Advanced Software Engineering
20MCA107

Balachandran K P

kpbala@gmail.com

9037223801

Course Outcomes:

- Get a full view of the Software life cycle
- Gain a deep knowledge of Software Planning, Analysis and Design and Software Engineering Models
- Have a great comprehension of Coding Practices, Version Control using ‘git’ and Software Quality
- Acquire ample grasp of Design Patterns
- Get deeply familiarised with Software Testing and its automation
- Start using Agile Methodology
- Begin to apply CI/CD techniques in Software development

Course Outcomes:

- Understand the software development as an engineering process and its stages.
- Understand Software development lifecycle (SDLC).
- Understand software engineering models.
- Learn how to prepare software requirements specification, approaches and methodologies to prepare requirement specifications document
- Understand writing industry-grade software programs, following style guides and coding standards.
- Learn core concepts of software version control system and common operations with Git distributed version control system.
- Understanding software quality concepts with respect to software requirement specifications document, what to conform to at various stages of SDLC.
- Understand what to ensure at various stage of SDLC to ensure quality of developed software system.

Course Outcomes:

- Learn Object Oriented Programming concepts comprehensively.
- Learn the concept of Design Patterns, category of patterns, and how to select appropriate design patterns.
- Understand Unit testing concepts and xUnit architecture.
- Learn Unit testing frameworks and writing unit testing for Java and one of PHP or Python.
- Understand the concepts Continuous Integration and Continuous Delivery (CICD).
- Knowledge of Git distributed version control system to use in a product environment.
- Knowledge of OOP paradigm and software Design Patterns to design the software system.
- Knowledge of unit testing frameworks such as Junit, uniitest, phpdbg for wiring units tests in a software production environment.
- Knowledge of software testing CI/CD practices.

Course Outcomes:

- Understand software testing concepts and principles.
- Learn common approaches to ensure software quality through testing.
- In-depth understanding of various types of testing methodologies
- Learn about testing automation and understand commonly used test automation types.
- Learn to use Robot framework
- Understand the concepts of Agile methodology.
- Learn to use Scrum framework for implementing Agile methodology for executing a software development process.
- Learn to monitor a software development project using a Scrum tool.
- Understand the concepts of Software Configuration Management.
- Learn about build and deployment environments.
- Understand the concepts of Continuous Integration and essential practices.
- Understand the concepts of deployment automation and learn to use Ansible.

Syllabus

Module 1 [8 hrs]

Introduction to Software Engineering: What is Software Engineering, Characteristics of Software. Life cycle of a software system: software design, development, testing, deployment, Maintenance.

Project planning phase: project objectives, scope of the software system, empirical estimation models, COCOMO, staffing and personnel planning.

Software Engineering models: Predictive software engineering models, model approaches, prerequisites, predictive and adaptive waterfall, waterfall with feedback (Sashimi), incremental waterfall, V model; Prototyping and prototyping models.

Software requirements specification, Eliciting Software requirements, Requirement specifications,

Software requirements engineering concepts, Requirements modelling, Requirements

documentation. Use cases and User stories.

Syllabus

Module 2 [10 hrs]

Programming Style Guides and Coding Standards; Literate programming and Software
documentation; Documentation generators, Javadoc, phpDocumentor.
Version control systems basic concepts; Concept of Distributed version control system and Git;
Setting up Git; Core operations in Git version control system using command line interface (CLI):
Clone a repository; View history; Modifying files; Branching; Push changes, Clone operation, add, commit, log, diff commands, conflict resolution. Pushing changes to the master; Using Git in IDEs and UI based tools.

Software Quality: Understanding and ensuring requirements specification quality, design quality, quality in software development, conformance quality.

Syllabus

Module 3 [10 hrs]

OOP Concepts; Design Patterns: Basic concepts of Design patterns, How to select a design pattern, Creational patterns, Structural patterns, Behavioural patterns. Concept of Anti-patterns.

Unit testing and Unit Testing frameworks, The xUnit Architecture, Writing Unit Tests using at least one of Junit (for Java), unittest (for Python) or phpdbg (PHP).

Writing tests with Assertions, defining and using Custom Assertions, single condition tests, testing for expected errors, Abstract test.

Syllabus

Module 4 [10 hrs]

Concepts of Agile Development methodology; Scrum Framework.

Software testing principles, Program inspections, Program walkthroughs, Program reviews;

Blackbox testing: Equivalence class testing, Boundary value testing, Decision table testing, Pairwise testing, State transition testing, Use-case testing; White box testing: control flow testing, Data flow testing.

Testing automation: Defect life cycle; Regression testing, Testing automation; Testing non-functional requirements.

