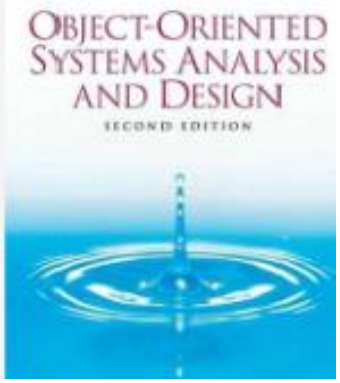
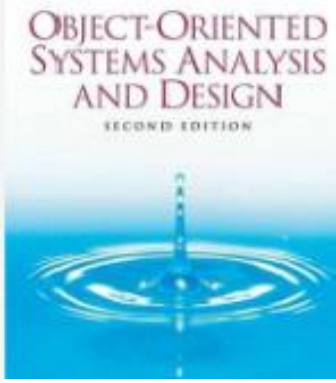
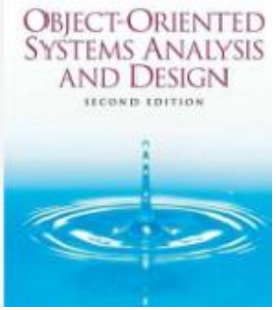
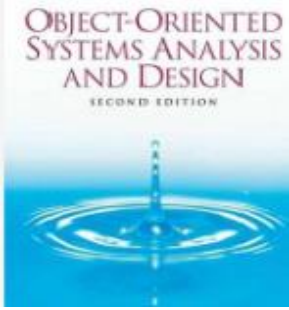




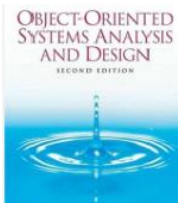
Object Oriented System Analysis and Design (OOSAD)
INSY3063



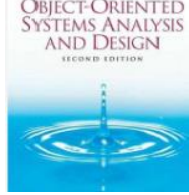
CHAPTER VII



Chapter VII: Object Oriented System Implementation



CHAPTER VI



Chapter 7: Object Oriented Implementation(Coding, Testing and Maintenance.)

7.1 The Cost of Change

7.2. OO Implementation

7.3 Testing Philosophies

7.4 Full Lifecycle Object-Oriented Testing (FLOOT)

7.5 Regression Testing

7.6 Quality Assurance

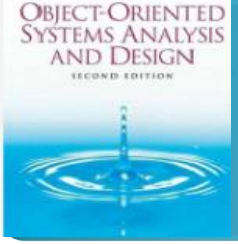
7.7. Testing by Users

7.8 Test-Driven Development (TDD)

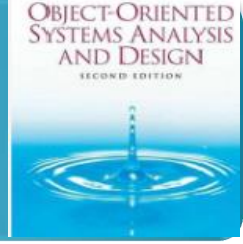
7.9. Transition/Deployment issues

7.10. OO Software Maintenance (Evolution)

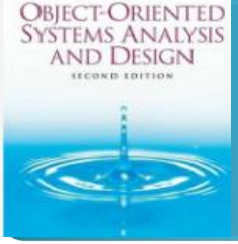
7.11. Documentation



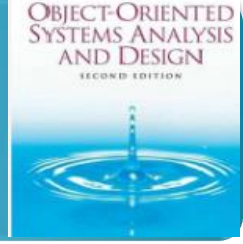
Objectives



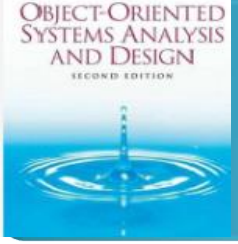
- **At the end of this unit students will be able to**
  - Understand what OO Implementation is
  - Identify things in testing philosophies
  - Identify different techniques in The full-lifecycle object oriented testing (FLOOT) methodology
  - Distinguesh the objectives of Quality Assurance
  - Understand Test Driven Development
  - Identify the issues at the Deployment stage in developing an information system.
  - Understand OO Software Maintenance
  - Identfy types of Documentation in developing an information system.



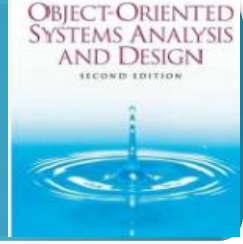
The Cost of Change



- **The Cost of Change**.
  - **The Cost of Change** in software development refers to the e***xpenses incurred when making alterations or updates to a software system after it has been designed, developed, and deployed.*** This concept emphasizes that changes become increasingly costly as the software development lifecycle progresses, with the highest costs typically associated with modifications made **during or after deployment**.
  - **The cost of change in software development** encompasses various **factors** related to the ***timing, complexity, and impact of modifications to software systems***.
  - By considering **these factors** throughout the development lifecycle and **adopting agile practices** such as iterative development, ***continuous integration, and automated testing, organizations can mitigate the cost of change and deliver more adaptable and resilient software solutions***

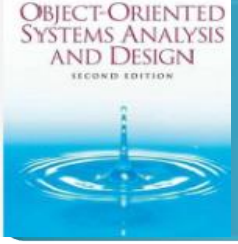


OO Implementation

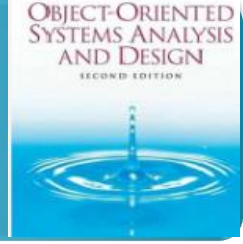


➢ Introduction

- **UML** provides precise and complete models for the three major activities of system development: **analysis, design, and implementation**.
- The three types of **modeling—behavioral, structural, and dynamic** **are** used iteratively throughout the development of the system.
- **Analysis starts** with modeling the behavioral requirements of the system in **conceptual terms**; that is, it models the product as the business wants it to behave, regardless of **how this behavior is achieved.**
- This behavioral model is then transformed into a conceptual model of the system's structure and, finally, into an interactive model.

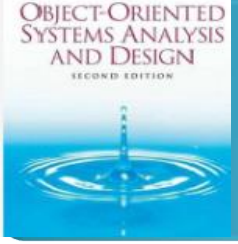


OO Implementation

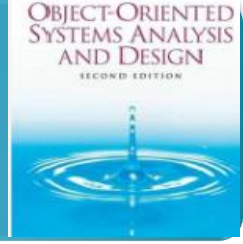


➢ Introduction

- **Design and implementation** use the same types of modeling (with a few new ones), but with different priorities and details. UML supports the required views and the priorities of all three development activities with the same set of basic building blocks.
- **Implementation**, Turns the **blueprints of design into an actual product**. **Programming** is usually the most important component of this activity, but it is not the only one.
- Once the users approve the preliminary design, a detailed design of the system is created and **code is generated**

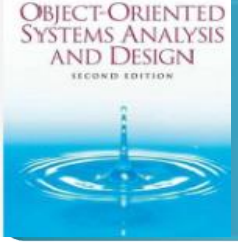


OO Implementation

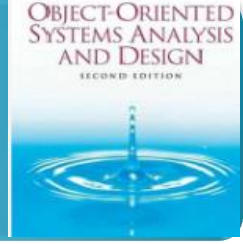


What is Implementation

- The process of development starts with **gathering requirements** and **analysis**, and continues with **design**. It is concluded by **implementation**, a range of activities including **programming, testing, deployment, and maintenance**.
- **Implementation** is a set of activities that transform the **abstract concepts of analysis and the detailed specifications of design into an actual product that people use.**
- **Implementation** is **the last activity of the system development life cycle**.
- It often overlaps with other activities of the development life cycle and is **highly complex and iterative by itself**.
- **The primary purpose of implementation** is to deliver an operational system that meets the needs of the users of the system.

