Module - I

Life cycle of a software system:

- Software Des
- Development
- Testing
- Deployment
- Maintenance

- Producing a software application is relatively simple in concept: ***Take an idea and turn it in to a useful progra***
- Unfortunately for projects of any real scope, there are countless ways that a simple concept can go wrong.
- Programmers may not understand what users want or n so they build the wrong application.

- The program might be full of bugs that it's frustrating t
  impossible to fix, and can't be enhanced over time.
- Software engineering includes techniques for avoiding
  many pitfalls.
- It ensures the final application is effective, usable, and
  maintainable.

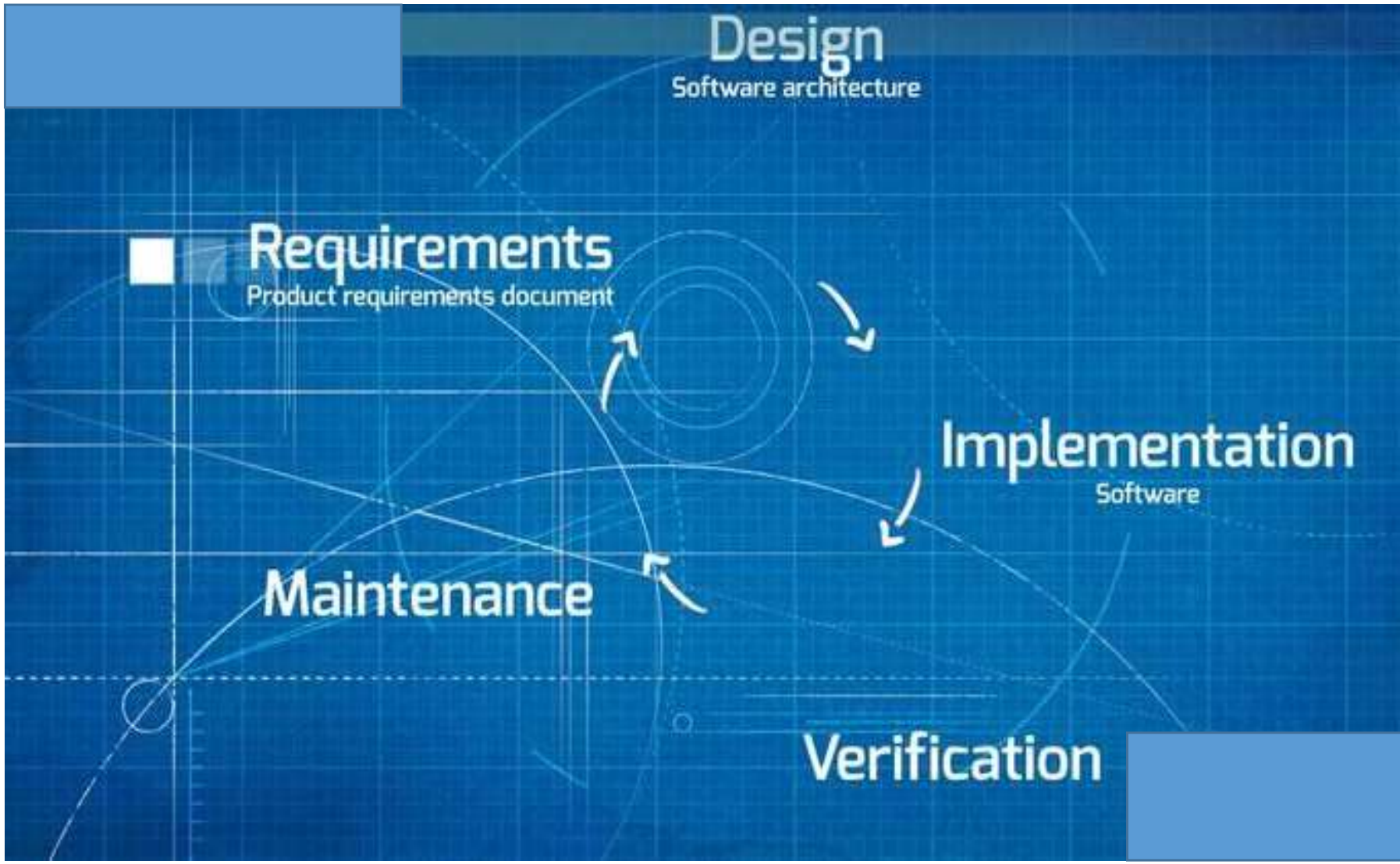
- It helps you meet mile stones on schedule and produce finished project on time and within budget.
- Perhaps most important, software engineering gives us flexibility to make changes to meet unexpected deman without completely affecting our schedule and budget constraints.

- The different steps that we need to take to keep a software engineering project on track.

- These are more or less the same for any large project although there are some important differences.