Syllabus

Module 5[10 hrs]

Software Configuration Management: Using version control, Managing dependencies, Managing software configuration, Managing build and deployment environments.

Continuous Integration: Prerequisites for continuous integration, Essential practices.

Continuous Delivery: Principles of Software delivery, Introduction and concepts. Build and deployment automation, Learn to use Ansible for configuration management.

Test automation (as part of continuous integration), Learn to set up test automation cases using Robot Framework.

Software

- A set of meaningful programs organized in a logical order for a specific set of objectives and associated documentations

Engineering

- Applying Procedures, Methods and Tools for building something.
- Makes things measurable

Process and Product

- Process is the step by step procedure to produce a Product
- Product is the end result of a process
- Process and Product are mutually dependent. Without one the other can not exists

Software products

- Generic products
  - Stand-alone systems that are marketed and sold to **any customer** who wishes to buy them.
  - Examples – PC software such as editing, graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.
- Customized products (Bespoke)
  - Software that is commissioned by **a specific customer** to meet their own needs.
  - Examples – embedded control systems, air traffic control software, traffic monitoring systems.

Software—New Categories

- Open world computing—pervasive, ubiquitous, distributed computing due to wireless networking. How to allow mobile devices, personal computer, enterprise system to **communicate across vast network**.
- Netsourcing—the Web as a computing engine. How to architect simple and sophisticated applications to target end-users worldwide.
- Open source—"free" source code open to the computing community (a blessing, but also a potential curse!)
- Also
  - Data mining
  - Grid computing
  - Cognitive machines
  - Software for nanotechnologies

What is Software Engineering?

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- That is from the early stages of system specification to maintaining the system after it has gone into use.
- Software engineering is the establishment and use of engineering principles in order to obtain economically feasible software that is reliable and works efficiently on real machines.

Software Engineering Definition

The seminal definition:

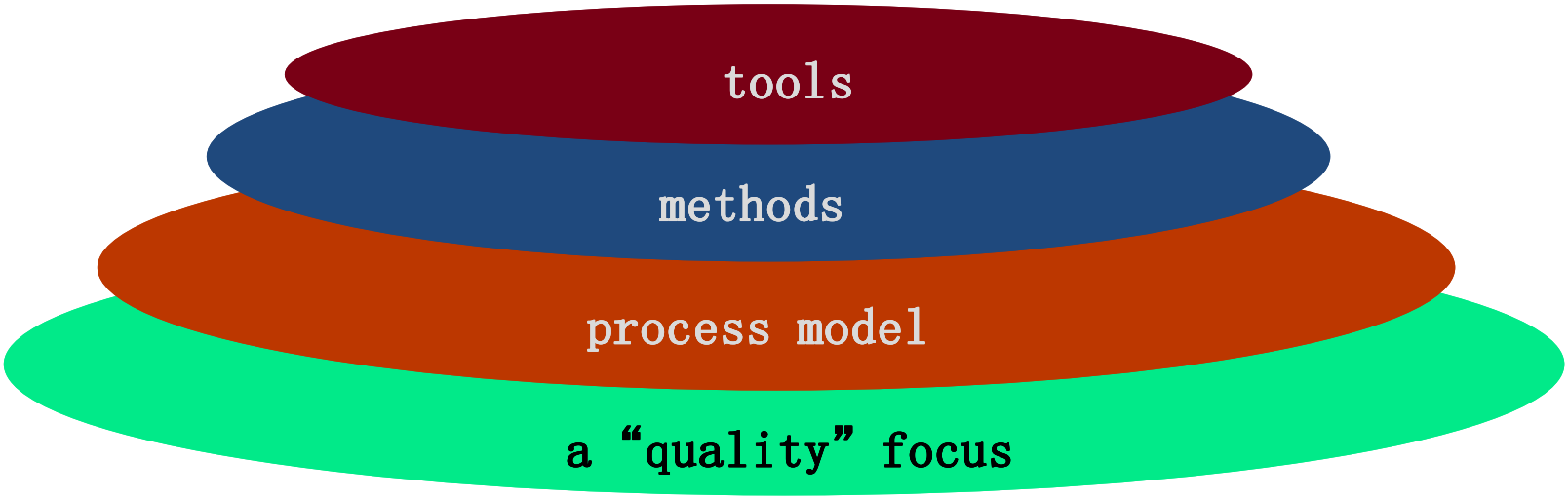
[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

The IEEE definition:

Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

Software engineering is about managing all the sources of complexity to produce effective software

A Layered Technology



- **Process** layer as the foundation defines a framework with activities for effective delivery of software engineering technology. Establish the context where products (model, data, report, and forms) are produced, milestone are established, quality is ensured and change is managed.
- **Method** provides technical how-to’s for building software. It encompasses many tasks including communication, requirement analysis, design modeling, program construction, testing and support.
- **Tools** provide automated or semi-automated support for the process and methods.