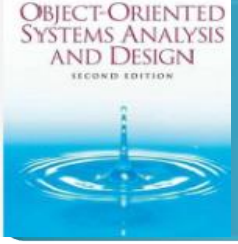


OO Implementation

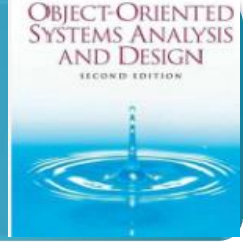


➢ Object-Oriented (OO) Implementation

- **Implementation** starts with **coding and programming**. It includes **testing** that is conducted both in parallel to programming and at its conclusion, and **deployment** that consists of **installation, configuration, and user training**. **Maintenance** is often necessary after deployment and, depending on the reasons, may require a full cycle of development, including analysis and design.
- **Object-Oriented (OO) Implementation** refers to the process of designing and creating software systems using object-oriented programming (OOP) principles.
- In **OOP**, software is **organized into objects**, which are **instances of classes**. These objects encapsulate data and behavior, allowing for **a modular and flexible approach to software development.**



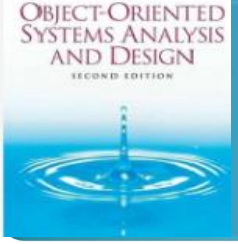
OO Implementation



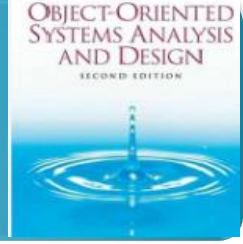
➢ Object-Oriented (OO) Implementation

❖ In implementation process the following concepts have to be under application

- **Classes and Objects**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**
- **Abstraction**
- **Modularity: -** refers to breaking down a software system into smaller, manageable parts (modules or classes). Each module/class handles a specific aspect of the system, making it easier to understand, maintain, and extend.
- **Interfaces**
- **Design**
- **Patterns**



OO Implementation



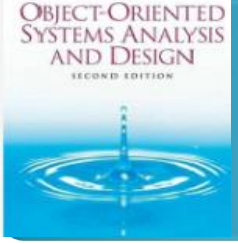
➢ Object-Oriented (OO) Implementation

❖ **In OO implementation**, developers leverage these principles and techniques to create software systems that are **robust, scalable, and maintainable**. By organizing code into classes and objects and applying concepts like **encapsulation, inheritance, polymorphism, and abstraction**, developers can build complex systems while managing complexity effectively.

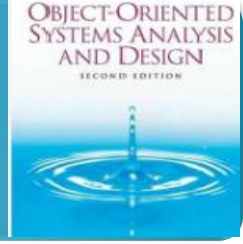
I. The Choice of the Language

❖ The choice of the programming language depends on many factors, among which technical excellence is only one.

❖ In most cases, how a programming language is selected is similar to how a human language is selected: You use it because that is what you know or because that is what somebody else with whom you have to work knows

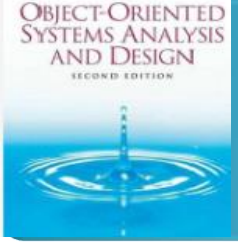


OO Implementation

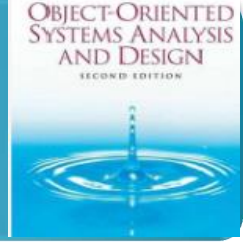


➢ Object-Oriented (OO) Implementation

- There is nothing **inherently wrong** with this picture. Language is **a means of communication, be it with humans or machines**
- It is only after you have mastered the basics of the language that you can communicate effectively. In other words, **the efficiency of using a language to solve a problem has a proportionate relationship to the efficiency of using the language itself.**
- Depending on the methodology and type of information system, developers can Choose the suitable programming language.
- Several programming languages support object-oriented programming (OOP) and are suitable for developing OO systems. Some of the most popular ones include:

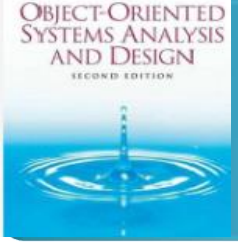


OO Implementation

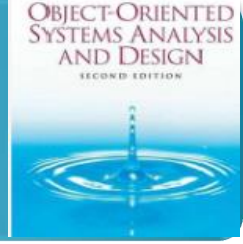


➢ Object-Oriented (OO) Implementation

1. **Java**: Java is one of the most widely used programming languages for building object-oriented systems. It has strong support for OOP concepts like classes, objects, inheritance, polymorphism, and interfaces. Java's platform independence, extensive standard library, and robust ecosystem make it a popular choice for enterprise-level applications, web development, and Android app development.
2. **C++**: C++ is a powerful and versatile programming language known for its efficiency and performance. It offers support for both procedural and object-oriented programming paradigms. C++ allows low-level memory manipulation and provides features like classes, inheritance, polymorphism, and templates. It is commonly used in game development, system programming, and performance-critical applications.

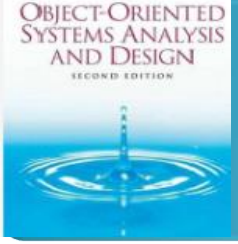


OO Implementation

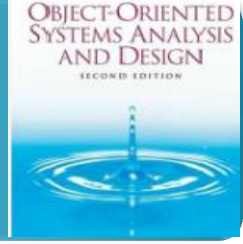


➢ Object-Oriented (OO) Implementation

3. **C#:** C# is a modern, object-oriented programming language developed by Microsoft. It is closely related to Java and provides similar OOP features such as classes, inheritance, polymorphism, and interfaces. C# is commonly used for developing desktop applications, web applications (with ASP.NET), and games (with Unity game engine).
4. **Python**: Python is a high-level, dynamically typed programming language known for its simplicity and readability. It supports object-oriented programming along with other paradigms like procedural and functional programming. Python's syntax makes it easy to learn and use, and it offers features like classes, inheritance, polymorphism, and encapsulation. Python is widely used in web development, data science, artificial intelligence, and scripting.



OO Implementation



- **Object-Oriented (OO) Implementation**

5. **Ruby:** Ruby is a dynamic, reflective programming language known for its simplicity and productivity. It has a clean and concise syntax that promotes writing elegant and maintainable code. Ruby is purely object-oriented, meaning that everything in Ruby is an object. It supports features like classes, inheritance, mixins, and meta programming. Ruby on Rails, a popular web application framework, is built using Ruby.
6. **JavaScript**: JavaScript is a versatile programming language primarily used for client-side web development. With the introduction of ES6 (ECMAScript 2015), JavaScript gained significant improvements in supporting object-oriented programming. It provides features like classes, inheritance, encapsulation (with closures), and polymorphism. JavaScript is now commonly used for both front-end and back-end web development. (**PHP is also the dynamic PL for web based Information systems**)

OO Implementation

- **Object-Oriented (OO) Implementation**

- These are just a few examples, and there are many other programming languages that support object-oriented programming to varying degrees.
- The choice of programming language depends on factors such as **project requirements, developer expertise, performance considerations, and ecosystem support.**

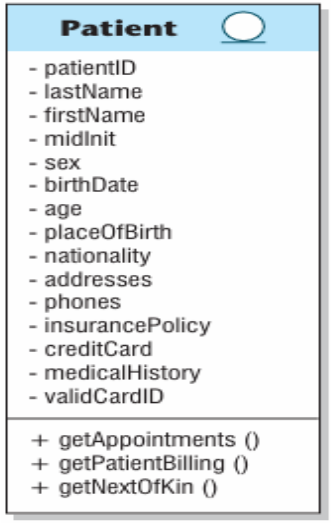
II. **CODING**

- Code is the **brick and the mortar** of a software product. High-quality software cannot result from low-quality code
- Coding is the process of transforming **Design into Software**

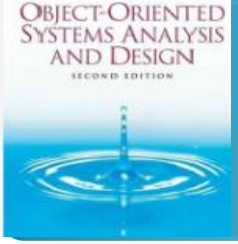
OO Implementation

➢ Object-Oriented (OO) Implementation

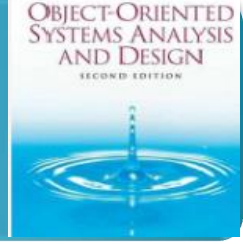
❖ Coding: Transforming Design into Software




```
public class Patient
  {
  1  //Instance Variables.
     private String myFirstName;
     private String myLastName;
     private Date myBirthday;
     private int myID;
…
     // Constructor: NEW Patient 2
     //(ID will be generated when saved).
     Patient ()
        {
           ...
        }
     // Constructor: EXISTING Patient
     //(ID must be provided by the client).
     Patient(int patientID)
     3 {
           ...
           myPersistence = new PatientPersistence();
           myPersistence.getProfile(patientID);
           myFirstName = myPersistence.FirstName();
           myLastName = myPersistence.LastName();
           ...
        }
     // First Name Accessor Methods (get & set)
     public String getFirstName 4
        {
           return myFirstName;
        }
  5  public void setFirstName(String patientFirstName)
        {
            myFirstName = patientFirstName;
        }
...
}
```



OO Implementation



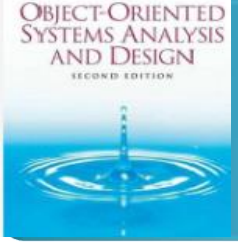
➢ Object-Oriented (OO) Implementation

❖ In the Above slide **right** is only **a fraction of the code necessary to implement the Patient class on the left.**

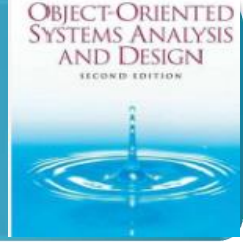
1. Variables that the object uses to store what it knows. These include, but are not limited to, the **attributes**. "Date" and "String" are actually Java classes, the instances of which are used as **variables**.