- They are:

1. Requirements Gathering (Requirements Analysis)
2. Design
3. Development (coding)
4. Testing
5. Deployment (Implementation)
6. Maintenance


Design
Software architecture
Requirements
Product requirements document
Implementation
Software
Maintenance
Verification

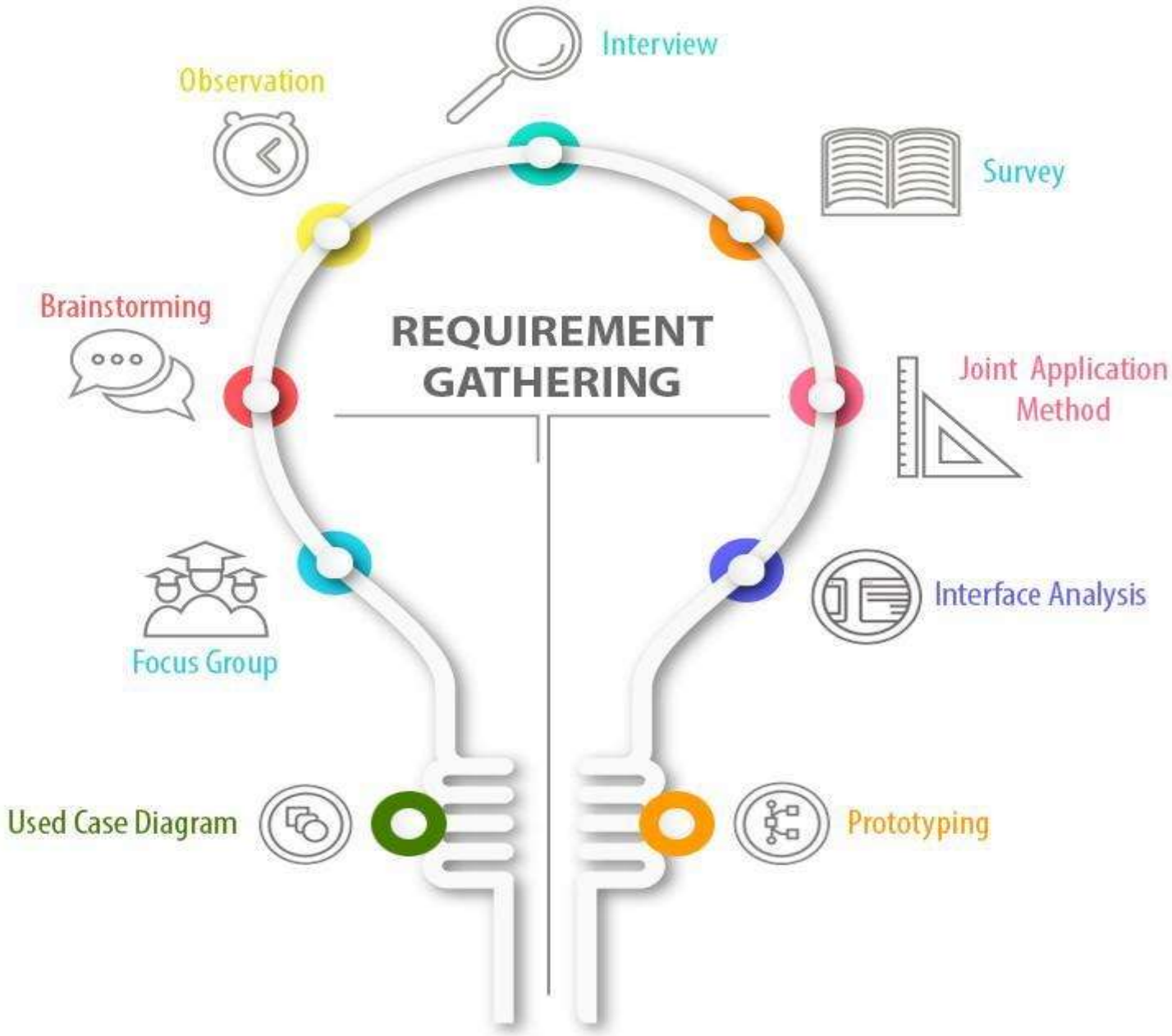
Requirements Gathering

- No big project can succeed without a plan.
- Sometimes a project doesn't follow the plan closely, but every project must have a plan.
- The plan tells project members what they should be doing, wh how long they should be doing it, and most important what th project's goals are.

- They give the project direction.
- One of the first steps in a software project is figuring out the requirements.
- We need to find out what the customers want and what the customers need.
- Depending on how well defined the user's needs are, this can time‑consuming.

- Once the customers' wants and needs are clearly specified, th
  we can turn them into requirements documents.
- Those documents tell the customers what they will be getting
  they tell the project members what they will be building.
- Throughout the project, both customers and team members c
  refer to the requirements to see if the project is heading in the
  direction.

- Requirements are the features that your application must prov
- At the beginning of the project, we gather requirements from t customers to figure out what we need to build.
- Throughout development, we use the requirements to guide development and ensure that we are heading in the right direc
- At the end of the project, we use the requirements to verify tha finished application actually does what it's supposed to do.


Interview
Observation
Survey
Brainstorming
REQUIREMENT GATHERING
Joint Application Method
Interface Analysis
Focus Group
Used Case Diagram
Prototyping

Characteristics of good requirements

- **Clear**
  - Good requirements are clear, concise, and easy to understand.
  - requirements cannot be vague or ill‑defined.
  - Each requirement must state in concrete
- **Unambiguous (**not open to more than one interpretation)
  - As we write requirements, do our best to make sure we can't think of an interpret them other than the way we intend.

