbed out).

•Component tests typically take longer to run.

•**Acceptance tests**test that the application meets the acceptance criteria decided by the business, including both the functionality provided by the application and its characteristics such as capacity, availability, security, and so on.

•Acceptance tests are best written in such a way that they run against the whole application in a production-like environment.

•Acceptance tests can take a long time to run—it’s not unheard of for an acceptance test suite to take more than a day to run sequentially.

**Keep the Build and Test Process Short**

•Problems for long build and test Process:

O People will stop doing a full build and running the tests before they check in. You will start to get more failing builds.

O The continuous integration process will take so long that multiple commits will have taken place by the time you can run the build again, so you won’t know which check-in broke the build.

O People will check in less often because they have to sit around for ages waiting for the software to build and the tests to run.

•There are a number of techniques that you can use to reduce the build time.

•The first thing to consider is making your tests run faster. XUnit-type tools, such as JUnit and NUnit, provide a breakdown of how long each test took in their output. Find out which tests are performing slowly, and see if there’s a way to optimize them or get the same coverage and confidence in your code with less processing. This is a practice that you should perform regularly.

•At some point you will need to split your test process into multiple stages.

•Your first action should be creating two stages. One should compile the software, run your suite of unit tests that test individual classes making up your application, and create a deployable binary. This stage is called the commit stage.

•The second stage should take the binaries from the first stage and run the acceptance tests, as well as integration tests, and performance tests if you have them.

•Modern CI servers make it easy to create staged builds in this way, run multiple tasks concurrently, and aggregate the results up so you can see the state of your build at a glance.

•**Managing Your Development Workspace**

•Developers should always work from a known good starting point when they begin a fresh piece of work. They should be able to run the build, execute the automated tests, and deploy the application in an environment under their control.

•The first pre requisite to achieve this is careful configuration management, not just of source code, but also of test data, database scripts, build scripts, and deployment scripts.

•The second step is configuration management of third-party dependencies, libraries, and components.

•The final step is to make sure that the automated tests, including smoke tests, can be run on developer machines.

**CHAPTER-4**

“**Implementing a Testing Strategy”** introduces the various kinds of manual and automated testing that form an integral part of every project, and discusses how to decide which strategy is appropriate for your project.

**Implementing a Testing Strategy**

•Manual acceptance testing is applied to many projects to verify that a piece of software conforms to its functional and non functional requirements.

•In project, testers collaborate with developers and users to write automated tests from the start of the project. These tests are written before developers start work on the features that they test.

•The automated test suite is run by the CI system every time a change is made to the application also serves as a set of regression tests.

•Aim is to help to plan and implement effective automated testing systems.

•The design of a testing strategy is primarily a process of identifying and prioritizing project risks and deciding what actions to be taken to migrate them.

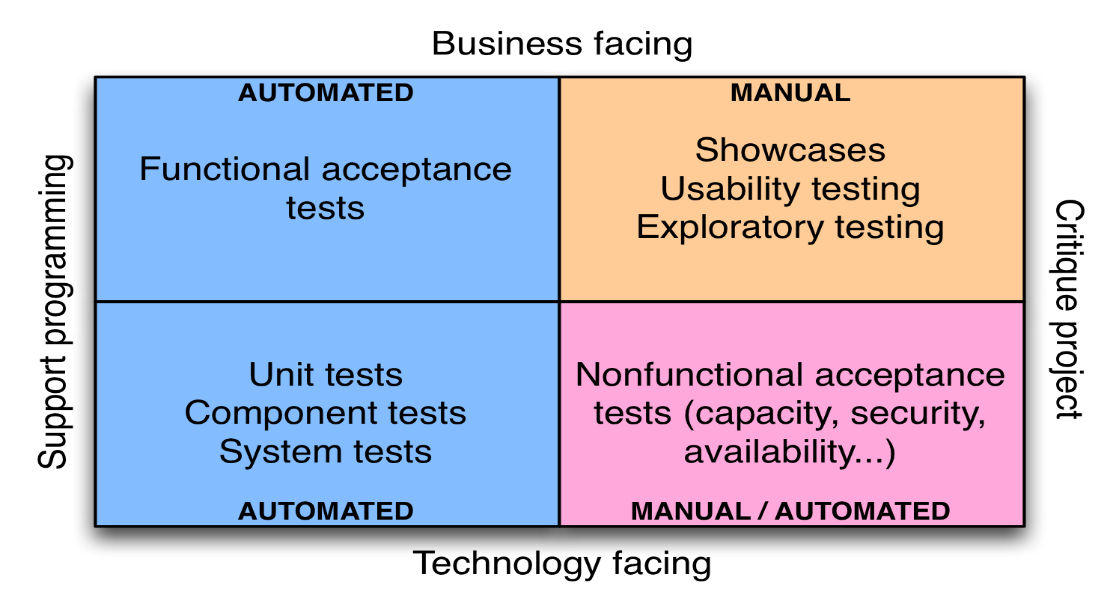
•A good testing strategy establishes confidence that the software is working as it should ie , reduced support costs, fewer bugs and improved reputation.

•A comprehensive automated test suite provides a complete and up to date form of application documentation.

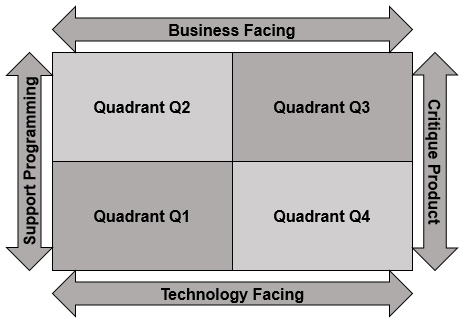
**Types of Tests**

•Many kinds of testing exist. “Testing quadrant diagram” is widely used to model the various types of tests to ensure the delivery of a high-quality application.

