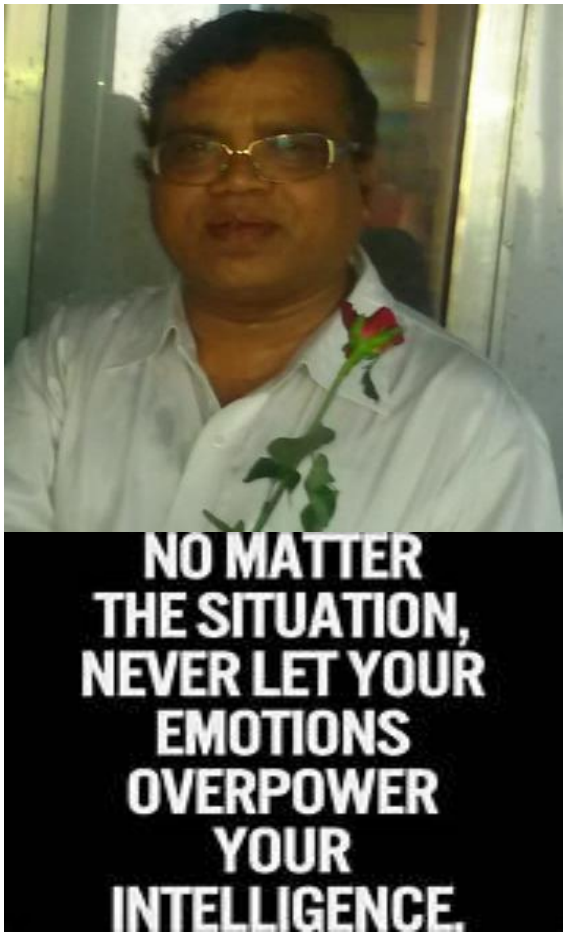
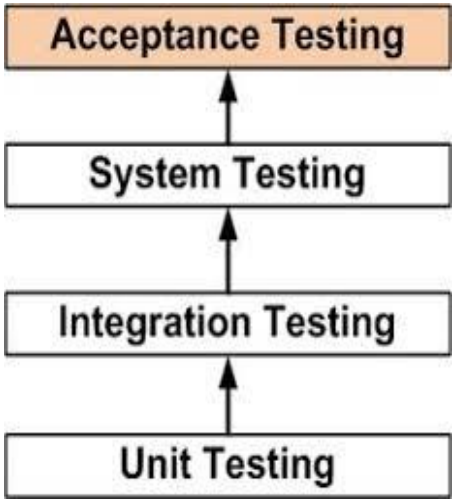


TESTING LOGIC

- Unit Testing-White Box Testing
- Integration Testing-White Box Testing
- System Testing- (all components of systems)-black box testing
- Acceptance Testing-software ready to be released into the market.



By

Dr. Bandu B. Meshram

PhD(Computer Engineering), LLM(Constitutional law)
Computer Hacking and Forensic Investigator-EC Council Certified, USA

Professor and Former Head,
Department of Computer Engineering & Information Technology,
VJTI, Matunga, Mumbai-19


LOGICAL APPEALS & EVIDENCE
Study.com

OO SOFTWARE TESTING METHODS

Types of software requirements

Business requirements	User requirements	Software requirements
Outline measurable goals for the business.	Reflect specific user needs or expectations.	Identify features, functions, non-functional requirements and use cases.
Define the *why* behind a software project.	Describe the *who* of a software project.	Delve into the *how* of a software project.
Match project goals to stakeholder goals.	Highlight how users interact with it.	Describe software as functional modules and non-functional attributes.
Maintain a BRD with requirements, updates or changes.	Create a URS, or make them part of the BRD.	Compose an SRS, and, optionally, an FRS.

A business requirements document (BRD)

Business needs drive many software projects. A business requirements document (BRD) outlines measurable project goals for the business, users and other stakeholders. Business analysts, leaders and other project sponsors create the BRD at the start of the project. This document defines the *why* behind the build. For software development contractors, the BRD also serves as the basis for more detailed document preparation with clients.

- the basic format of a BRD statement looks like:
- *"The [project name] software will [meet a business goal] in order to [realize a business benefit]."*
- And here's a detailed BRD statement example:
- *"The laser marking software will allow the manufacturing floor to mark text and images on stainless steel components using a suitable laser beam in order to save money in chemical etching and disposal costs."*

- For this example, the purpose of the proposed software project is to operate an industrial laser marking system, which is an alternative to costly and environmentally dangerous chemicals, to mark stainless steel product parts.
- Organizations prepare a BRD as a foundation for subsequent, more detailed requirements documents. Ensure that the BRD reflects a complete set of practical and measurable goals -- and meets customer expectations.
- Finally, the BRD should be a living document. Evaluate any future requirements, updates or changes to the project against the BRD to ensure that the organization's goals are still met.