- **Consistent**
  - That means not only that they cannot contradict each other, but that they a provide so many constraints that the problem is unsolvable.
  - Each requirement must also be self‐consistent.(it must be possible to achi
- **Prioritized**
  - We might like to include every feature but don't have the time or budget, something's got to go. At this point, we need to prioritize the requirement
- **Verifiable**
  - If we can't verify a requirement, how do we know whether we have met i
  - Being verifiable means the requirements must be limited and precisely de

MOSCOW METHOD - a common system for prioritizing application feat

- **M - Must.** These are required features that must be included. They are for the project to be considered a success.
- **S - Should.** These are important features that should be included if pos
- **C - Could.** These are desirable features that can be omitted if they wor schedule.
- **W - Won't.** These are completely optional features that the customers l l will not be included in the current release.

REQUIREMENT CATEGORIES

Audience-Oriented Requirements:

- These categories focus on different audiences and the different points of view that audience has.

Business Requirements:

- Business requirements layout the project's high-level goals.

User Requirements:

- User requirements (which are also called stake holder requirements), describe how will be used by the end users.

Functional Requirements:

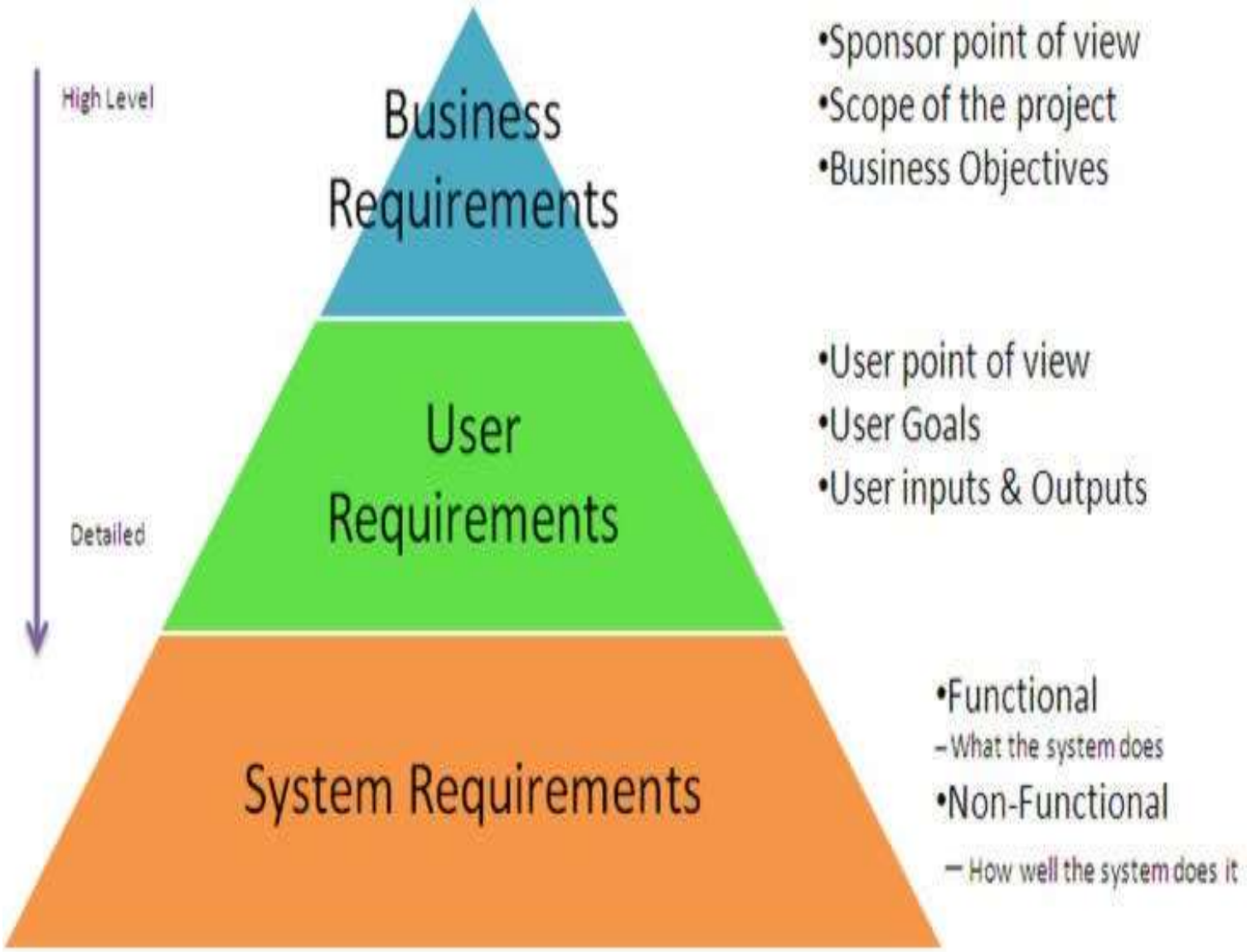
- Functional requirements are detailed statements of the project's desired capabilities

Non-functional Requirements:

- Non-functional requirements are statements about the quality of the application's be constraints on how it produces a desired result.
- They specify things such as the application's performance, reliability, and security characteristics.

Implementation Requirements:

- Implementation requirements are temporary features that are needed to transition to new system but that will be later discarded.