2. The 1st constructor method. This particular method has **no arguments** and, therefore, is used to **instantiate a new patient.**

3. The 2nd constructor method instantiates a **patient** that **already exists** in the **database**. (**It is also an example of "overloading" methods**.) The Patient object immediately creates another object that retrieves its "**state**" from the database.



OO Implementation



- **Object-Oriented (OO) Implementation**

4. This **"get" method** exposes one of the attributes to the outside world. Without such **accessor methods**, other objects cannot access to private variables.

5. The **"set" method** allows outside objects to change the value of **a private attribute**. Here, through code, the object can **control** the **input value**

- Code Review Code review is an important but often neglected activity.

Software testing philosophies

- **Software testing philosophies** represent the **fundamental principles, approaches, and beliefs** that guide how testing is conducted and managed within an organization. These philosophies shape the **mindset, practices, and culture** surrounding testing activities. Here are some **common software testing philosophies:**

1. **Prevention over Detection:**
   - Focuses on early involvement in the development process, including requirements analysis, design reviews, and code inspections, to identify and address issues proactively.
   - Emphasizes the importance of preventing defects and errors from occurring in the first place rather than relying solely on detecting and fixing them through testing.
2. **Shift Left:**
   - Advocates for moving testing activities earlier in the software development lifecycle, starting from the requirements and design phases and continuing throughout development.
   - Encourages collaboration between developers, testers, and other stakeholders to identify and address defects as soon as possible, reducing the cost and impact of late-stage issues.

➢ Software testing philosophies

3. Risk-based Testing:

- ❖ Prioritizes testing efforts based on the perceived risk and potential impact of defects on the software system and its stakeholders.
- ❖ Focuses testing resources and efforts on areas of the system that are critical, complex, or likely to contain defects, while allocating fewer resources to lower-risk areas.

4. Exploratory Testing:

- ❖ Promotes an exploratory and investigative approach to testing, where testers explore the software system dynamically, learning about its behavior, risks, and quality as they test.
- ❖ Encourages creativity, adaptability, and intuition in testing activities, allowing testers to uncover unexpected issues and behaviors that may not be covered by scripted test cases.

5. Context-Driven Testing:

- ❖ Recognizes that testing is highly contextual and that testing approaches and techniques should be tailored to fit the specific context of the project, team, and organization.
- ❖ Encourages testers to adapt and apply testing practices flexibly, drawing on their skills, experience, and understanding of the project context to achieve effective testing outcomes.

➢ Software testing philosophies

6. **Agile Testing:**
   - ❖ Aligns testing practices with agile software development methodologies, emphasizing collaboration, responsiveness, and incremental delivery.
   - ❖ Integrates testing activities into the agile development process, with testers working closely with developers, product owners, and other team members to ensure that quality is built into the product from the outset.
7. **Continuous Testing:**
   - ❖ Advocates for integrating testing into the continuous integration and continuous delivery (CI/CD) pipeline, with automated tests running continuously throughout the development process.
   - ❖ Emphasizes fast feedback loops, early defect detection, and rapid validation of changes to ensure that software updates can be delivered quickly and confidently.

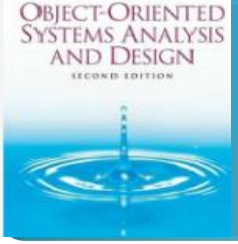
➢ Software testing philosophies

8. User-Centric Testing:

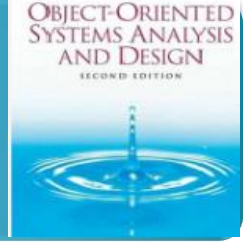
- ❖ Places a strong emphasis on testing from the perspective of end-users, focusing on usability, accessibility, and user experience.
- ❖ Involves real users in testing activities, gathering feedback and insights to ensure that the software meets their needs, preferences, and expectations.

9. Quality Assurance vs. Quality Assistance:

- ❖ Challenges the traditional role of quality assurance (QA) as a separate and independent function responsible for ensuring quality through testing and inspection.
- ❖ Advocates for a shift towards quality assistance, where everyone in the development team takes responsibility for quality and collaborates to prevent defects and deliver high-quality software.
- ❖ These **software testing philosophies** are not mutually exclusive and can be combined and adapted to suit the needs and context of different organizations, projects, and teams. By embracing these philosophies, organizations can foster a culture of quality, innovation, and continuous improvement in their software development and testing practices.



Testing



➢ Software testing philosophies

- ❖ **Testing** is the last defense in assuring software quality. **Testing** allows us to verify the quality of the product or its components in a systematic manner, requirement by requirement and feature by feature. And if the conditions of testing are realistic enough, we might discover unexpected problems and solve them before the product is deployed.

➢ Finding "**bugs**" is included in everybody's expectations from testing. **A bug** is a flaw in the development of the software that causes a discrep ancy between the expected result of an operation and the actual result.

➢ Software testing must provide answers to two important questions:

1. **Validation**: Are we building the right product? The purpose of validation is to ensure that the system has implemented all requirements, so that each function can be traced back to a particular business requirement
2. **Verification**: Are we building the product right? Verification is similar to validation, except that is does not go all the way back to the business requirements, but to the requirements of the activity immediately before the present one. For example: Does logical modeling correspond to the conceptual model? Or: does physical modeling satisfy the demands of the logical model? And so on

Levels of Testing

- Software must be tested for each phase of implementation and deployment. No single test can validate or verify the quality of a piece of software. The details of what tests should be conducted before a software can be declared "ready" depends on the methodology, the testing tools, and, last but not least, the architecture of the system.
- In general we can identify the following categories:

1. **Unit Testing.** "**Unit**" is the smallest piece of code that has an identity. In object-oriented languages, this unit is **a class.** In "**structured**" languages, this unit is **a function**. Unit testing does not mean that we can test a class (or a function) in isolation from others. As you must have gathered by now, classes often use the services of other classes and, therefore, are dependent on them.

- The point of unit testing is to have a small test program, usually called a "test harness," that focuses on the smallest possible collection of classes.

➢ Levels of Testing

- ❖ The test harness invokes each method in the **class** and if the method has **parameters**, allows the programmer (or the tester) to **enter various values** for the **same variable**. The tactic to follow is to first test classes that have **no dependencies** on other classes, then the circle around them that are dependent only on those classes, and so on.
- ❖ As you can imagine, **unit testing** can start simple, but if the software is complex enough it cannot remain simple, or even possible.

2. **Component Testing**. We have strongly recommended component-based development before. Here, we must emphasize its significance in testing. Component testing is not a replacement for unit testing, but follows a similar tactic and should start when unit testing stops. Whereas unit testing can become **cumber some at a high level of complexity**, a component must hide class-level complexities **behind clearly defined and mission-oriented interfaces**. One definition of component is a binary piece of software that is physically separate from other pieces. In the context of component-based development, however, our definition is somewhat different: "A component is a relatively independent, encapsulated, and replaceable unit of software that

Levels of Testing

- The test harness invokes each method in the **class** and if the method has **parameters**, allows the programmer (or the tester) to **enter various values** for the **same variable**. The tactic to follow is to first test classes that have **no dependencies** on other classes, then the circle around them that are dependent only on those classes, and so on.
- As you can imagine, **unit testing** can start simple, but if the software is complex enough it cannot remain simple, or even possible.

2. **Component Testing**. We have strongly recommended component-based development before. Here, we must emphasize its significance in testing. **Component testing** is not a replacement for unit testing, but follows a similar tactic and should start when unit testing stops. Whereas unit testing can become **cumber some at a high level of complexity**, **a component must hide class-level complexities behind clearly defined and mission-oriented interfaces**.

➢ Levels of Testing

2. **Component Testing**. One definition of component is **a binary piece of software that is physically separate from other pieces**. In the context of component-based development, however, our definition is somewhat different: "A component is a relatively independent, encapsulated, and replaceable unit of software that provides services specified by one or more interfaces." This is the relevant definition for component testing. **Component testing,** of course, is ***contingent upon a component-based architecture***. If the whole application is composed of one component, that is, if the application is "monolithic," ***component testing becomes meaningless***. Components can be dependent on other components, and often are. The rule is the dependencies must always be one-way: If component B is dependent on component A, then component A cannot be dependent on component B. If such a circular dependency exists, then something in design has gone wrong and the test results are not helpful or dependable.