Components of Software Engineering

- Partition the system into different components
- Understand the relationship between various components
- Define relationship in terms of inputs, outputs and processes
- Understand the role of hardware and software
- Identify the key operational and functional requirements
- Model the system for analysis and development
- Discuss the system with the customer

Importance of Software Engineering

- More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.

- It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of system, the majority of costs are the costs of changing the software after it has gone into use.

Why Software Engineering?

- **Major Goals:**

- To increase software **productivity** and **quality**.
- To effectively control software **schedule** and planning.
- To reduce the **cost** of software development.
- To meet the **customers**' needs and requirements.
- To enhance the conduction of software engineering **process**.
- To improve the current **software engineering practice**.
- To support the engineers' activities in a systematic and efficient manner.

Why Learn Software Engineering ?

- Software engineers play an important role in today's information driven society.
- Software engineering is ***one of the fastest growing professions with excellent job prospects*** predicted throughout the coming decade.
- Software engineering brings together various skills and responsibilities.

- Studying software engineering opens up a wide range of career opportunities like Software Developer, Test Engineer, Project Manager, Software Architect, Software Quality Manager, Doctoral Student/Scientist

Characteristics of Software

- Software can be characterized by any number of qualities.
- **External qualities**, such as usability and reliability, are visible to the user.
- **Internal qualities** are those that may not be necessarily visible to the user, but help the developers to achieve improvement in external qualities.
- For example, good requirements and design documentation might not be seen by the typical user, but these are necessary to achieve improvement in most of the external qualities.

The most commonly discussed qualities of software are:

- **Reliability**
- **Correctness**
- **Performance**
- **Usability**
- **Interoperability**
- **Maintainability**
- **Evolvability**
- **Repairability**
- **Portability**
- **Verifiability**
- **Traceability**
- **And few negative qualities like** Fragility, Immobility, Needless complexity, Needless repetition, Opacity, Rigidity, Viscosity **that should be avoided**

Reliability (1/4)

- Software reliability can be defined informally in a number of ways.
- For example, can the user "**depend on**" the software?
- Other characterizations of a reliable software system include:
  - The system "stands the test of time."
  - There is an absence of known catastrophic errors (those that disable or destroy the system).
  - The system recovers "gracefully" from errors.
  - The software is robust.
  - Downtime is below a certain threshold. (**time during which a machine, especially a computer, is out of action or unavailable for use.**)
  - The accuracy of the system is within a certain tolerance.
  - Real-time performance requirements are met consistently

How do you measure software reliability? (2/4)

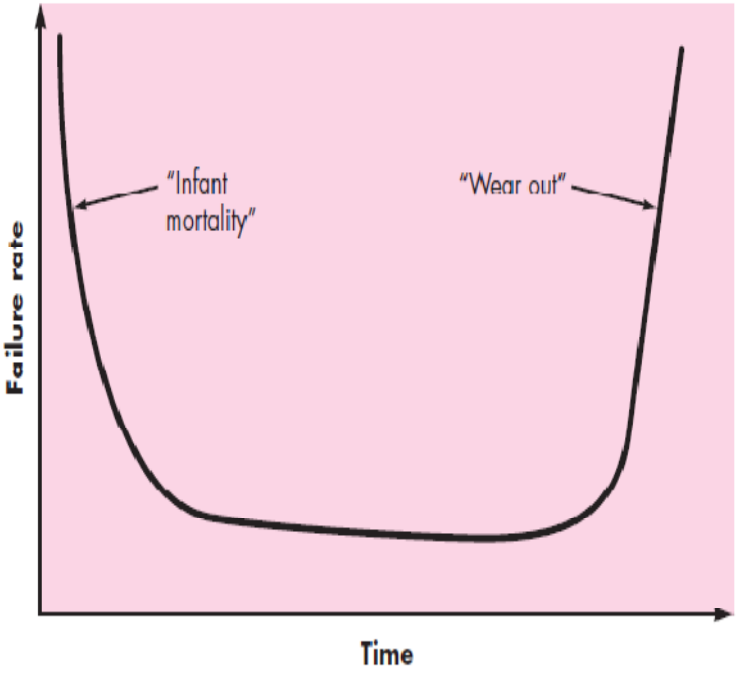
- Software reliability can be defined in terms of statistical behavior;
- that is, the probability that the software will operate as expected over a specified time interval.
- Let *S* be a software system and let *T* be the time of system failure. Then the reliability of *S* at time *t*, denoted *r(t),* is the probability that *T* is greater than *t* ; that is, **r(t) =P(T > t)**
- This is the probability that a software system will operate without failure for a specified period.
- Thus, a system with reliability function *r(t)* = 1 will never fail22