High Level
Detailed
Business Requirements
User Requirements
System Requirements
•Sponsor point of view
•Scope of the project
•Business Objectives
•User point of view
•User Goals
•User inputs & Outputs
•Functional
– What the system does
•Non-Functional
– How well the system does it

Systems Development Life Cycle (SDLC)
Life-Cycle Phases

Initiation

Begins when a sponsor identifies a need or an opportunity. Concept Proposal is created.

System Concept Development

Defines the scope or boundary of the concept. Includes Systems Boundary Document, Cost Benefit Analysis, Risk Management Plan and Feasibility Study.

Planning

Develops a Project Management Plan and other planning documents. Provides the basis for acquiring the resources needed to achieve a solution.

Requirements Analysis

Analyzes user needs and develops user requirements. Creates a detailed Functional Requirements Document.

Design

Transforms detailed requirements into complete, detailed System Design Document. Focuses on how to deliver the required functionality.

Development

Converts a design into a complete information system. Includes acquiring and installing systems environment; creating and testing databases; preparing test case procedures; preparing test files; coding, compiling, refining programs; performing test readiness review and procurement activities.

Integration and Test

Demonstrates that the developed system conforms to requirements as specified in the Functional Requirements Document. Conducted by Quality Assurance staff and users. Produces Test Analysis Reports.

Implementation

Includes implementation preparation, implementation of the system into a production environment, and resolution of problems identified in the Integration and Test Phase.

Operation Maintena

Describ to opera maintai informa systems in a pro environ include Implem and In-l Review

Design

- Software design sits at the technical kernel of software engineeri
- The importance of design is ***quality***
- Design is the only way that you can accurately translate stakeho requirements into a finished software product or system.
- Software design is an iterative process through which requireme translated into a “blueprint” for constructing the software

Quality Attributes

- It is commonly known as **FURPS**.
  - *Functionality*
  - *Usability*
  - *Reliability*
  - *Performance*
  - *Supportability*

- ***Functionality*** is assessed by evaluating the feature set and capabilities of the pr generality of the functions that are delivered, and the security of the overall syst
- ***Usability*** is assessed by considering human factors
- ***Reliability*** is evaluated by measuring the frequency and severity of failure, the a output results, the ability to recover from failure
- ***Performance*** is measured by considering processing speed, response time, reso consumption, throughput, and efficiency
- ***Supportability*** combines the ability to extend the program (extensibility), adapt serviceability—these three attributes represent a more common term, maintaina

- Design can be of **two types**:- high level design and low level design.

HIGH-LEVEL DESIGN

- The high-level design includes such things as decisions about what platform as desktop, laptop, tablet, or phone), what data design to use and the project at a relatively high level.
- We break the project into different modules that handle the project's major a functionality.
- We should make sure that the high-level design covers every aspect of the r
- It should specify what the pieces(modules) do and how they should interact, include as few details as possible about how the pieces do their jobs.

LOW-LEVELDESIGN

- After high-level design breaks the project into pieces, we can assign those groups within the project so that they can work on low-level designs.
- The low-level design includes information about how that piece of the pro work.
- Better interactions between the different pieces of the project that may requ here and there.

High-level design focuses on what.

Low-level design begins to focus on how.

Three characteristics for a good design

- The design must implement all of the explicit requirements contained in requirements model, and it must accommodate all of the implicit require ments desired by stakeholders.
- The design must be a readable, understandable guide for those who gen code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing data, functional, and behavioral domains from an implementation perspe

- **Modular design** involves the decomposition of software behavi encapsulated software units.
- **Modularity** is achieved by grouping together logically related elements, such as statements, procedures, variable declarations, attributes, and so on in increasingly greater levels of detail
- The main objectives in seeking modularity are to foster **high co and low coupling**

- **Cohesion** relates to the relationship of the elements of a module.
- Seven levels of cohesion in order of strength
  1. Coincidental — parts of the module are not related but are simply bundled i module.
  2. Logical — parts that perform similar tasks are put together in a module.
  3. Temporal — tasks that execute within the same time span are brought togeth
  4. Procedural — the elements of a module make up a single control sequence.
  5. Communicational — all elements of a module act on the same area of a data
  6. Sequential — the output of one part of a module serves as input for another
  7. Functional — each part of the module is necessary for the execution of a sin

- High cohesion implies that each module represents a single part of the solution. Therefore, if the system ever needs modification, then the pa needs to be modified exists in a single place, making it easier to chang
- **Coupling** relates to the relationships between the modules themselves
- There is great benefit in reducing coupling so that changes made to on do not propagate to others (that is, they are hidden)
- This principle of "information hiding," also known as Parnas partition cornerstone of all software design.
- Low coupling limits the effects of errors in a module (lower "ripple ef reduces the likelihood of data integrity problems.

- Coupling has also been characterized in increasing levels as follows:
  - **No direct coupling -** all modules are completely unrelated.
  - **Data coupling -** when two modules interact with each other by means of passing data
  - Stamp coupling - when a data structure is passed from one module to another, but that modu only some of the data elements of the structure. (When multiple modules share common dat work on different part of it.)
  - **Control coupling** - one module passes an element of control to another; that is, one module controls the logic of the other. (Two modules are called control-coupled if one of them decid function of the other module or changes its flow of execution.)
  - **Common coupling** — if two modules both have access to the same global data.
  - **Content coupling** — one module directly references the contents of another (When a modu access or modify or refer to the content of another module)

- **Parnas partitioning :-** Software partitioning into software units with low c and high cohesion can be achieved through the principle of information hid

  ✔ In this technique, a list of difficult design decisions or things that are li change is prepared.