User requirements statements

- There is no universally accepted standard for user requirements statements, but here's one common format:
- *"The [user type] shall [interact with the software] in order to [meet a business goal or achieve a result]."*
- A user requirement in that mold for the industrial laser marking software example looks like:
- *"The production floor manager shall be able to upload new marking files as needed in order to maintain a current and complete library of laser marking images for production use."*

- Software requirements typically break down into:
- functional requirements
- nonfunctional requirements
- domain requirements- Domain requirements are expectations related to a particular type of software, purpose or industry vertical. Domain requirements can be functional or nonfunctional.
- Domain requirements typically arise in military, medical and financial industry sectors, among others. One example of a domain requirement is for software in medical equipment:
- *The software must be developed in accordance with IEC 60601 regarding the basic safety and performance for medical electrical equipment.*

OO CLASS TESTING

CREATING BACKGROUND FOR OO TESTING

Start OO testing

TESTING UNIT-SMALLEST PART OF THE CODE-UNIT

OO Software Architecture

OO software = Framework reuse (purchased code) + developed code.

There are two types of frameworks

- Called frameworks are very much like traditional libraries in that the application code calls the framework when some framework service is needed.
- Calling framework reverse the role of the framework and the application code, rather than the other way around.

Managing OO framework Reuse

Managing OO framework reuse is divided into four focal points.

- Managing architectural differences
- Managing design assumption Interaction
- Managing framework deployment
- Configuration management

Possible Measure for OO Software Quality
-IDENTIFY FOLLOWING TO MAKE EVERY THING CORRECT ANALYSIS AND DESIGN

- Drake summarized an impressively long list of possible measures for OO software quality measurement as shown below:

1) Number of use cases and user based scenario scripts
2) Methods size
3) Number of classes
4) Methods per class
5) Public methods per class
6) Private methods per class
7) Friend methods per class
8) Friend classes per class

9) Inline methods per class
10) Virtual methods per class
11) Explicit inline methods per class
12) Comment percentages
13) Physical size
14) Lines of code
15) Effort
16) Span of reference between variables
17) Maximum nesting depth
18) Average variable name length
19) Cyclomatic complexity

20) Depth of inheritance tree
21) Number of children
22) Coupling between objects
23) Number of instances variables per class
24) Number of unique message sent
25) Number of class inherited
26) Number of class inherited from system base class
27) Estimated errors per methods
28) Number of problem reports per class
29) Number of problem report per use case
30) Reuse ratio
31) Estimated time to develop
32) Number of executable test paths per method
33) Average number of logical branch links per path for a method.

Object Oriented Metrics

- A variety of measures and metrics were developed towards the measurement of object-oriented software. Some well-known proposals are such as **Chidamber and Kemerer's metrics (CKM)** suite for object-oriented design and Henderson-Sellers' multi-dimensional framework for OO metrics. The CKM identified a part of architectural attributes of OO software, and proposed the following 6 metrics for OO software measurement:
- Weighted Method Per Class.
- Number of Children of Class.
- Depth of Inheritance Tree.
- Coupling Between Object Class.
- Response For a Class.
- Lack of Cohesion in Methods.

Weighted Method Per Class

$$\text{WMC} = \sum_{i=1}^{n} c_i$$

- c_i is the complexity (e.g., volume, cyclomatic complexity, etc.) of each method

Viewpoints

- The number of methods and complexity of methods is an indicator of ***how much time and effort is required to develop and maintain*** the object.

- The ***larger the number of methods*** *in an object, the greater the potential* ***impact on the children.***

- Objects with ***large number of methods*** are likely to be more application specific, ***limiting the possible reuse***

Number of Children

- NOC is the number of subclasses immediately subordinate to a class.

  Viewpoints
- As NOC grows, reuse increases - but the abstraction may be diluted.
- Depth is generally better than breadth in class hierarchy, since it promotes reuse of methods through inheritance.
- Classes higher up in the hierarchy should have more sub-classes then those lower down.
- NOC gives an idea of the potential influence a class has on the design: classes with large number of children may require more testing.

Depth of Inheritance Tree

- DIT is the maximum length from a node to the root (base class)

Viewpoint

- Lower level subclasses inherit a number of methods making behavior harder to predict
- Deeper trees indicate greater design complexity

Coupling Between Classes

- CBO is the number of collaborations between two classes (fan-out of a class C)
  - the number of other classes that are referenced in the class C (a reference to another class, A, is an reference to a method or a data member of class A)

Viewpoints

- As collaboration increases reuse decreases
- High fan-outs represent class coupling to other classes/objects and thus are undesirable
- High fan-ins represent good object designs and high level of reuse
- Not possible to maintain high fan-in and low fan outs across the entire system

Response for Class

- RFC is the number of methods that could be called in response to a message to a class (local + remote)

Viewpoints

As RFC increases

- testing effort increases
- greater the complexity of the object
- harder it is to understand

Unit testing

- Focus on smallest unit/element of software-modules, components, class, methods.
- Unit testing makes heavy use of white box testing.
- Individual unit tested independently
- Unit testing verify

  -internal logic of the unit
  -internal design
  -internal parts
  -error handling

- Possible units
- Packages (namespaces, assemblies)
- Types (classes, structs)
- Members of types: Procedures, functions and methods
- Single commands and expressions