•In this diagram, the tests are categorized according to whether they are business-facing or technology-facing, and whether they support the development process or are used to critique the project.



Testing quadrant diagram



Agile Testing Quadrants by Brian Marick

**1. Business-Facing Tests That Support the Development Process.**

•A Test is a business-facing test if it answers the questions framed with words from business domain.

•These are understood by the business experts and would interest them so that behavior of the system can be explained in the real time scenario.

•The tests in this quadrant are more commonly known as functional or acceptance tests. Acceptance testing ensures that the acceptance criteria for a business are met.

•Acceptance tests, can test all kinds of attributes of the system being built, including functionality, capacity, usability, security, modifiability, availability, and so on.

•When the acceptance tests pass, whatever requirements or stories they are testing can be said to be complete. Thus, in an ideal world, customers or users would write acceptance tests, since they define the success criteria for each requirement.

•Acceptance tests should be run when your system is in a production-like mode.

•Manual acceptance testing is typically done by putting an application in a user acceptance testing (UAT) environment.

**Automating Acceptance Tests**

•Properties:

1.They make the feedback loop faster (developers can run automated tests without having to go to testers).

2.They reduce the workload on testers.

3. They free testers to concentrate on exploratory testing and higher-value activities instead of boring repetitive tasks.

4. Your acceptance tests represent a powerful regression test suite.

5. By using human-readable test and test suite names, as advocated by behavior-driven development, it is possible to auto generate requirements documentation from your tests. So your requirements documentation is never out-of-date—it can be generated automatically with every build.

•Tools like Cucumber, Selenium and Twist are designed to allow analyst to write requirements as executable test scripts.

•Automated acceptance tests can be costly to maintain. By following good practices and using appropriate tools it is possible to dramatically reduce the cost of creating and maintaining automated acceptance tests.

**2.Technology-Facing Tests that Support the Development Process.**

•These automated tests are written and maintained exclusively by developers.

•There are three kinds of tests that fall into this category:

••**Unit tests** test a particular piece of the code in isolation.

O Unit tests should not involve calling the database, using the file system, talking to external systems, or, in general, interaction between components of a system.

O This enables them to run very fast so you can get early feedback on whether changes have broken any existing functionality.

•**Component tests** test larger clusters of functionality.

O They are typically slower, since they can require more involved setup and perform more I/O, talking to databases, the file system, or other systems.

O Sometimes, component tests are known as “integration tests”.

•**Deployment tests** are performed whenever you deploy your application. They check that the deployment worked—in other words, that your application is correctly installed, correctly configured, able to contact any services it requires, and that it is responding.

**3. Business-Facing Tests That Critique the Project**

•These manual tests verify that the application will in fact deliver to the users the value they are expecting.

•This is not just a matter of verifying that the application meets its specifications; it is also about checking that the specifications are correct.

•A particularly important form of business-facing, project-critique tests are **showcases**.

O Agile teams perform showcases to users at the end of every iteration to demonstrate the new functionality that they have delivered. Functionality should also be demonstrated to customers as often as possible during development, so as to ensure that any misunderstandings or specification problems are caught as early as possible.

•**Exploratory testing** is as a form of manual testing in which “the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests”.

O Exploratory testing is a creative learning process that will not only discover bugs, but also lead to the creation of new sets of automated tests, and potentially feed into new requirements for the application.

•**Usability testing** is done to discover how easy it is for users to accomplish their goals with your software.

O Usability testing is therefore the ultimate test that your application is actually going to deliver value to users.

O Usability testers gather metrics, noting how long it takes users to finish their tasks, watching out for people pressing the wrong buttons, noting how long it takes them to find the right text field, and getting them to record their level of satisfaction at the end.

•Finally, you can give your application to real users using beta testing programs.

**4.Technology-Facing Tests That Critique the Project.**

•Acceptance testing comes in two categories: functional tests and non-functional tests. By non-functional tests, we mean all the qualities of a system other than its functionality, such as capacity, availability, security, and so forth.

•The tests used to check whether the acceptance criteria have been met, and the tools used to run the tests tend to be quite different from those used to verify conformance to functional acceptance criteria.

•These tests often require considerable resources such as special environments to run on and specialized knowledge to set up and implement, and they often take a long time to run (whether or not they are automated).

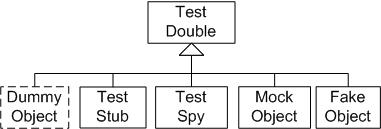
**Test Doubles**

•A key part of automated testing involves replacing part of a system at run time with a simulated version.

•A test double is an object that can stand in for a real object in a test.

•In this way, the interactions of the part of the application under test with the rest of the application can be tightly constrained, so that its behaviour can be determined more easily. Such simulations are generally known as test doubles.

The various types of test doubles:



**1.Dummy objects** are passed around but never actually used. Usually they are just used to fill parameter lists.

**2.Fake objects** actually have working implementations, but usually take some shortcut that makes them not suitable for production. (A good example of this is the in-memory database ie. the use of a small-footprint, diskless database instead of a full-featured disk-based database.

**3.Stubs** provide canned answers to the calls made during the test, usually not responding at all to anything outside what’s programmed in for the test. A stub has no logic, and only returns what you tell it to return.

**4.Spies** are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.

**5.Mocks** are pre-programmed with expectations that form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don’t expect and are checked during verification to ensure they got all the calls they were expecting.