How do you measure software reliability? (3/4)

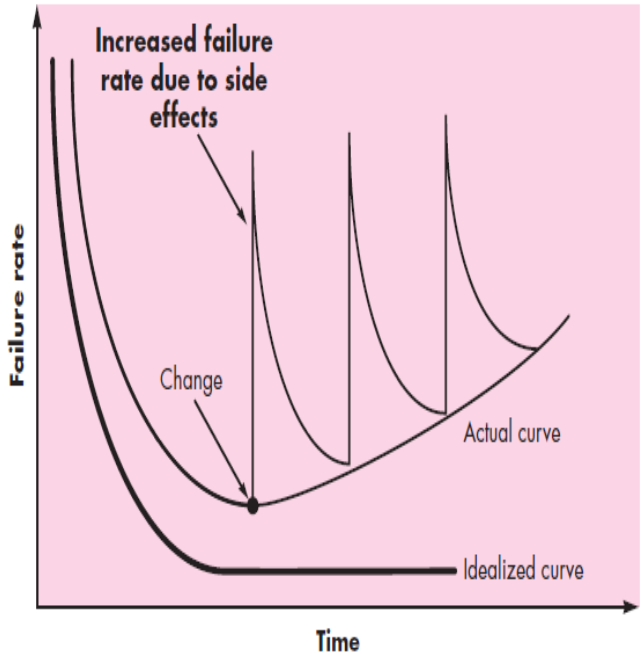
- The failure intensity is initially high, as would be expected in new software as faults are detected during testing.
- The number of failures would be expected to decrease with time, presumably as failures are uncovered and repaired.
- The **bathtub curve** is often used to explain the failure function for physical components that wear out, electronics, and even biological systems.
- We expect a large number of failures early in the life of a product (from manufacturing defects) and then a steady decline in failure incidents until later in the life of that product when it has "worn out" or, in the case of biological entities, died.

How do you measure software reliabilit

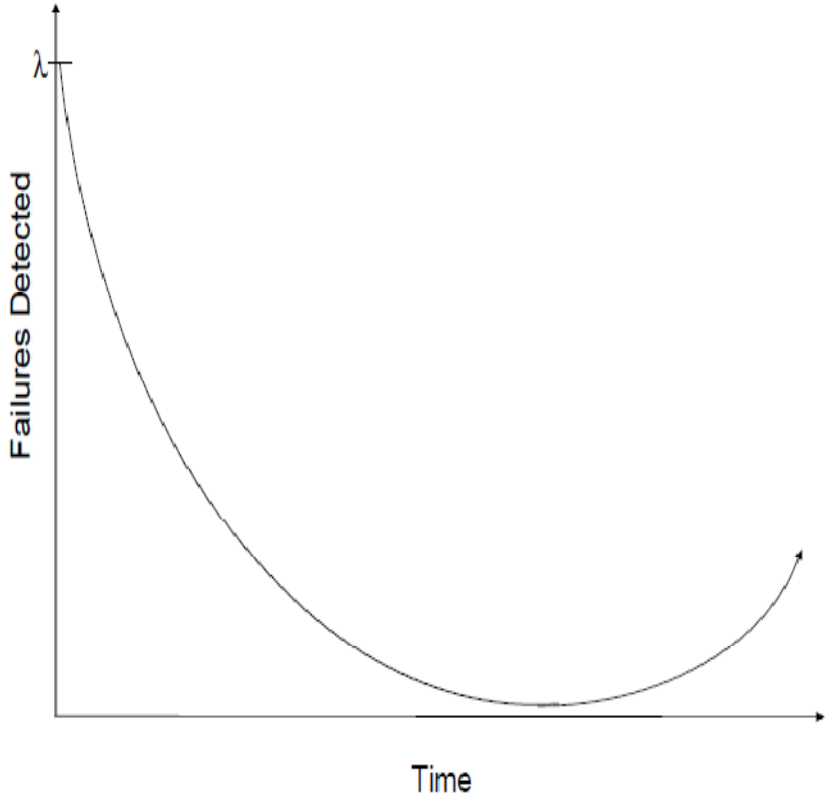
Failure curve for hardware



Failure curve for software



Bathtub curve



Correctness

- Software correctness is closely related to reliability and the terms are often used interchangeably.
- The main difference is that minor deviation from the requirements is strictly considered a failure and hence means the software is incorrect.
- Correctness can be measured in terms of the number of failures detected over time.