Software Engineer divide the software systems into

1. Divide the systems into packages : Packages P1,P2, P3, P4....Pn
2. Divide Packages into subpackages: Each package say P1 is further divided into number of packsges P1.1 , P1.2,P1.3,.....P 1.n

3. Divide the subpackages into modules-classes

   Each package say p1.1 is divided into classes say C1, C2.. ,Cn

4. Each class has attributes and methods.
5. Test OCL Constraints of attributes and Metho
6 Test smallest unit using white box testing

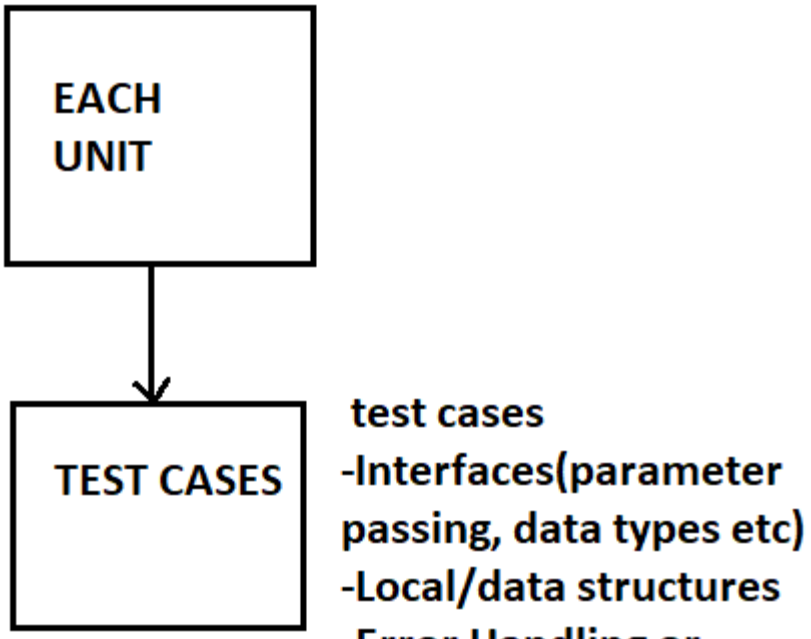
- Class Calculator

  -, +, # attributes.

  -, +, # Methods-add(), subtract(), Multiplication()

There may be many more test cases for testing a certain module. It depends on the developers' creativity because only they are responsible for designing product modules.

Unit Testing

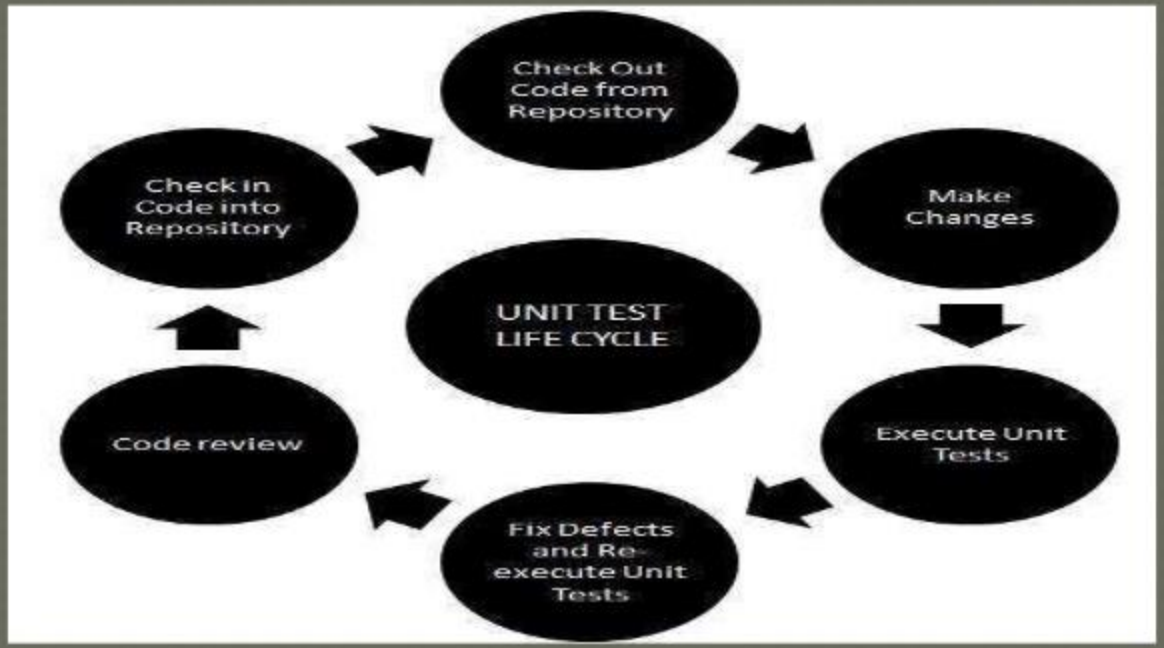


- Before proceeding website testing, a tester may focus on a certain element, for example, login page, and thereby execute unit checking.
- Usually, such components as username, password, and ‘Sign in’ button should be checked.