  ✔ Code units are then designated to “hide” the eventual implementation d design decision or feature from the rest of the system.

  ✔ Thus, only the function of the code units is visible to other modules, no method of implementation.

  ✔ Changes in these code units are therefore not likely to affect the rest of

- Parnas partitioning “hides” the implementation details of software features decisions, low-level drivers, etc., in order to limit the scope of impact of fu changes or corrections.
- By partitioning things likely to change, only that module need be touched change is required without the need to modify unaffected code.
- This technique is particularly applicable and useful in embedded systems. allows easier future modification due to hardware interface changes and re amount of code affected.

Development

- After we have created the high-and low-level designs, it's time fo programmers to get to work.
- The programmers continue refining the low-level designs until th how to implement those designs in code.
- As the programmers write the code, they test it to make sure it do contain any bugs.

- During the Development Phase, the system developer takes th detailed logical information documented in the previous phase transforms it into machine-executable form, and ensures that a the individual components of the automated system/applicatio function correctly and interface properly with other componen within the system/application.
- The Development Phase includes several activities that are the responsibility of the developer.

Activities:-

- The developer places the outputs under configuration control and performs control.
- The developer also documents and resolves problems and non-conformance the software products and tasks.
- The developer selects, tailors, and uses those standards, methods, tools, and programming languages that are documented, appropriate, and established b organization for performing the activities in the Development Phase.
- Plans for conducting the activities of the Development Phase are developed documented and executed. The plans include specific standards, methods, to actions, and responsibility associated with the development and qualificatio requirements including safety and security.
- Verify that the software product covering the documented and baselined req in is a sufficient state of readiness for integration and formal testing by an a test group (i.e. other than development personnel.)
- During the Development Phase, the final Test Plan is prepared.

Programming tips

- Be alert
- Write for people not for computer
- Comment first
- Write self documenting codes
- Keep it small
- Stay focused
- Avoid side effects
- Validate results
- Use exceptions
- Don't repeat code

- The best code starts out with a good design.
- A **code smell** refers to an indicator of poor design or co
- **Refactoring** refers to a behavior-preserving code transformation enacted to improve some feature of the software, which is evidenced by the code smell.

Testing

- Effective software testing will improve software quality.
- Even poorly planned and executed testing will improve software quali finds defects.
- Testing is a life-cycle activity; testing activities begin from product in and continue through delivery of the software and into maintenance.
- Collecting bug reports and assigning them for repair is also a testing a
- But as a life-cycle activity, the most valuable testing activities occur a beginning of the project.

- Even if a particular piece of code is thoroughly tested and contains r
  few) bugs, there's no guarantee that it will work properly with the o
  parts of the system.
- One way to address the problems like this, is to perform different ki
  tests.
- First developers test their own code. Then testers who didn't write t
  test it. After a piece of code seems to work properly, it is integrated
  rest of the project, and the whole thing is tested to see if the new co
  anything.

The terms error, bug, fault, and failure:-

- Use of "**bug**" is that an error crept into the program through no action. The preferred term for an error in requirement, design, o is "**error" or "defect**."
- The manifestation of a defect during the operation of the softwa system is called a **fault**.
- A fault that causes the software system to fail to meet one of its requirements is called a **failure**.

- **Verification**, or testing, determines whether the products of a gi phase of the software development cycle fulfill the requirements established during the previous phase.
- **Verification answers the question "Am I building the produc right?"**
- **Validation** determines the correctness of the final program or so with respect to the user's needs and requirements.
- **Validation answers the question "Am I building the right pr**

Purpose of Testing

- Testing is the execution of a program or partial program with known inputs a that are both predicted and observed for the purpose of finding faults or devi the requirements.
- Testing will **flush out errors**, this is just one of its purposes.
- The other is to **increase trust in the system**.
- Testing must increase faith in the system, even though it may still contain un faults, by ensuring that the software meets its requirements.
- A **good test** is one that has a high probability of finding an error. A **successfu** that uncovers an error.

Basic principles of software testing

- These are the most helpful and practical rules for the tester.
  - All tests should be traceable to customer requirements.
  - Tests should be planned long before testing begins.
  - Remember that the Pareto principle applies to software testing.

    (The Pareto principle states that for many outcomes roughly 80% of consec
    from 20% of the causes.)
  - Testing should begin “in the small” and progress toward testing “in the
  - Exhaustive testing is not practical.
  - To be most effective, testing should be conducted by an independent pa

- Testing is a well-planned activity and should not be conducted willy nilly, nor u
  the last minute, just as the code is being integrated.
- The most important activity that the test engineer can conduct during requirem
  engineering is to ensure that each requirement is testable.
- A requirement that cannot be tested cannot be guaranteed and, therefore, must
  or eliminated.
- Wide range of testing techniques for unit testing, integration testing, and syster
  testing.
- Any one of these test techniques can be either insufficient or computationally u
  Therefore, some combination of testing techniques is almost always employed.