Performance

- Performance is a measure of some required behavior — often with respect to some relative time constraint.
- For example,
  - the baggage inspection system may be required to process 100 pieces of luggage per minute.
  - a photo reproduction system might be required to digitize, clean, and output color copies at a rate of one every two seconds.
- One method of measuring performance is based on mathematical or algorithmic complexity. Another approach involves directly timing the behaviour of the completed system with logic analyzers and similar tools.

Usability

- Usability is a measure of how easy the software is for humans to use.
- Software usability is synonymous with ease-of-use, or user-friendliness.
- Usually informal feedback from users, as well as surveys, focus groups, and problem reports are used to measure/determine usability

Interoperability

- This quality refers to the ability of the software system to coexist and cooperate with other systems.
- In many systems, special software called middleware is written to enhance interoperability. In other cases, standards are used to achieve better interoperability.
- For example, in embedded systems, the software must be able to communicate with various devices using standard bus structures and protocols.
- Interoperability can be measured in terms of compliance with open system standards. These standards are typically specific to the application domain.

Open Systems Vs Open Source Code

- An open system is an extensible collection of independently written applications that cooperate to function as an integrated system.
- This concept is related to interoperability.
- Open systems differ from open source code, which is source code that is made available to the user community for improvement and correction.
- An open system allows the addition of new functionality by independent organizations through the use of interfaces whose characteristics are published.
- Any software engineer can then take advantage of these interfaces, and thereby create software that can communicate using the interface.
- Open systems also permit different applications written by different organizations to interoperate.

Maintainability, Evolvability and Repairability

- A software system in which changes are relatively easy to make has a high level of **maintainability**.
- Maintainability can be decomposed into two contributing properties evolvability and repairability.
- **Evolvability** is a measure of how easily the system can be changed to accommodate new features or modification of existing features.
- **Repairability** is the ability of a software defect to be easily repaired.
- Measuring these qualities is not always easy, and is often based on anecdotal observation. This means that changes and the cost of making them are tracked over time.

Portability

- Software is portable if it can run easily in different environments.
- The term environment refers to the hardware on which the system resides, the operating system, or other software in which the system is expected to interact.
- Portability is difficult to measure, other than through anecdotal observation.
- Portability is achieved through a deliberate design strategy in which hardware- dependent code is confined to the fewest code units as possible.

Verifiability

- A software system is verifiable if its properties, including all of those previously introduced, can be verified easily.
- One common technique for increasing verifiability is through the insertion of software code that is intended to monitor various qualities such as performance or correctness.
- Modular design, rigorous software engineering practices, and the effective use of an appropriate programming language can also contribute to verifiability.

Traceability

- Traceability is concerned with the relationships between requirements, their sources, and the system design.
- A high level of traceability ensures that the software requirements flow down through the design and code and then can be traced back up at every stage of the process.
- Traceability can be obtained by providing links between all documentation and the software code.

A set of software code **qualities in the negative** - these are qualities of the code that **need to be reduced or avoided altogether**.

- **Fragility** — When changes cause the system to break in places that have no conceptual relationship to the part that was changed. This is a sign of poor design.
- **Immobility** — When the code is hard to reuse.
- **Needless complexity** — When the design is more elaborate than it needs to be. This is sometimes also called "gold plating."

- **Needless repetition** — This occurs when cut-and-paste (of code) is used too frequently.
- **Opacity** — When the code is not clear.
- **Rigidity** — When the design is hard to change because every time you change something, there are many other changes needed to other parts of the system.
- **Viscosity** — When it is easier to do the wrong thing, such as a quick and dirty fix, than the right thing.

Negative Code Qualities and Their Positives

Negative Code Quality	Positive Code Quality
Fragility	Robustness
Immobility	Reusability
Needless complexity	Simplicity
Needless repetition	Parsimony
Opacity	Clarity
Rigidity	Flexibility
Viscosity	Fluidity

Symptoms for Poor Design (1/3)

When any one of the following occurs, the design will go wrong

- Rigidity – The design is hard to change
- Fragility – The design is easy to break
- Immobility - The design is hard to reuse
- Viscosity - The design is hard to do the right thing
- Needless Complexity – Overdesign
- Needless Repetition – Mouse abuse
- Opacity - Disorganized Expression

Rigidity : The tendency for the software to be difficult to change, even in simple ways. A design is rigid, if a single change causes a cascade of subsequent changes in dependent modules. The more modules need changes, the more rigid the design is.

Fragility : The tendency of a program to break in many places when a single change is made. The new problems may be in areas that have no conceptual relationship with the area that was changed. Fixing those problems may lead to even more problems. As the fragility of a module increases, the likelihood that a change will introduce unexpected problems.