3 test cases for unit testing

- The ‘Sign in’ button should be active only in the case if valid values are entered in the ‘Account’ and ‘Password’ fields.
- The determined length is defined for both ‘Username’ and ‘Password’ fields.
- Only correct input values can be contained in the proper field.

Unit Testing Life Cycle :



Unit Testing Techniques:

- **Black Box Testing**
- **White Box Testing**
- **Gray Box Testing**

Gray Box Testing is a software testing technique which is a combination of Black Box Testing technique and White Box Testing technique.

In Black Box Testing technique, tester is unknown to the internal structure of the item being tested and in White Box Testing the internal structure is known to tester.

The internal structure is partially known in Gray Box Testing. This includes access to internal data structures and algorithms for purpose of designing the test cases.

Integration Testing

- After unit testing, integration Testing is done immediately. How all units work together is checked.
- Application Work Efficiently Without Any Error.

Integration Testing

OO does not have a hierarchical control structure so conventional top-down and bottom-up integration tests have little meaning

Integrating operations one at a time into a class is often impossible because of the direct and indirect interactions of the components that make up the class.

Integration applied three different incremental strategies:

- **Thread-based testing: integrates classes required to respond to one input or event**
- **Use-based testing: integrates classes required by one use case**
- **Cluster testing: integrates classes required to demonstrate one collaboration**

Types of Errors Found During Integration Testing

Messaging errors:

Failure to meet a requirement, i.e., **no method to send or receive a message**

Incompatible method and message in sender and receiver

Incorrect instantiation or destruction of objects

User interface errors:

A given sequence of user actions does not have the expected effect on the component.

Random Integration Testing

Multiple Class Random Testing

1. For each client class, use the list of class methods to generate a series of random test sequences. Methods will send messages to other server classes.
2. For each message that is generated, determine the collaborating class and the corresponding method in the server object.
3. For each method in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits
4. For each of the messages, determine the next level of methods that are invoked and incorporate these into the test sequence

Client.cpp program and Server.cpp program

The basic mechanisms of **client**-**server** setup are: A **client** app send a request to a **server** app. The **server** app returns a reply. Some of the basic data communications between **client** and **server** are: File transfer - sends name and gets a file. , socket programming, client server-handshaking communication

Cluster (Integration) Testing

A cluster is a collection of classes (possibly from different systems) cooperating with each other via messaging.

It assumes that each class has been tested individually

Cluster testing is considered a second level of integration testing

Methods for Forming Clusters

Function-based clustering

- Based on requirements and use cases
- Difficult to perform if requirements were not available during the design phase

Subject-based clustering

- Based on subject areas that need to test separately

Project Schedule-based clustering

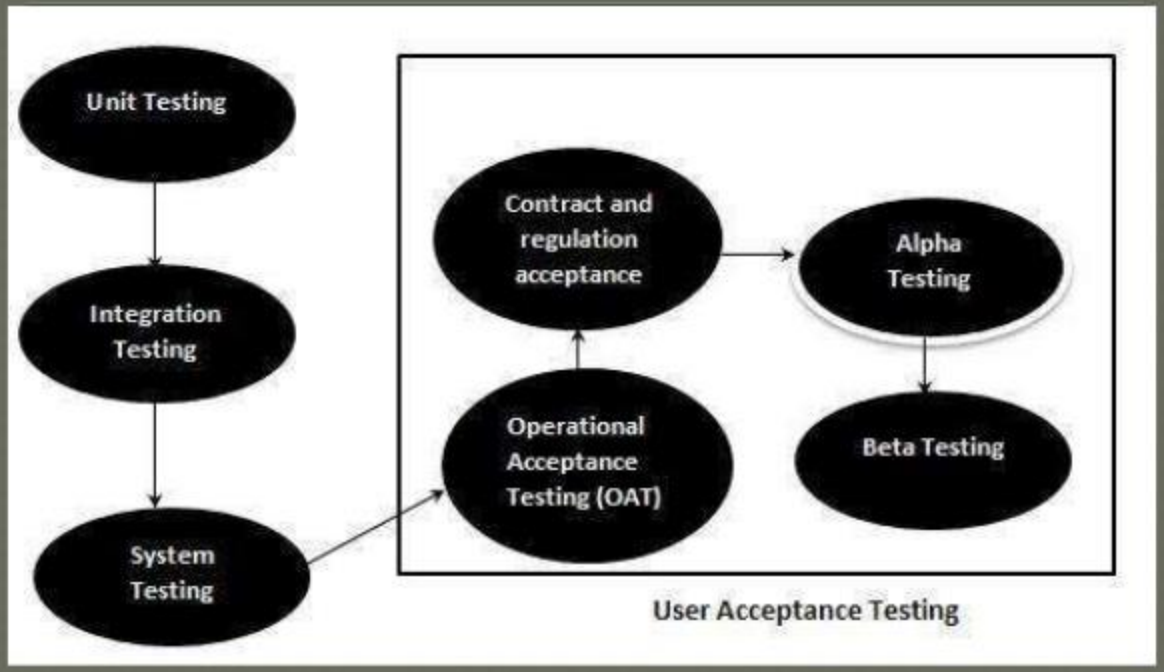
Usability Testing: User Friendly

This testing is recommended during the initial design phase of SDLC, which gives more visibility on the expectations of the users.