Who Tests the Software?

developer

Understands the system

but, will test "gently"

and, is driven by "delivery"

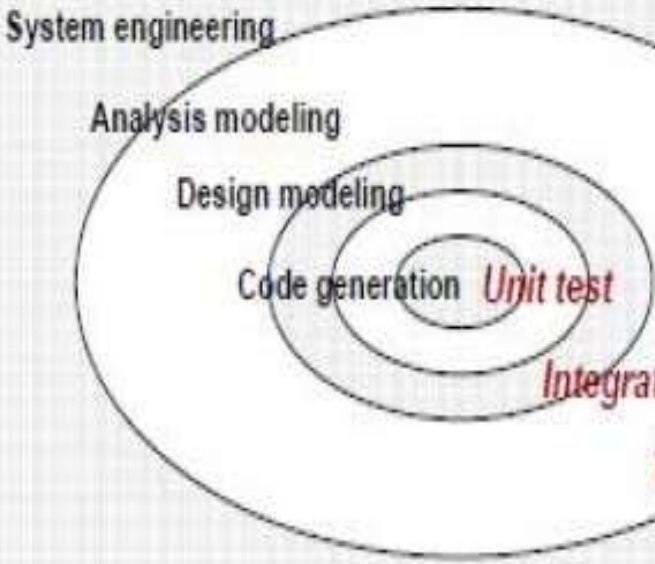
independent tester

Must learn about the system,

but, will attempt to break it

and, is driven by quality

Testing Strategy



Levels of Testing (Types of Testing)

- **Unit testing:-** Verifies the correctness of a specific piece of code
  - Since unit test is the first chance to catch errors, it is extremely
  - Focus on each unit (eg: component, class or web App content c

    the software as implemented in the source code.
  - sometimes called **desk checking.**

Black Box Testing

- In black box testing, only inputs and outputs of the unit are consider the outputs are generated based on a particular set of inputs is ignor
- Some widely used black box testing techniques include:
  - Exhaustive testing
  - Boundary value testing
  - Random test generation
  - Worst case testing

- **Brute force or exhaustive testing** involves presenting each code un every possible input combination.
  - Brute force testing can work well in the case of a small number o each with a limited input range
- **Boundary value or corner case testing** solves the problem of comb explosion by testing some very tiny subset of the input combinations as meaningful “boundaries” of input.

- **Random test case generation, or statistically based testing**, can be

  both unit and system level testing.

  - This kind of testing involves subjecting the code unit to many ran

    generated test cases over some period of time.

  - The purpose of this approach is to simulate execution of the softw

    under realistic conditions.

- **Equivalence class testing** involves partitioning the space of pos

  inputs to a code unit or group of code units into a set of represen

  inputs.

- **Example:-**

  Auto manufacturers don’t have a crash test dummy representing every po

  human being. Instead, they use a handful of representative dummies — s

  average, and large adult males; small, average, and large adult females; p

  female; toddler, etc. These categories represent the equivalence classes.

Disadvantages to black box testing :-

- One disadvantage is that it can bypass unreachable or d code.
- In other words, black box testing only tests what is expe happen, not what wasn't intended.
- **White box or clear box testing** techniques can be used with this problem.

White Box Testing

- White box testing (sometimes called **clear or glass box testing**) test the structure of the underlying code. For this reason it is also **structural testing**.
- Black box tests are data driven, White box tests are logic driven.
- White box testing are designed to exercise all paths in the code u
- White box testing also has the advantage that it can discover thos paths that cannot be executed.

- The following white box testing strategies
  - ✔ DD path testing
  - ✔ DU path testing
  - ✔ McCabe's basis path method
  - ✔ Code inspections
  - ✔ Formal program proving

- **DD path testing, or decision-to-decision path testing** is b
  on the control structure of the program.
- In DD testing, a graph representation of the control structu
  the program is used to generate test cases that traverse the g
  essentially from one decision branch (for example, if-then
  statement) to another in well-defined ways.
- Depending on the strength of the testing, different combina
  of paths are tested.

- **DU (define-use) path testing** is a data-driven white box tes technique that involves the construction of test cases that ex all possible definition, change, or use of a variable through software system.
- DU path testing is actually rather sophisticated because ther hierarchy of paths involving whether the variable is observe changed.

- **McCabe's basis path method -** McCabe's metric to deter
  the complexity of code.
- McCabe's metric can also be used to determine a lower bo
  the number of test cases needed to traverse all linearly
  independent paths in a unit of code.

- McCabe also provides a procedure for determining the linear independent paths by traversing the program graph.
- This technique is called the **basis path method**
- The basis path method begins with the selection of a baseline which should correspond to some “ordinary” case of program execution along one of the programs.
- McCabe advises choosing a path with as many decision node possible.

- **Code Inspections:-** In code inspections, the author of some colle software presents each line of code to a review group of peer sof engineers.
- Code inspections can detect errors as well as discover ways for i the implementation.
- This audit also provides an opportunity to enforce coding standa

- **Formal Program Proving :-** Formal program proving is a kind box testing using mathematical techniques in which the code is a theorem and some form of calculus is used to prove that the pr correct.
- This form of verification requires a high level of training and is generally, for only limited purposes because of the intensity of a required.

Integration Testing:-

- Integration testing involves testing of groups of components int
  to create a system or sub-system.
- The tests are derived from the system specification.
- Verifies that a block of code is working properly
- It checks that the existing code calls the new method correctly, a
  new method can call other methods correctly.