Symptoms for Poor Design (2/3)

Immobility : A design is immobile when it contains parts that could be useful in other systems. But effort and risk involved with separating those parts from the original system are too great, This is un fortunate but very common occurrence. It is hard to separate the system into components that can be reused in other systems.

Viscosity : Doing thing right is harder than doing things wrong.

Two forms : Viscosity of the software : Viscosity of Environment

When faced with a change, the developers usually find more than one ways to make that changes. Some of them preserve the design some others do not. When design preserving methods are methods that are hard to employ, then the viscosity is high. The environment is slow and inefficient the viscosity of environment comes.

Needles of Complexity : A design contains needless complexity when it contains elements that aren't currently useful. This frequently happens when developers anticipate changes to the requirements and pit facilities in the software to deal with potential changes. The effect is often opposite, preparing for too much contingencies the design become filled with constructs that are never used.

Symptoms for Poor Design (3/3)

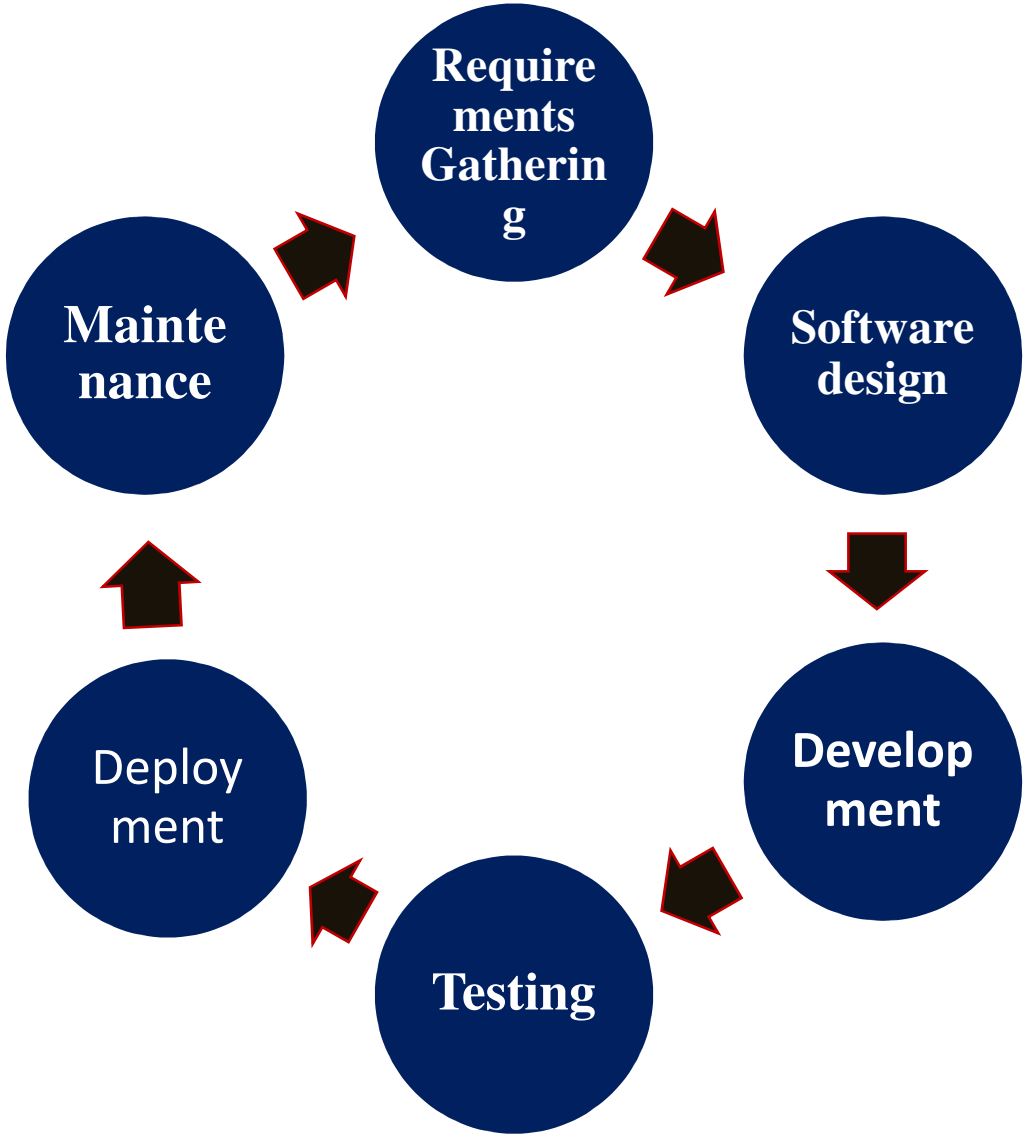
Needless Repetition : : The design contains repeating structures that could be unified under a single abstraction. When code appears again and again, in slightly different forms, the developers are missing an abstraction. Finding all the repetition and eliminating it with an appropriate abstraction may not be high priority but it would go a long way toward making the system easier to understand and maintain. When there is redundant code in the system, the job of changing the system can become hard.