➢ Levels of Testing

3. **Integration Testing**. Integration testing is really a continuation of compo nent testing, and there is no firm dividing line between the two. However, whereas in component testing the focus is on individual components, in integration testing every component has to be tested again once a new component is integrated into the whole.
4. **System Testing**. System testing is conducted when all components of a sys tem or subsystem are in place. In effect, system testing is a kind of "dress rehearsal" before the system is shown to the paying clients. System testing is usually long in duration, as every feature must be tested. It can be frustrating as well if previous levels of testing have been ignored or have not been thorough. But it can also reveal problems that cannot be detected by the previous test. These problems essentially revolve around one question: Does the product satisfy requirements? Even if the software passes every previous test with high marks, it can fail here.

➢ Levels of Testing

5. **Acceptance testing** : A product passes acceptance testing when the people who have paid for its development accept it. If the product is for in-house use or is custom software, then the process is rather straightforward: You receive the "test results" by demonstrating the software to the clients and/or by letting them use it for a while. Nevertheless, complications can arise: Who defines the level of the acceptance testing? Who creates the test scripts? Who executes the tests? What is the "pass/fail" criteria for the acceptance test? When and how do we get paid? Acceptance testing for mass-market software is more complicated: If you release the software to the market and wait to get feedback, the "test" will most probably fail. What you have to do, in essence, is to "invent" the client before releasing the software.

➢ Levels of Testing

5. Acceptance testing

There are **two methods** for achieving this goal:

i. **Usability Study.** You hire people to play the client by using the software. The "actors" must be relatively qualified for testing the software and should be given a free hand, but you can help them by providing general test scripts or pointing out features that you would like them to test.

ii. **Beta Release.** Beta release works on a voluntary basis: People will invest time on a prerelease software if they are excited about its promises. Of course, you can always encourage them further by other incentives. For example, you can sell them the final product at a reduced price if they return "bug reports" or provide you with feedback

- **Testing & Object Orientation:** Object orientation can **reduce bugs,** but **not automatically**.
- It can even produce its own kind of unwanted side effects. Through **encapsulation** and **information hiding**, object-oriented technology can reduce defects that result from exposing variables to free-for-all access.

- **Full Lifecycle Object-Oriented Testing (FLOOT)**
- **The full-lifecycle object-oriented testing (FLOOT)** methodology is a collection of
- testing and validation techniques for verifying and validating object-oriented software.
- **FLOOT** is an approach to software testing that focuses on testing object-oriented systems throughout the entire software development lifecycle.
- **FLOOT** encompasses various testing techniques and activities at different stages of development, from requirements analysis to maintenance and evolution.

An overview of FLOOT :

1. **Requirements Analysis**:
   - In the requirements analysis phase, FLOOT involves identifying testable requirements and defining test cases based on these requirements.
   - Test cases may include functional tests, non-functional tests, and acceptance criteria to validate system behavior and quality attributes.

- **Full Lifecycle Object-Oriented Testing (FLOOT)**
- **An overview of FLOOT :**

2. **Design Phase:**
   - During the design phase, FLOOT focuses on designing testable classes and components.
   - Test-driven development (TDD) is often employed, where tests are written before the implementation to drive the design and ensure testability.
3. **Implementation Phase:**
   - In the implementation phase, FLOOT involves writing unit tests to verify the behavior of individual classes and components.
   - Unit tests focus on testing the functionality, boundary conditions, and edge cases of each method and class.
4. **Integration Testing:**
   - Integration testing in FLOOT verifies the interactions and collaborations between classes and components.
   - Test doubles, such as mocks, stubs, and fakes, may be used to simulate dependencies and isolate components for testing.

- **Full Lifecycle Object-Oriented Testing (FLOOT)**
- **An overview of FLOOT :**

5. System Testing:

- System testing validates the behavior of the entire system, including its interactions with external systems and dependencies.
- FLOOT includes functional testing, usability testing, performance testing, security testing, and other types of system-level testing.

6. Acceptance Testing:

- Acceptance testing ensures that the system meets the stakeholders' requirements and expectations.
- FLOOT involves collaborating with stakeholders to define acceptance criteria and conduct acceptance tests to validate these criteria.

- **Full Lifecycle Object-Oriented Testing (FLOOT)**
- **An overview of FLOOT :**

7. Regression Testing:

- Regression testing in FLOOT ensures that changes or updates to the system do not introduce new defects or regressions.
- Automated regression tests are executed regularly to verify the integrity of the system after modifications.

8. Maintenance and Evolution:

- Throughout the maintenance and evolution phase, FLOOT continues to validate and verify the system as it evolves.
- Regression testing, performance monitoring, and defect analysis are ongoing activities to ensure the stability and reliability of the system.

- **Full Lifecycle Object-Oriented Testing (FLOOT)**
- **An overview of FLOOT :**

9. **Continuous Integration and Continuous Testing:**
   - FLOOT integrates testing into the continuous integration (CI) and continuous delivery (CD) pipelines to automate testing and ensure early detection of issues.
   - Automated build and test processes are used to continuously validate changes and deliver high-quality software increments.
10. **Feedback and Improvement:**
   - FLOOT emphasizes feedback loops and continuous improvement based on testing results, defect analysis, and stakeholder feedback.
   - Lessons learned from testing are used to refine testing processes, improve test coverage, and enhance overall software quality.

☞ **By adopting Full Lifecycle Object-Oriented Testing (FLOOT**), organizations can ensure comprehensive testing coverage, early detection of defects, and the delivery of high-quality software products that meet stakeholders' requirements and expectations.

Regression Testing

- Regression testing is a software testing technique used to verify that recent code changes or updates have not adversely affected existing functionality. It involves retesting the unchanged parts of the software system along with the modified components to ensure that no new defects have been introduced and that the existing features continue to work as expected.
- **Concepts of regression testing:**

1. **Purpose**:
   - The primary purpose of regression testing is to ensure that changes to the software, such as bug fixes, enhancements, or updates, do not unintentionally introduce new defects or regressions.
   - Regression testing helps maintain the stability and reliability of the software system by detecting and preventing the reintroduction of previously fixed issues.
2. **Scope**:
   - Regression testing typically focuses on testing the affected areas of the software system, including modified code, related functionalities, and integration points.
   - It also involves testing the unchanged parts of the system to ensure that they have not been adversely affected by the changes.

➢ Regression Testing

3. **Types of Regression Testing**:
   - i. **Unit Regression Testing**: Tests individual units of code (functions, methods, or classes) to verify that recent changes have not broken existing functionality.
   - ii. **Integration Regression Testing**: Tests the interactions between different modules or components to ensure that changes do not disrupt system integration.
   - iii. **System Regression Testing**: Tests the entire system to validate end-to-end functionality and detect any regressions in the overall behavior.
   - iv. **Selective Regression Testing**: Selectively chooses test cases based on the impact analysis of code changes to optimize testing efforts.
   - v. **Complete Regression Testing**: Executes all existing test cases to thoroughly validate the system after changes, ensuring comprehensive coverage.
4. **Regression Test Suites**:
   - ❖ Regression test suites consist of a set of test cases that are selected or prioritized for execution during regression testing.
   - ❖ Test suites may include automated tests, manual tests, or a combination of both, depending on the nature of the changes and the available resources.
   - ❖ Test suites are maintained and updated over time to accommodate new features, changes, and bug fixes.

➢ Regression Testing

5. **Automation**:
   - ❖ Automated regression testing is often employed to streamline the testing process, reduce manual effort, and increase test coverage.
   - ❖ Test automation frameworks and tools, such as Selenium, JUnit, TestNG, and NUnit, are used to automate the execution of regression test cases.
   - ❖ Continuous integration (CI) and continuous delivery (CD) pipelines integrate automated regression testing into the software development process, ensuring that tests are executed automatically with each code change.
6. **Regression Test Selection**:
   - ❖ Regression test selection techniques help identify and prioritize test cases for execution based on the impact of code changes.
   - ❖ Techniques such as code coverage analysis, dependency analysis, and change impact analysis are used to select relevant test cases and optimize testing efforts.

➢ Regression Testing

7. **Frequency**:
   - ❖ Regression testing is performed regularly throughout the software development lifecycle, with the frequency depending on the pace of development, the frequency of code changes, and the criticality of the software system.
   - ❖ It is often conducted as part of the continuous integration and continuous delivery (CI/CD) process to ensure that changes are validated promptly and consistently.