- **Incremental Integration Testing:-** This is a strategy that partitio system in some way to reduce the code tested. Incremental testin includes:
  - **Top-Down Testing**
  - **Bottom-Up Testing**
  - **Other kinds of system partitioning**

In practice, most integration involves a combination of these st

- **Top-Down Testing -** This kind of testing starts with a high-level s integrates from the topdown, replacing individual components by (dummy programs) where appropriate.
- **Bottom-Up Testing -** Bottom up testing is the reverse of top-dow which we integrate individual components in levels, from the bott until the complete system is created.
- **Other kinds of system partitioning -** These include pair-wise int sandwich integration, neighbourhood,integration testing, and inter testing.

Other kinds of system partitioning

- **Pair-wise integration testing -** pairs of modules or functionality together.
- **Sandwich integration -** combination of top-down and bottom up which falls somewhere in between big-bang testing (one big test testing of individual modules.
- **Neighborhood integration testing -** portions of program functio are logically connected somehow are tested in groups.
- **Interface testing** - takes place when modules or subsystems are to create larger systems.

- **Component interface testing:-** Studies the interactions between co

  Common strategy for component level testing is to think of the int

  between components as one component sending a message to ano

- **Stress testing:** executes a system in a manner that demands resourc

  abnormal quantity, frequency, or volume.

  - the system is subjected to a large disturbance in the inputs;
  - to see how the system fails (gracefully or catastrophically).
  - with cases and conditions where the system is under heavy load

- **Burn-in testing** is a type of system-level testing done in the fa

  which seeks to flush out those failures appearing early in the life

  system, and thus to improve the reliability of the delivered produ

- **Cleanroom testing** is more than a kind of system testing. It is r

  testing philosophy.

  - The principal tenant of cleanroom software development is that **given su**

    time and care, error-free software can be written.

  - Cleanroom software development relies heavily on code inspections and

    program validation.

- **Regression testing:-** Test the program's entire functionality to thing changed when you added new code to the project.
  - Regression testing (which can also be performed at the unit le used to validate the updated software against the old set of te that have already been passed.
  - Regression testing helps to ensure that changes (due to testing other reasons) do not introduce unintended behavior or additi errors.

- **Acceptance testing:-** Determine whether the finished applica

  meets the customer's requirements.

  - Usually customer or representative sits down with the applica

    runs through all the user cases (during requirement analysis)

    sure everything works as advertised.

  - Most software builders use a process called alpha and beta te

    uncover errors that only the end user able to find.

Alpha test:-

- First round testing by selected customers or independent
- version of the complete software is tested by customer un supervision of the developer at the developer's site

Beta test:-

- Second round testing after alpha test.
- version of the complete software is tested by customer at her own site without the developer being present.

- **Software Fault Injection :-** Fault injection is a form of c
  software testing that acts like “crashtesting” the software
  demonstrating the consequences of incorrect code or data
  - The main benefit of fault injection testing is that it can demon
    the software is unlikely to do what it shouldn’t.
  - Eg: type a letter when the input called is for a number
- **Security testing:** verifies that protection mechanisms bu
  system will, in fact, protect it from improper penetration.

- **Automated testing:-** Automated testing tools let you define and the results they should produce.
  - After running a test, the testing tool can compare the resu with expected results.
- **System testing:-** End to end run through of the whole syste
  - System test exercise every part of the system to discover bugs as possible.

- **Performance Testing:** test the run-time performance of so within the context of an integrated system
- **Deployment testing(configuration testing):** exercises the software in each environment in which it is to operate. It a examines the installation procedures and installation softw
- **Recovery testing:** forces the software to fail in a variety o and verifies that recovery is properly performed.

- **Compatibility test:-** Focus on compatibility with different environments such as computers running older operating sys versions. Also with older versions of files, databases etc…
- **Destructive test :-** Makes the application fail so that you ca its behaviour when the worst happens.
- **Installation test:-** Make sure that you can successfully insta system on a fresh computer

- **Functional test:-** Deals with features the application provides
- **Exhaustive testing :-** Testing a method with every possible in
  - It proves that a method works correctly under all circumsta it's the best you can possibly do.
  - Because most methods take too many possible inputs, it wo most of the time.
- **Accessibility test:-** Tests the application for accessibility by th visual, hearing or other impairments.

When should you stop testing?

- There are several criteria that can be used to determine when testing sh

1. When you run out of time.
2. When continued testing causes no new failures.
3. When continued testing reveals no new faults.
4. When you can't think of any new test cases.
5. When you reach a point of "diminishing returns."
6. When mandated coverage has been attained.
7. When all faults have been removed

Deployment

- What is deployment?
  - Getting software out of the hands of the developers in
    hands of the users.
- The software is delivered to the customer who evaluates
  delivered product and provides feedback based on the
  evaluation.

Key issues around deployment

1. **Business processes:-** Most large software systems require the custome change the way they work.
2. **Training:-** No point in deploying software if the customers can't use it
3. **Deployment itself:-** How physically to get the software installed.
4. **Equipment:-** Is the customer's hardware up to the job?
5. **Expertise:-** Does the customer have the IT expertise to install the softv
6. **Integration:-** can it be integrated with other systems of the customer.