Usability Testing also known as User Experience(UX) Testing, is a testing method for measuring how easy and user-friendly a software application is.

A small set of target end-users, use software application to expose usability defects. Usability testing mainly focuses on user's ease of using application, flexibility of application to handle controls and ability of application to meet its objectives.

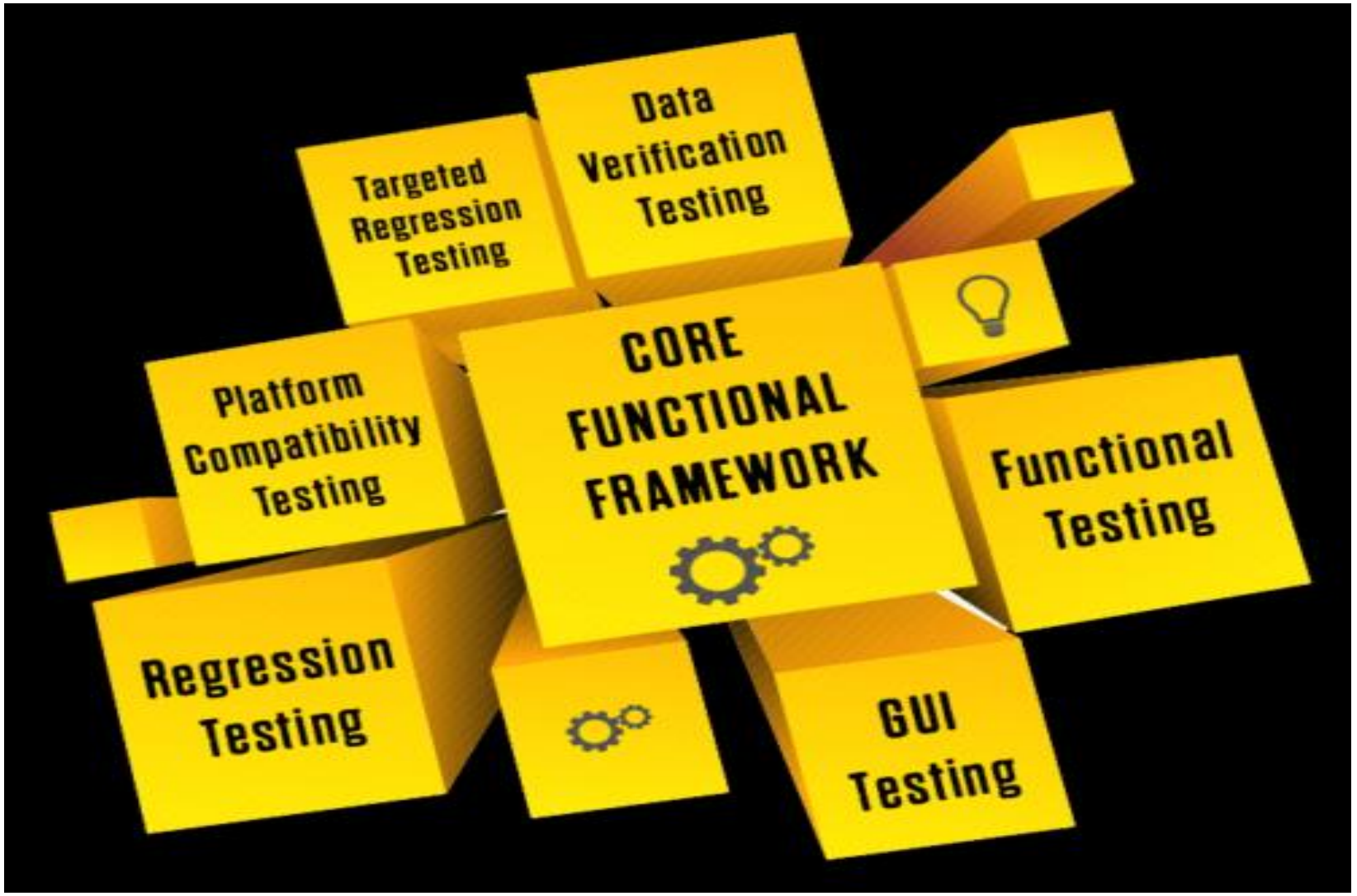
UAT - Diagram


Unit Testing
Integration Testing
System Testing
Contract and regulation acceptance
Alpha Testing
Operational Acceptance Testing (OAT)
Beta Testing
User Acceptance Testing