☞ By conducting **regression testing**, organizations can ***mitigate the risk of introducing defects into their software systems, maintain the integrity of existing functionality, and ensure a high level of quality and reliability in their products.***

Quality Assurance (QA)

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN
SECOND EDITION

➢ Quality Assurance (QA)

❖ **Quality Assurance (QA)** is a set of systematic activities implemented within a software development process to ensure that the product being developed meets specified requirements and quality standards. QA encompasses various processes, methodologies, and techniques aimed at preventing defects, identifying issues early, and ensuring that the final product meets customer expectations.

➢ **Quality Assurance in software development:**

1. **Quality Planning**:
   - ❖ Quality planning involves defining quality objectives, criteria, and standards for the software product.
   - ❖ It includes establishing quality metrics, such as defect density, code coverage, and customer satisfaction, to measure and evaluate the quality of the product throughout the development lifecycle.
2. **Process Definition and Compliance:**
   - ❖ QA involves defining and implementing development processes, methodologies, and standards that promote consistency, repeatability, and quality in software development.
   - ❖ Processes may include requirements management, design reviews, code inspections, testing procedures, and configuration management.
   - ❖ Compliance with established processes ensures that development activities are performed according to best practices and industry standards.

Quality Assurance (QA)

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

➢ Quality Assurance in software development:

3. **Requirement Analysis and Validation:**
   - ❖ QA activities begin with analyzing and validating requirements to ensure that they are clear, complete, and achievable.
   - ❖ Techniques such as requirements reviews, traceability matrices, and validation workshops are used to verify that requirements meet stakeholder needs and expectations.
4. **Design and Code Reviews:**
   - ❖ QA involves conducting design and code reviews to identify potential issues, defects, and deviations from standards early in the development process.
   - ❖ Reviews may involve peer inspections, walkthroughs, and formal meetings to assess the quality, correctness, and maintainability of designs and code.
5. **Testing and Quality Control:**
   - ❖ QA includes testing activities to verify and validate that the software product meets specified requirements and quality standards.
   - ❖ Testing encompasses various types and levels, including unit testing, integration testing, system testing, acceptance testing, and regression testing.
   - ❖ Quality control measures, such as defect tracking, issue resolution, and root cause analysis, are used to identify, prioritize, and address defects found during testing.

Quality Assurance (QA)

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

➢ Quality Assurance in software development:

6. Configuration Management:

- QA involves managing configuration items, such as source code, documentation, and build artifacts, to ensure version control, traceability, and integrity throughout the development lifecycle.
- Configuration management practices include version control, change management, baselining, and release management to manage changes and ensure consistency across project artifacts.

7. Continuous Improvement:

- QA promotes a culture of continuous improvement by monitoring, analyzing, and optimizing development processes and practices.
- Lessons learned from project retrospectives, post-mortem analyses, and performance metrics are used to identify areas for improvement and implement corrective actions.

Quality Assurance (QA)

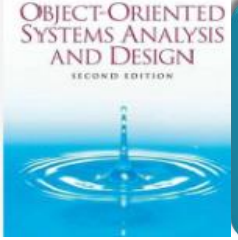
OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

➢ Quality Assurance in software development:

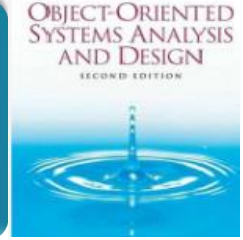
8. Customer Satisfaction and Feedback:

- ❖ QA focuses on meeting customer needs and expectations by soliciting feedback, measuring satisfaction, and incorporating customer input into the development process.
- ❖ Customer feedback mechanisms, such as surveys, feedback forms, and user forums, help gather insights and prioritize enhancements to improve the overall quality of the product.

❑ By implementing effective practices, organizations can reduce the risk of defects, enhance the reliability and usability of their software products, and ultimately deliver value to their customers.



Testing your system in its entirety



➢ Testing your system in its entirety

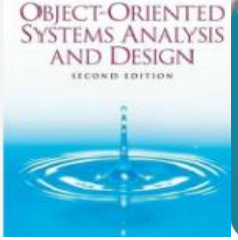
- **Testing your system in its entirety** involves verifying that all ***components***, ***modules***, *and* ***interactions work together as expected*** *to* ***deliver the intended functionality and meet the specified requirements*****.**

➢ Comprehensive guide to testing your system end-to-end:

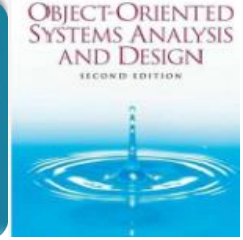
1. **Define Test Scenarios**:
   - Identify and define test scenarios that cover various aspects of system functionality, including typical use cases, edge cases, and error conditions.
   - Consider scenarios that span multiple components or modules and involve interactions with external systems, databases, and user interfaces.
2. **Create Test Data:**
   - Prepare test data that mirrors real-world usage scenarios and covers a wide range of input values and conditions.
   - Ensure that test data includes both valid and invalid inputs to validate the behavior of the system under different circumstances.

Testing your system in its entirety

3. **Integration Testing:**
   - Conduct integration testing to verify the interactions and collaborations between different components, modules, and external systems.
   - Test how components communicate, exchange data, and handle dependencies, ensuring that integration points are robust and resilient.
4. **System Testing:**
   - Perform system testing to validate the end-to-end functionality of the entire system, including its user interfaces, business logic, and data processing capabilities.
   - Test key user workflows, scenarios, and business processes to ensure that the system meets user requirements and expectations.
5. **User Acceptance Testing (UAT):**
   - Involve end-users or representatives from the target audience in user acceptance testing to validate that the system meets their needs and usability requirements.
   - Collect feedback from users and stakeholders to identify any usability issues, inconsistencies, or missing features.



Testing your system in its entirety



➢ Testing your system in its entirety

6. **Performance Testing:**
   - ❖ Conduct performance testing to evaluate the system's responsiveness, scalability, and reliability under different load conditions.
   - ❖ Test the system's performance metrics, such as response times, throughput, and resource utilization, to ensure that it meets performance requirements.
7. **Security Testing:**
   - ❖ Perform security testing to identify and mitigate potential security vulnerabilities and threats in the system.
   - ❖ Test for common security issues such as injection attacks, authentication bypass, and sensitive data exposure, and ensure compliance with security standards and regulations.
8. **Compatibility Testing:**
   - ❖ Test the system's compatibility with different browsers, devices, operating systems, and environments to ensure broad accessibility and usability.
   - ❖ Verify that the system functions correctly and displays properly across various configurations and platforms.

➢ Testing your system in its entirety

9. **Regression Testing:**
   - ❖ Re-run regression tests to ensure that recent changes or updates have not introduced new defects or regressions in the system.
   - ❖ Verify that existing functionality remains intact after modifications and that no unintended side effects have occurred.
10. **Documentation and Reporting:**
   - ❖ Document test results, including test cases, test data, test execution logs, and any issues or defects found during testing.
   - ❖ Generate test reports to communicate the status of testing efforts, highlight any areas of concern, and provide recommendations for further action.

Testing your system in its entirety

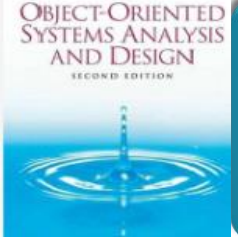
OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN
SECOND EDITION

- **Testing your system in its entirety**.

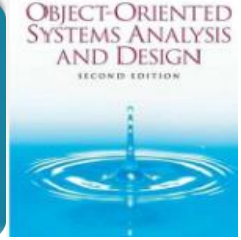
11. Continuous Monitoring and Improvement:

- Continuously monitor the system's performance, reliability, and usability in production environments.
- Gather feedback from users, system administrators, and stakeholders to identify opportunities for improvement and address any issues that arise.

- **By following these steps and best practices,** you can ensure that your system is thoroughly ***tested, reliable, and meets the needs of your users and stakeholders across its entire lifecycle.***



Testing by Users



➢ Testing by Users

❖ **Testing by users**, often referred to as **User Acceptance Testing (UAT),** is a crucial phase in the software development lifecycle where end-users or representatives of the target audience evaluate the system to ensure it meets their requirements and expectations.

➢ **Guide to conducting user testing effectively:**

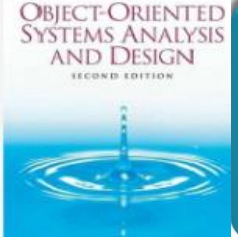
1. **Plan UAT Activities**:
   - ❖ Define the objectives, scope, and criteria for user testing, including the features and functionalities to be tested and the acceptance criteria for success.
   - ❖ Identify the target user groups and stakeholders who will participate in the testing process.
2. **Prepare Test Scenarios:**
   - ❖ EDevelop test scenarios and user stories that reflect real-world usage scenarios and business processes.
   - ❖ nsure that test scenarios cover a range of typical use cases, edge cases, and critical workflows.
3. **Create Test Data:**
   - ❖ Prepare test data that aligns with the test scenarios and represents various user roles, permissions, and data states.
   - ❖ Ensure that test data includes both valid and invalid inputs to validate the system's behavior under different conditions.