Maintenance

- The software product will *enter a maintenance mode after delive* which it will experience many recurring life cycles as errors are and corrected and features are added.
- The process of changing a system after it has been delivered.
- Software maintenance is the "…correction of errors, and implem of modifications needed to allow an existing system to perform n tasks, and to perform old ones under new conditions"

- As soon as the users start pounding away on our software, they w bugs.
- Of course, when the users find bugs, we need to fix them.
- Fixing a bug sometimes leads to another bug, so now we get to fi as well.
- The software maintenance phase activities generally consist of a reengineering processes to prolong the life of the system.

- **Re-engineering:-** is the process of taking an old or un-maintai system and transforming it until it's maintainable.
- If our application is successful, users will use it a lot, and they even more likely to find bugs. They also think up of enhancem improvements, and new features that they want added immedi

- Generally maintenance tasks are grouped into the following four categories: (3 types of maintenance)

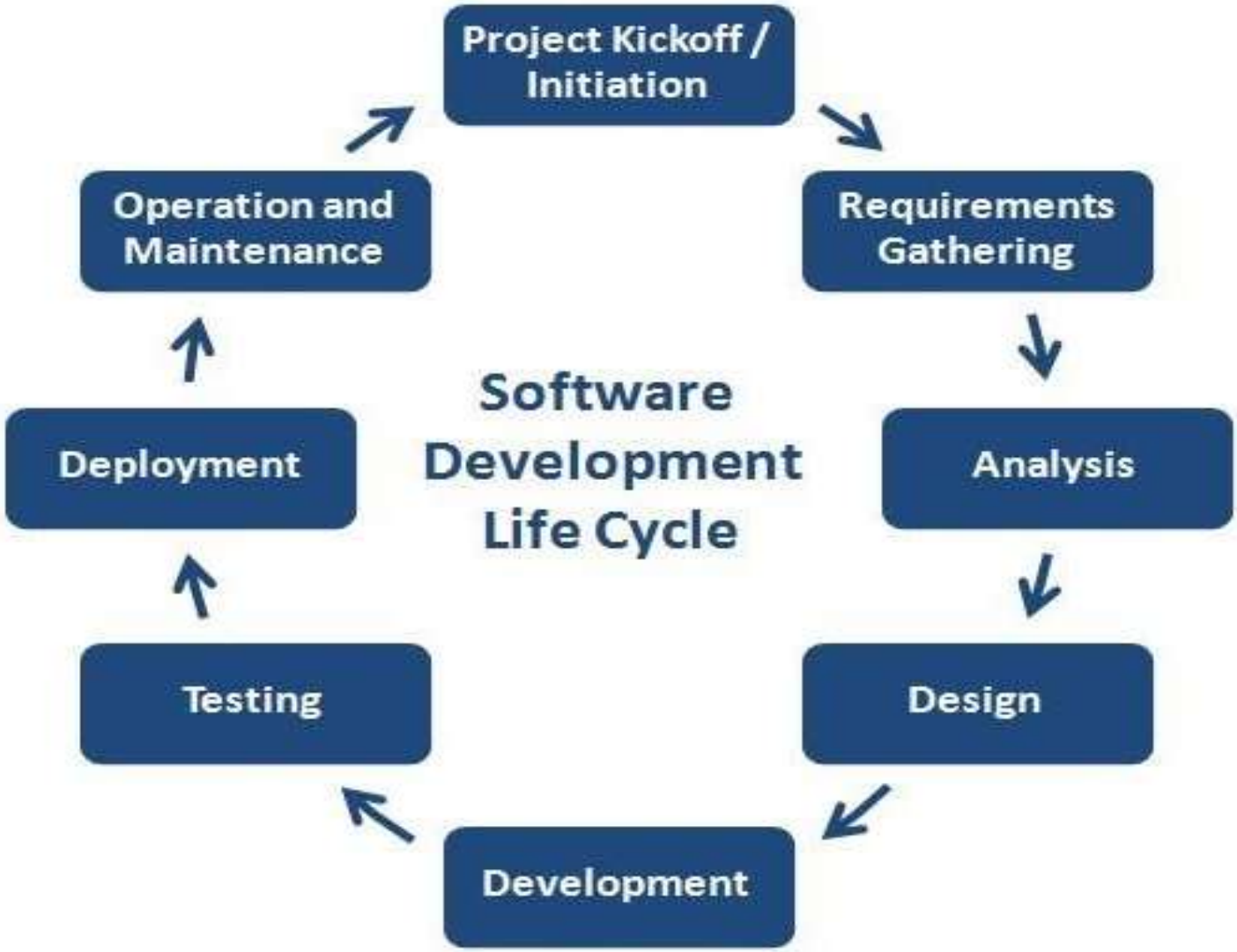
➢ **Corrective—** changes that involve maintenance to correct erro (Fixing bugs)

➢ **Perfective —** all other maintenance including enhancements, documentation changes, efficiency improvements, and so on. (Imp existing features and adding new ones)

➢**Adaptive—** changes that result from external changes to the system must respond. (Modifying the application to mee changes in the application's environment)

➢ **Preventive—** Restructuring the code to make it more maintainable.

- **Fixing bugs and vulnerabilities:-** not only in code, but also and requirements
- **Adapting to new platforms and software environments:-** new hardware, new OS, new support software
- **Supporting new features and requirements:-** necessary as operating environments change and in response to competitive pressures


Project Kickoff / Initiation
Requirements Gathering
Analysis
Design
Development
Testing
Deployment
Operation and Maintenance
Software Development Life Cycle