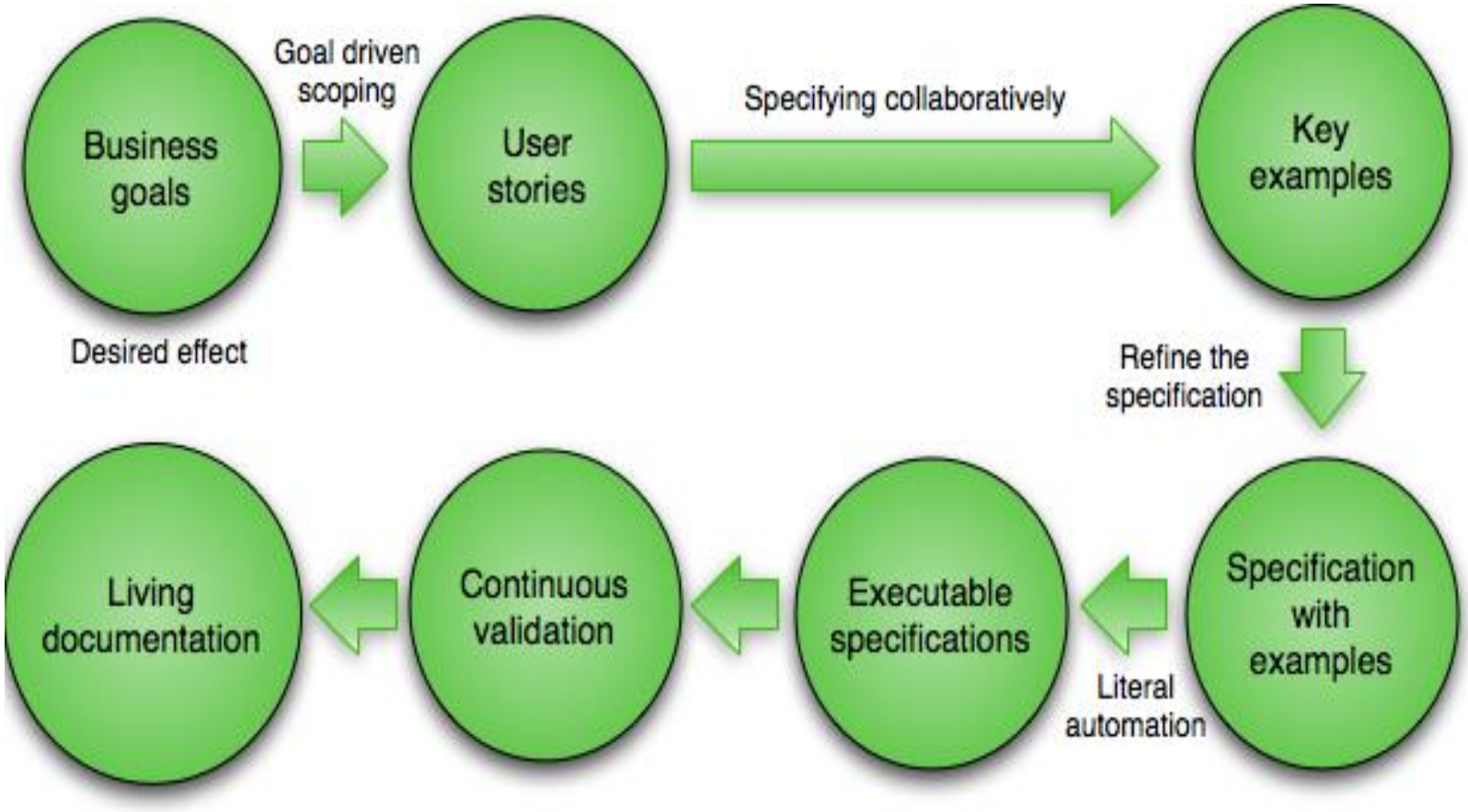
Acceptance testing types

Type of acceptance testing	Tester category	Testing Environment	Nature of developed software	Source of Requirements	Nature of Requirements
Alpha	Developer	at developer site under controlled Environment	Specific to target client	legal document (Software Requirement Specification document)	Mainly functional requirements
Beta	Experienced user	in real working Environment	Customised	No legal document	Functional and non-functional requirements
UAT	Developer and target client	Under controlled Environment	Specific Client	Business specifications	Mainly functional requirements
Contract	Developer as per contract document	Under controlled Environment	Specific Client	legal document	Functional and non-functional requirements
Operational	System administrator	Under Controlled or real Environment	Any	No legal document	Mainly non-functional requirements