Testing by Users

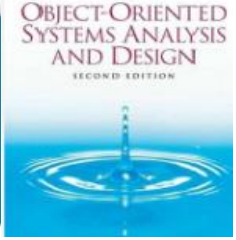
OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN
SECOND EDITION

➢ Testing by Users.

4. **Provide Training and Support:**
   - ❖ Offer training and guidance to users participating in UAT to familiarize them with the system's features, functionalities, and testing procedures.
   - ❖ Provide user documentation, tutorials, and support resources to assist users during testing.
5. **Execute Test Scenarios:**
   - ❖ Have users execute the predefined test scenarios and user stories in the actual system environment.
   - ❖ Encourage users to explore different features, perform typical tasks, and simulate real-world usage to uncover any issues or usability concerns.
6. **Capture Feedback and Issues:**
   - ❖ Collect feedback from users as they navigate through the system and perform test scenarios.
   - ❖ Encourage users to report any issues, bugs, or usability problems they encounter, including suggestions for improvements or enhancements.



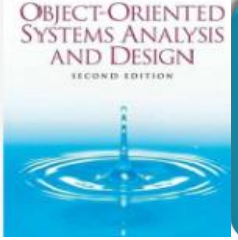
Testing by Users



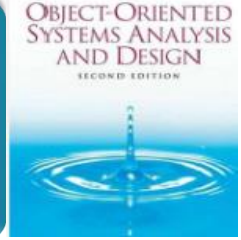
➢ Testing by Users.

7. **Track and Prioritize Issues:**
   - ❖ Document and track reported issues, including their severity, impact, and steps to reproduce.
   - ❖ Prioritize issues based on their impact on usability, functionality, and business objectives, and assign them to appropriate stakeholders for resolution.
8. **Iterate and Retest:**
   - ❖ Address reported issues and implement necessary fixes or enhancements based on user feedback.
   - ❖ Conduct follow-up testing sessions or iterations of UAT to validate that reported issues have been resolved satisfactorily.
9. **Obtain Sign-off:**
   - ❖ Obtain sign-off or approval from stakeholders and users once UAT is completed and all identified issues have been addressed.
   - ❖ Ensure that users are satisfied with the system's functionality, usability, and performance before proceeding to production deployment.
10. **Documentation and Reporting:**
    - ❖ Document UAT results, including test outcomes, identified issues, resolved defects, and user feedback.
    - ❖ Generate UAT reports summarizing the testing process, findings, and recommendations for further action.

➢ By involving **user test process** , you can gain valuable insights into their ***needs, preferences, and expectations, ultimately ensuring that the system meets their requirements and delivers a positive user experience.***



Test-Driven Development (TDD)

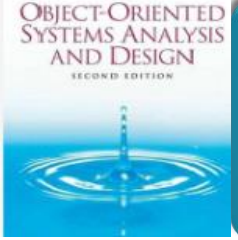


➢ Test-Driven Development (TDD)

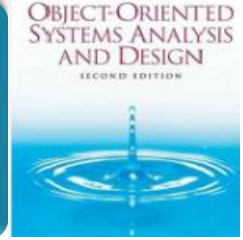
- Test-Driven Development (TDD) is a software development approach where tests are written before the actual code is implemented. It follows a cycle of "red-green-refactor," where tests are initially written and executed, then just enough code is written to pass those tests, followed by refactoring the code to improve its design while keeping the tests passing.

➢ **Breakdown of the TDD process:**

1. **Write a Test (Red)**:
   - Start by writing a test that defines the behavior or functionality you want to implement.
   - The test should fail initially since the corresponding code hasn't been written yet.
2. **Run the Test (Red):**
   - Execute the test and observe it fail. This failure indicates that the test is correctly identifying the absence of the desired functionality.
3. **Write the Code (Green):**
   - Write the minimum amount of code necessary to make the test pass.
   - The focus is on writing code that satisfies the test's criteria without introducing unnecessary complexity or functionality.

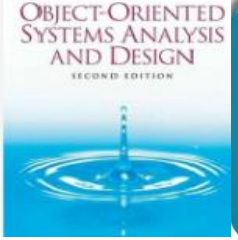


Test-Driven Development (TDD)

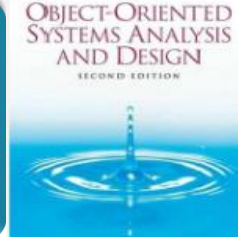


➢ Test-Driven Development (TDD)

4. **Run the Test (Green):**
   - ❖ Execute the test again. If the code changes made in the previous step were successful, the test should now pass.
   - ❖ This step confirms that the newly written code meets the specified requirements and behaves as expected.
   - ❖ **Refactor (Refactor):**
   - ❖ Refactor the code to improve its design, readability, and maintainability while ensuring that the tests continue to pass.
   - ❖ Focus on simplifying the code, removing duplication, and adhering to best practices without changing its behavior.
5. **Repeat the Cycle:**
   - ❖ Continue iterating through the "red-green-refactor" cycle, writing additional tests and code to implement new features, fix bugs, or improve existing functionality.
   - ❖ Each iteration adds small, incremental changes to the codebase while maintaining a high level of test coverage and ensuring that the system remains functional.



Test-Driven Development (TDD)



➢ Test-Driven Development (TDD)

❑ Key Benefits of Test-Driven Development (TDD):

1. **Improved Code Quality:**
   - ❖ TDD encourages writing clean, modular, and well-tested code from the outset, leading to higher overall code quality.
   - ❖ By focusing on writing tests that define the desired behavior upfront, TDD helps clarify requirements and reduce ambiguity in the development process.
2. **Faster Feedback Loop:**
   - ❖ TDD provides rapid feedback on the correctness of code changes, allowing developers to detect and address defects early in the development process.
   - ❖ Failures in tests highlight issues immediately, enabling developers to diagnose and fix them before they become more challenging and costly to resolve.
3. **Increased Confidence in Changes:**
   - ❖ TDD provides a safety net for making changes to the codebase by ensuring that existing functionality is not inadvertently broken.
   - ❖ Developers can refactor code with confidence, knowing that any regressions will be caught by the existing test suite.

Test-Driven Development (TDD)

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

➢ Test-Driven Development (TDD)

❑ Key Benefits of Test-Driven Development (TDD):

4. **Better Design and Maintainability:**
   - ❖ TDD encourages a modular, testable design by promoting the creation of small, focused units of code that are easy to understand and modify.
   - ❖ Refactoring code during the TDD process helps improve its design over time, making it more maintainable and adaptable to changing requirements.
5. **Reduced Debugging Time:**
   - ❖ By catching defects early and providing clear feedback on their causes, TDD reduces the time spent on debugging and troubleshooting issues.
   - ❖ Developers spend less time diagnosing problems and more time writing productive code.

➢ Overall, **Test-Driven Development (TDD)** is ***a powerful technique for building high-quality software systems iteratively while maintaining a focus on delivering value to users.***