Opacity : It is hard to read and understand. It does not express its intent well. The tendency of a module to be difficult to understand. Code can be written in clear and expressive manner, or it can be written in opaque and convoluted manner. The code that evolves over time tends to become more and more opaque with age. A constant effort to keep the code clear and expressive is required to keep the opacity to a minimum.

- Achieving these qualities is a direct result of a good software architecture, solid software design, and effective coding practices.
- There are many software qualities, some mainstream, others more esoteric or application-specific.

Life cycle of a software system: software design, development, testing, deployment, Maintenance.

Software Development Lifecycle An overview



Software Development Life Cycle (SDLC).

- Software Development Life Cycle (SDLC) is a process used by the software industry to design, develop and test high quality software.
- The SDLC aims to produce a high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.
- It is also called as Software Development Process.
- SDLC is a framework defining tasks performed at each step in the software development process.
- SDLC is a process followed for a software project, within a software organization.
- It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software.
- The life cycle defines a methodology for improving the quality of software and the overall development process.

Requirements Gathering

- First steps in a software project is figuring out the requirements.
- 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 be time-consuming.
- After determining the customers' wants and needs (which are not always the same), can turn them into requirements documents.
- Try to identify the customers and interact with them as much as possible so that the most useful application possible can be designed
- Because software is so malleable (able to be hammered or pressed into shape without breaking or cracking / easily influenced / flexible.), users frequently ask for new features up to the day before the release.
- Customers ask developers to shorten schedules and request last minute changes such as switching database platforms or even hardware platforms
- Requirement means what the system should and should not do.
- Mistake in requirements gathering leads to failure of the system

Software Design- High Level Design

- Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- Software design usually involves problem solving and planning a software solution. This includes both a low-level component and a high-level components
- After getting the project's requirements, Can start working on the high-level design. The high-level design includes such things as decisions about what platform to use (such as desktop, laptop, tablet, or phone), what data design to use (such as direct access, 2-tier, or 3-tier), and interfaces with other systems (such as external purchasing systems).
- The high-level design should also include information about the project architecture at a relatively high level. Should break the project into the large chunks that handle the project's major areas of functionality.
- Should make sure that the high-level design covers every aspect of the requirements.
- It should specify what the pieces do and how they should interact, but it should include as few details as possible about how the pieces do their jobs.

Low Level Design

- After high-level design breaks the project into pieces, can assign those pieces to 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 project should work. The design doesn't need to give every last tiny and minute details necessary to implement the project's major pieces, but they should give enough guidance to the developers who will implement those pieces.
- For example, the car racing application's database piece would include an initial design for the database.
- It should sketch out the tables that will hold the race, participants, and car information.
- Discover interactions between the different pieces of the project that may require changes here and there.
- The project's external interfaces might require a new table to hold e-mail, text messaging, and other information for fans.

Development

- After cresting the high- and low-level designs, it's time for the programmers to get to work.
- (Actually, the programmers should have been hard at work gathering requirements, creating the high-level designs, and refining them into low-level designs, but development is the part that most programmers enjoy the most.) The programmers continue refining the low-level designs until they know how to implement those designs in code.
- (In fact, in one of the technique in development techniques, is basically just keep refining the design to give more and more detail until it would be easier to just write the code instead. Then do exactly that.)
- As the programmers write the code, they test it to make sure it doesn't contain any bugs.
- It's a programming axiom that no nontrivial program is completely bug-free.
- As the programmers write the code, they test it to find and remove as many bugs as they reasonably can.

Testing (1/2)

- Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not.
- It is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual requirements.
- Effectively testing own code is extremely hard. If you just wrote the code, you obviously didn't insert bugs intentionally. If you knew there was a bug in the code, you would have fixed it before you wrote it. That idea often leads programmers to assume their code is correct , so they don't always test it as thoroughly as they should.
- Even if a particular piece of code is thoroughly tested and contains no (or few) bugs, there's no guarantee that it will work properly with the other parts of the system.
- One way to address both of these problems (developers don't test their own code well and the pieces may not work together) is to perform different kinds of tests.