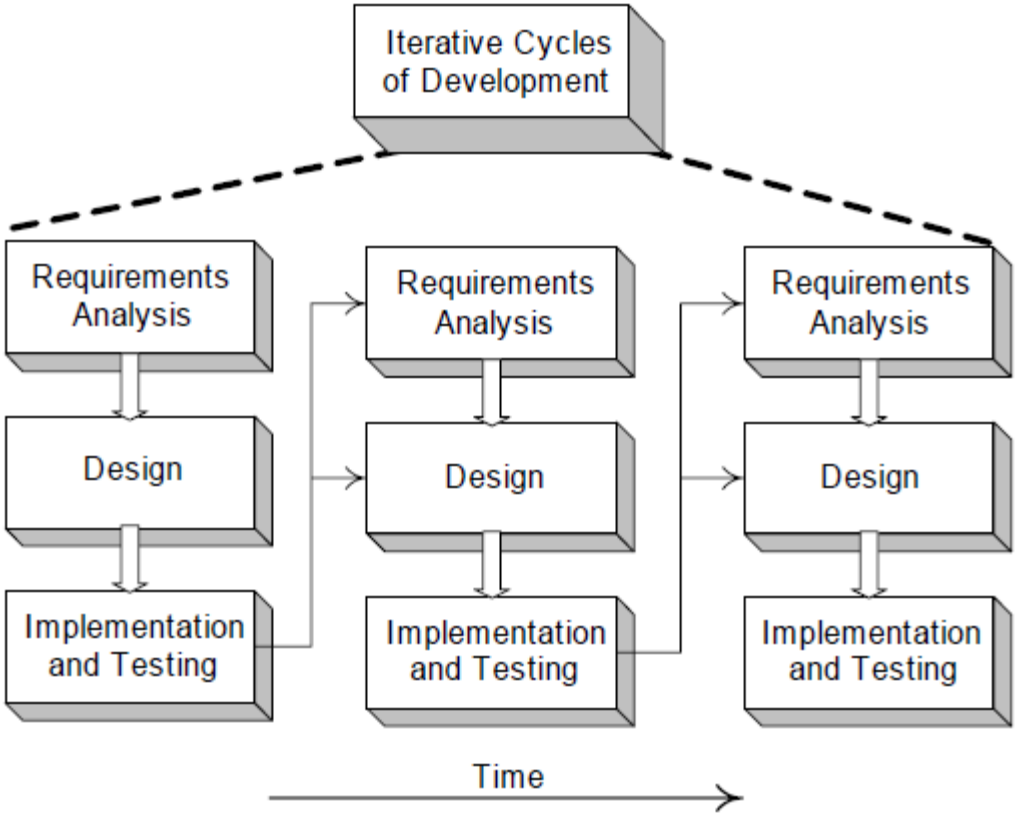
User Acceptance Testing services



Acceptance Testing Life Cycle Verification and validation



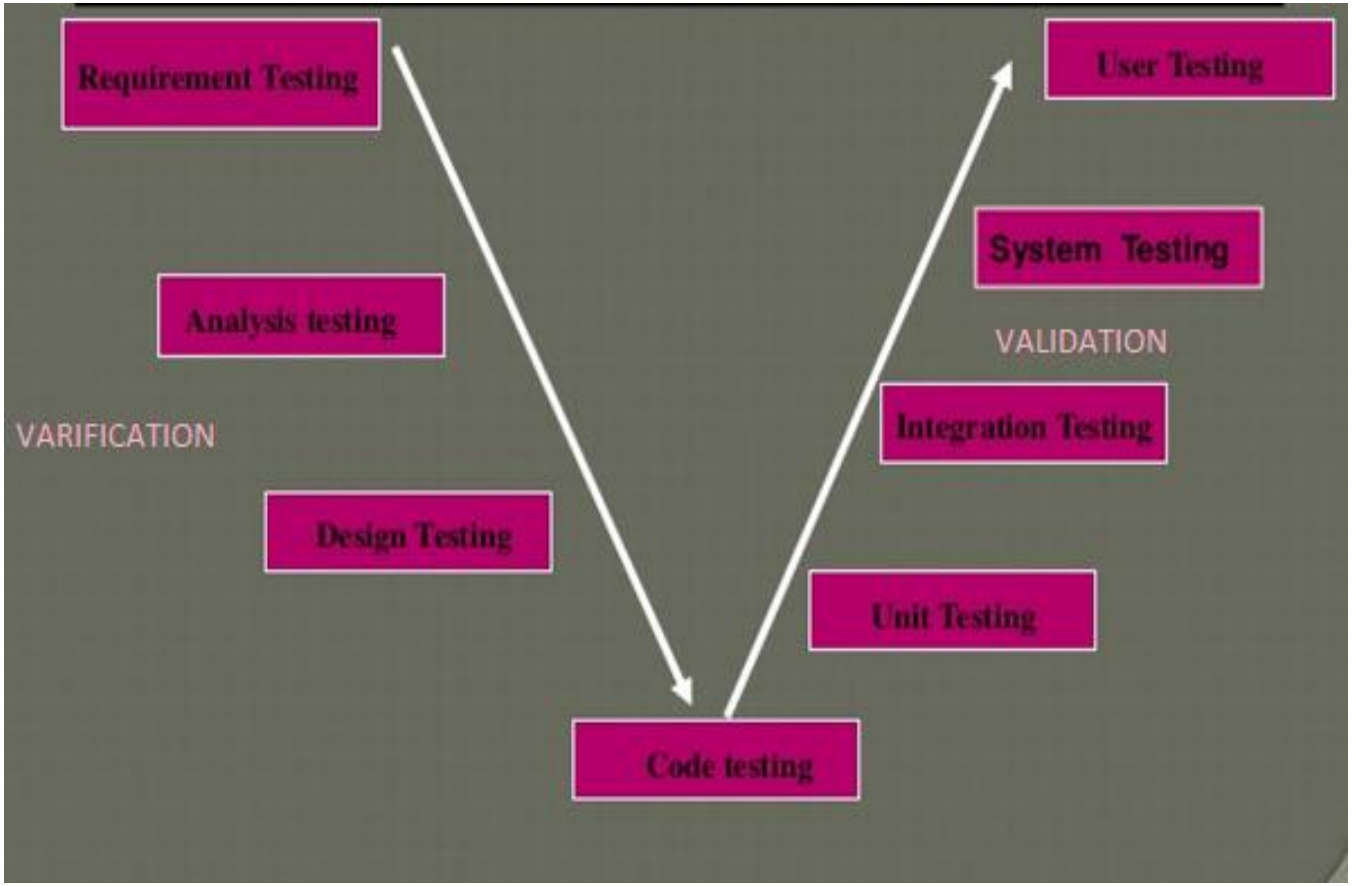
Code Changes and the Iterative Process



Implementation in an iteration influences later design.

Testing ： Verification over the time

V –MODEL for acceptance Testing



- **Class Testing:** Equivalent to unit testing for conventional SW.

– The concept of a unit changes in the OO context.
– The class or object is the smallest testable unit.
– Testing must focus on each method of the class and the state behavior of the class.
– It is important to consider the state of the class because this will effect the behavior, methods, of the class.
– It is often difficult to determine the state of the object.

- You may have to introduce new methods to look at the state variables for testing purposes.

- A bank account class.

– It may have methods to open the account, deposit funds, withdraw funds and close the account.

– The states of the account include

• open with positive balance,

• open with negative or zero balance and

• closed.

– How the methods behave depends on the state of the account.

• An account with a zero or negative balance will not allow the customer to withdraw funds.

• If positive it might allow customer to go to overdraft once.

• You could introduce a new method to determine the if the account is open or closed and if balance is positive.

- **Integration Testing:** Because OO SW does not have a hierarchical control structure, conventional top-down or bottom-up integration strategies are not applicable.

Two approaches for OO

Integration testing

- **Thread-based testing-**

– integrate the set of classes required to respond to one input or event of the system.

– Each thread is integrated and tested individually.

- **Use-based testing –**

– first construct and test the classes which use very few (if any) server classes (these are called the independent classes).

– The next step is to test the next layer of classes, the dependent classes, that use the independent classes.

- **Validation Testing**
- Conventional black-box testing derived from the analysis model can be used to test the OO software.
- Use cases are a good place to look when developing test cases.

- **Test cases and the Class hierarchy**

• Methods that are redefined in a subclass must be tested because it represents a new design and new code.

• Methods that are inherited must also be tested.

– A subset of the original tests can be executed to ensure it works in the derived class.

- **Random Testing:** When a variety of different operation sequences are randomly generated.

– Keep in mind the behavior life sequence of the class.

- **Partition testing:** Very similar to equivalence class partitioning for conventional SW. Partition the input and output of the class and design test cases to exercise each class.