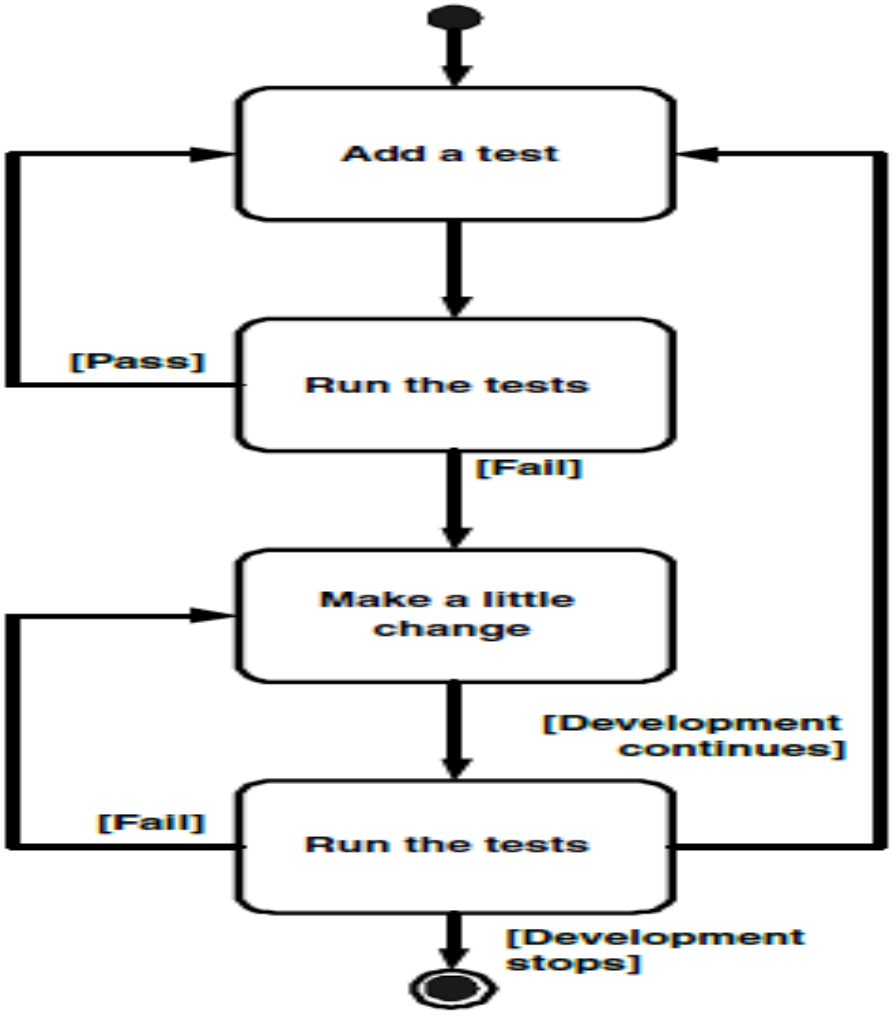
➢ By following the TDD approach, ***teams can achieve faster development cycles, increased code quality, and greater confidence in the reliability and maintainability of their software.***

Test-Driven Development (TDD)

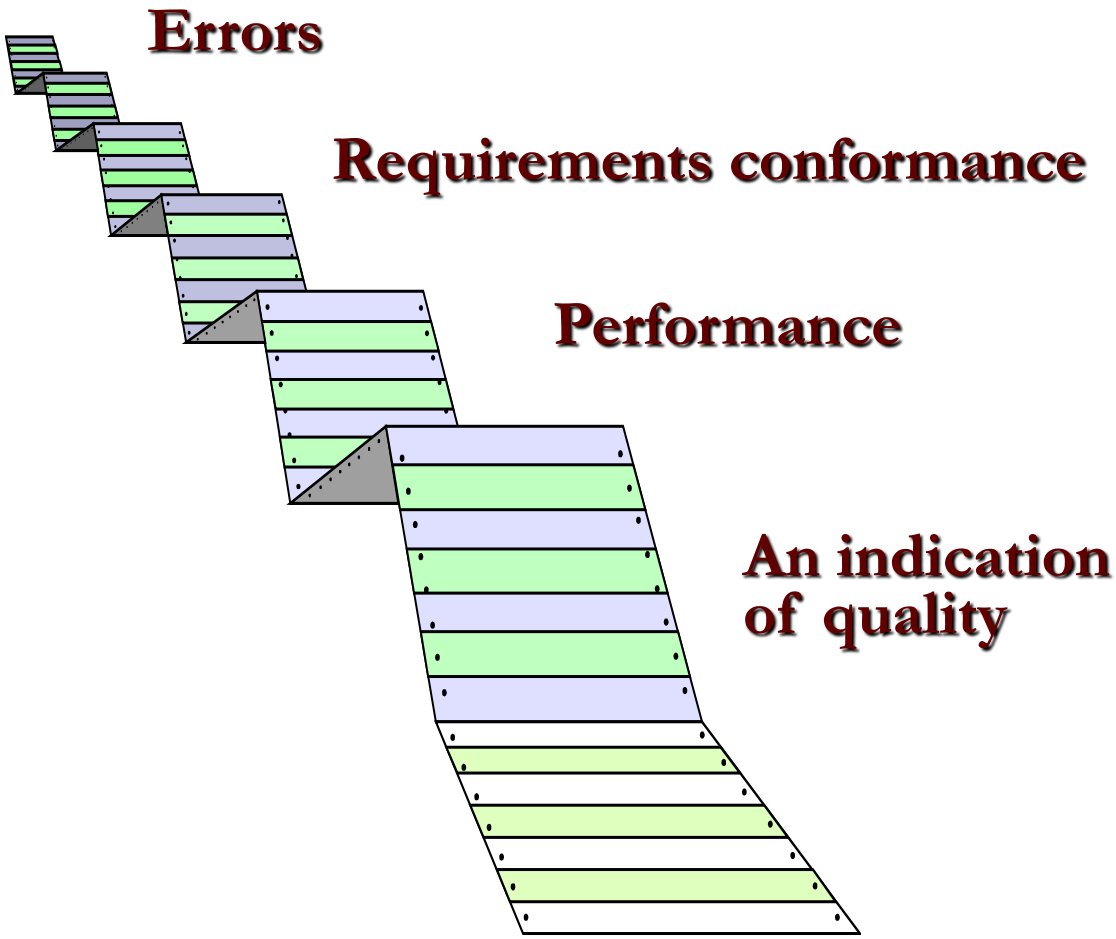
OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN
SECOND EDITION

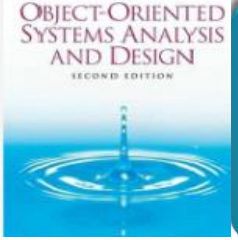
- **Test-Driven Development (TDD)**

- **The steps of TDD**

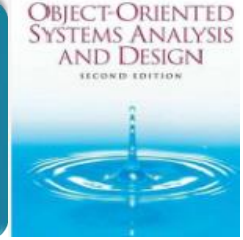


What Testing Shows…





OO Software Maintenance (Evolution)

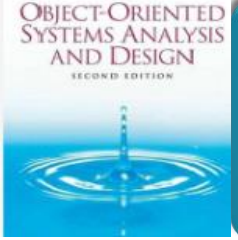


➢ Software Maintenance

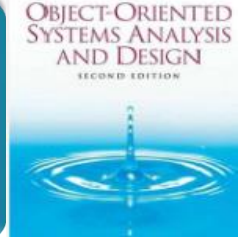
❖ **Software maintenance**, often referred to as software evolution, is the p**rocess of modifying a software product after it has been delivered to the customer**. This includes **making changes to the software to fix bugs, enhance features, improve performance, adapt to new environments, and address security vulnerabilities.**

❖ The need for software maintenance arises from **several factors:**

1. **Changing Requirements:-** As the business environment evolves, the requirements for the software may change. New features may need to be added, existing features may need to be modified, or certain features may need to be removed.
2. **Bug Fixes:-** Bugs are inevitable in software development. Maintenance involves identifying and fixing bugs to ensure that the software operates correctly.
3. **Performance Optimization:-** Over time, the performance of the software may degrade due to changes in data volume, usage patterns, or hardware configurations. Maintenance activities may involve optimizing the software to improve its performance.



OO Software Maintenance (Evolution)

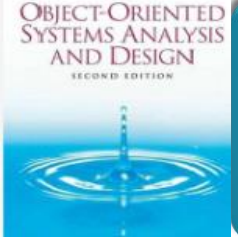


➢ Software Maintenance

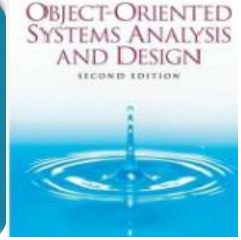
4. **Technology Updates:-** The underlying technologies and platforms on which the software runs may change. Maintenance involves updating the software to remain compatible with new operating systems, libraries, frameworks, or hardware.
5. **Security Updates:-** New security vulnerabilities may be discovered in the software or its dependencies. Maintenance activities involve patching these vulnerabilities to protect the software from security threats.

❑ **There are several types of software maintenance:**

1. **Corrective Maintenance**:- This involves fixing defects or bugs discovered in the software during its operation.
2. **Adaptive Maintenance:-** This involves modifying the software to adapt it to changes in the environment, such as changes in hardware, operating systems, or third-party software.



OO Software Maintenance (Evolution)



➢ Software Maintenance

3. **Perfective Maintenance**:- This involves enhancing the software **by adding new features or improving existing ones** to meet changing user requirements or to enhance performance.
4. **Preventive Maintenance:-** This involves **making changes to the software to prevent future problems**, such as improving error handling or optimizing code for better performance.

- **Effective software maintenance** requires careful planning, documentation, and coordination.
- It's important for organizations to establish processes and procedures for managing maintenance activities, including **prioritizing maintenance tasks, tracking changes, testing modifications, and deploying updates.**
- Additionally, maintaining clear communication with **stakeholders, including end-users and customers, i**s crucial to ensure that their needs are addressed effectively through maintenance efforts.