Testing (2/2)

- First developers test their own code.
- Then testers who didn't write the code test it. After a piece of code seems to work properly, it is integrated into the rest of the project, and the whole thing is tested to see if the new code broke anything.
- Any time a test fails, the programmers dive back into the code to figure out what's going wrong and how to fix it. After any repairs, the code goes back into the queue for retesting.
- Unfortunately, you can never be certain that you've caught every bug. If you run your tests and don't find anything wrong, that doesn't mean there are no bugs, just that you haven't found them.
- As programming pioneer Edsger W. Dijkstra said, "Testing shows the presence, not the absence of bugs."
- The best can do is test and fix bugs until they occur at an acceptably low rate. If bugs don't bother users too frequently or too severely when they do occur, then you're ready to move on to deployment.

Deployment

- Ideally, roll out the software, the users are overjoyed, and everyone lives happily ever after. Building a new variant of a product and releasing it on the Internet, the deployment may actually be that simple.
- Often, however, things don't go so smoothly. Deployment can be difficult, time -consuming, and expensive.
- For example, suppose you've written a new billing system to track payments from your company's millions of customers. Deployment might involve any or all of the following:
- ➢ New computers for the back-end database
- ➢ User training
- ➢ A new network
- ➢ New computers for the users
- ➢ On-site support while the users get to know the new system
- ➢ Parallel operations while some users get to know the new system and other users keep using the old system
- ➢ Special data maintenance chores to keep the old and new databases synchronized
- ➢ Massive bug fixing when the users discover dozens or hundreds of bugs that testing didn't uncover
- ➢ Other nonsense that no one could possibly predict

CUTOVER

Cutover is the process of moving users to the new application.

Including data conversion, testing, changeover to the new system, and user training. Compared with traditional methods, the entire process is compressed.

Cutover process involves a series of steps need to be planned, executed and monitored in order to make the project go live. It encompass Cutover Strategy, Cutover Plan, The Cutover Date, Agile Monitoring of the cutover activities and Controlling undesired variations. Its a crucial process and sets focus of the project on important activities which are necessary to make project result for actual use by the end user.

Staged Deployment : build a *staging area , a fully functional* environment where n practice deployment until worked out all the a flaw or imperfection likely to hinder the successful operation

After the installation working smoothly, test the new application in an environment that's more realistic than the one used by the developers. Use the staging area to find and fix a few final bugs before impose them on the users.

Use power users to help do the testing. certain that everything is ready perform the actual deployment on the user's computers. Prepare a deployment plan in case something unexpected goes wrong

Gradual Cutover

Install the new application for some users while other users continue working with their existing system. Move one user to the new application and thoroughly test it, if everything work well for that user, move the second user and so on until everyone is running the new application.

The advantage to this approach is that, this don't destroy every user's productivity if something goes wrong. The first few users may suffer a bit, but the others will continue with business as usual until finding a work-around in the installation procedure. Hopefully, stumble across most of the unexpected problems with the first couple of users, and deployment will be effortless for most of the others

One big drawback to this approach is that the system has a disorder during deployment. Some users are using one system while others are doing something different. Depending on the application, that can be hard to manage. May need to write extra tools to keep the two groups logically separated, or may need to impose temporary rules of operation on the users. A Gantt chart can be used represent this

Incremental Deployment

Release the new system's features to the users gradually. First, install one tool (possibly using staged deployment or gradual cutover to ease the pain). After the users are used to the new tool, give them the next tool.

This method doesn't work well with large monolithic applications because usually can't install just part of such a system

Parallel Testing

Depending on how complicated the new system is, might want to run in parallel for a while to shake the bugs out. Some users start

using the new system in parallel with the old one. They would use the new system to do their jobs just as if the new system were fully deployed.

Meanwhile another set of users would continue using the old system. The old system is the one that actually counts. The new one is used only to see what would happen if it were already installed. After a few days, weeks, or however long it takes to give enough confidence in the new system, then start migrating the other users to the new system

Maintenance

- **Software Maintenance** is the process of modifying a **software** product after it has been delivered to the customer.
- As soon as the users start pounding away on the software, they'll find bugs. (This is another software axiom. Bugs that were completely hidden from testers appear the instant users touch the application.)
- Of course, when the users find bugs, you need to fix them. As mentioned earlier, fixing a bug sometimes leads to another bug, so now you get to fix that one as well.
- If the application is successful, users will use it a lot, and they'll be even more likely to find bugs.
- They also think up a slew of enhancements, improvements, and new features that they want added immediately.
- This is the kind of problem every software developer wants to have: customers that like an application so much, they're clamouring (make a forceful demand) for more. It's the goal of every software engineering project, but it does mean more work.
- 4 types of Maintenance
  - ➢ Corrective Software Maintenance
  - ➢ Perfective Software Maintenance.
  - ➢ Adaptive Software Maintenance.
  - ➢ Preventive Software Maintenance


Thank
you