- 3 Examples:
  - **State-based**
  - **Attribute-based**
  - **Category-based**

- **State-based partitioning**

– Look at the states for the class.

– Determine which operations change the state of the class and which do not and design test cases to exercise the class.

– Test each method, while object is in each state.

– Design test cases to do this.

- **Attribute-based partitioning:**

– Look at the attributes of the class.

– Partition methods into those that use the attribute, modify it and those that do not use it.

– Design test cases for each partition.

- **Category-based partitioning**

– Look at the methods for the class.

– Partition the methods into categories based on their function.

- – Examples:

• initialization operations, computational, queries and termination operations.

– Design test cases for each partition

- Student Registration Example
- A student registration class.

– The methods include adding a class, dropping a class, transferring to a different section of the class and list classes.

– A student must first be registered with the university (opened as a student).

– Holds can be placed on a student and this effects whether he/she can register for a class.

– There are limits on how many credit hours a student may register for.

– A student may graduate (close a student).

- **Random testing** may generate:

– register, add class, transfer, add class, drop class, add class, hold, release hold, add class, …

Student Registration Example

- **State-based partitioning** :

– States:

- Registered
- Hold
- Full Load
- Partial Load
- Freedom

– Events:

- Register, Add, Drop, List, transfer, Hold, Release, Graduate

– Added methods:

- Display Hold
- Display credits

- **OO Test Case Design**

• Given the differences between conventional and OO SW, test case design is slightly different.

• OO test cases should be defined in the following way:

– Each test case should be uniquely identified and associated with the class to be tested.

– The purpose of the test should be stated.

– The following list should be developed for each test:

1. List of object states to be tested.
2. List of messages and operations (methods) to be tested
3. List of exceptions that may occur as the object is tested.
4. List of external conditions (think about system testing) that may change.

• Anything else needed for the test?

- **OO Test Case Design**
- Instead of testing each component, you will
- test each class.
- Make sure you identify the class you are testing.
- You must test each method, in each state for the class.
  – This information will be included in the purpose/condition.
- If needed, add methods to check state of the object.
  – You could use the result of these methods in your expected outcome.