OO Software Maintenance (Evolution)

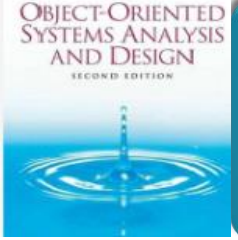
OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

➢ Software preservation

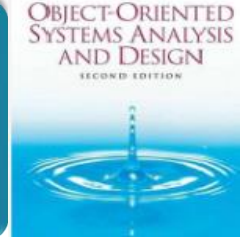
❖ **Software preservation** refers to the efforts aimed at safeguarding software and digital artifacts for future generations. It involves ensuring that software applications, operating systems, games, utilities, and other digital content remain accessible and usable over time, even as technology evolves.

❖ Key aspects of **software preservation**:

1. **Archiving**:- involves creating copies of software and related digital materials, including documentation, manuals, and metadata. These copies are stored in secure repositories to prevent loss due to hardware failures, software obsolescence, or other risks.
2. **Emulation**:- involves recreating the original computing environment in which the software was designed to run. Emulators mimic the behavior of legacy hardware and software systems, allowing old software to run on modern computing platforms.



OO Software Maintenance (Evolution)



➢ Software preservation

3. **Documentation:-** Documentation **plays a crucial role in software preservation** by providing information about the software's functionality, system requirements, installation instructions, and usage guidelines. Comprehensive documentation helps future users understand how to use the software effectively.
4. **Format Migration:-** Format migration involves converting software and digital content from obsolete or proprietary formats to more widely supported and sustainable formats. This ensures that software remains accessible and usable as technology changes.
5. **Legal and Ethical Considerations:** Software preservation efforts must navigate legal and ethical considerations, including **copyright law, licensing agreements, and intellectual property rights.** Preserving software in compliance with relevant laws and regulations is essential to avoid legal issues.

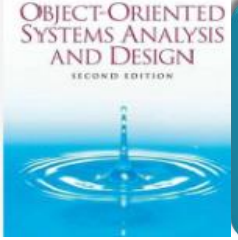
OO Software Maintenance (Evolution)

OBJECT-ORIENTED SYSTEMS ANALYSIS AND DESIGN SECOND EDITION

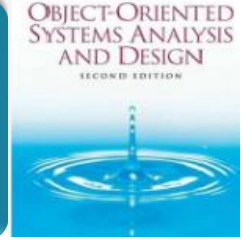
Software preservation

6. **Community Collaboration:-** Collaboration among researchers, archivists, developers, and enthusiasts is critical for successful software preservation. Open-source initiatives, online communities, and collaborative projects facilitate knowledge sharing and resource pooling to support preservation efforts.
7. **Accessibility**:- Making preserved software accessible to users is a key goal of software preservation. This may involve providing online access to archived software repositories, creating user-friendly interfaces for browsing and downloading software, and offering support for running preserved software on modern systems.

- **Software preservation** is essential for preserving cultural heritage, supporting historical research, and ensuring continuity in digital ecosystems. By documenting, archiving, and making software accessible to future generations, we can preserve valuable digital artifacts and contribute to the understanding of technological advancements and their impact on society.

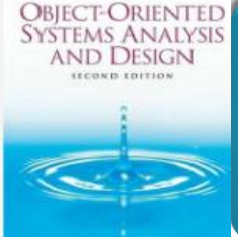


OO Software Maintenance (Evolution)

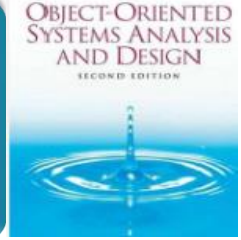


Software Maintenance

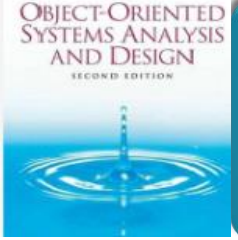
- **In general Maintenance** is often necessary after **deployment and, depending on the reasons, may require a full cycle of development, including analysis and design.**
- **Software** should never be designed under the illusion that it can be perfect and will never change.
- **"Maintenance"** may not be a term that is really appropriate for software, but what it implies is **unavoidable**: Things change and software that is already in use must be **fixed, upgraded, or replaced.**
- **Maintenance** becomes necessary for a variety of reasons, including **engineering defects, failures in analysis and design, obsolescence, change of requirements and expectations, and competition**



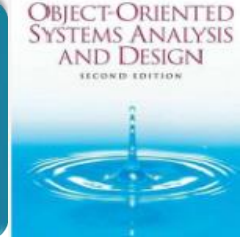
System and User documentation



- **User documentation**
  - **User documentation** must combine **abstract interaction with implementation-specific components and navigation.**
- **User Stories**
  - Agile teams make extensive use of paper artifacts **to plan, display, and store their project information.**
  - These paper artefacts include **user stories** represented on **cards, tasks on post-its, velocity or burn-down charts, etc.**
  - One of the four basic Agile principles is **Working software over comprehensive documentation .**
  - **OO Software Maintenance (Evolution)**

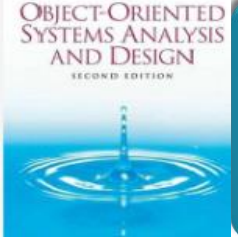


System and User documentation

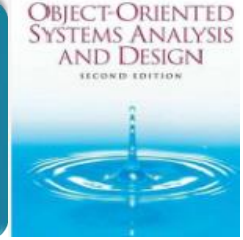


➢ User documentation

- It does not mean **no documentation at all, but rather avoid unnecessary documentation for efficient production.**
- Document in traditional way of development was meant to have a sophisticated way of development process.
- **Documentation** is something written after the software has been developed.
- Most of the software developers thinks that documentation is a collection of wordy, unstructured, contain thousands of pages that nobody wanted to write and nobody trust ?.

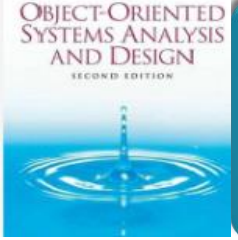


System and User documentation

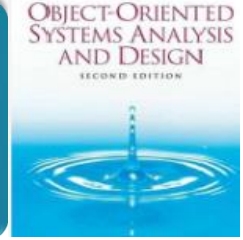


➢ User documentation

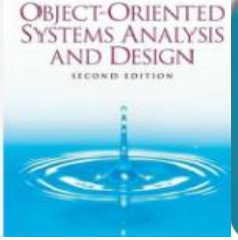
- The advantages of documentation varies from one product to another similarly the consequences of lack of documentation also changes along the product.
- Any product's future maintenance depends basically on the product's documentation.
- **Lack of documentation** means **problems in maintenance and having no documentation means more problems.**
- Besides **maintenance during the development process better documentation helps a lot new members in the development team.**
- Especially internal documentation helps in understanding of the tasks by all the members, every one knows who is doing what and also it has a great impact on the coordination among the team members.



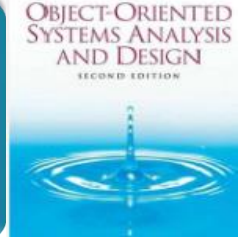
System and User documentation



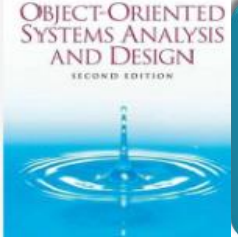
- The documentation of any agile software development process is a vital and valuable component but the amount of time and resources that are given to it must be controlled and optimized not to overwhelm the software development process.
- The usability of the intensive documentation is one of the traditional development limitations, since writing and up-to-date synchronization of the documentations with the code is a time-consuming process especially if the requirements are changing very frequent.
- Therefore, the documentation that matters in agile are the documentation that serves the working software and adds value to the process.



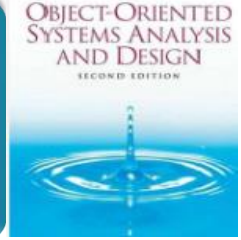
System and User documentation



- They serve **as information radiators** as they are often displayed in highly visible areas on story walls for the benefit of the entire team.
- A participant shared their experience of losing important project data in the form of their story board and all the paper post-its (representing the user stories and tasks):
- A lack of traditional documentation in the form of functional specifications makes it difficult for the two teams to work together.
- The project information captured in the form of **user stories** and tasks on pieces of paper do not easily translate into traditional documentation such as **functional specifications, design documentations, quarterly reports, etc., as used by non-agile teams**

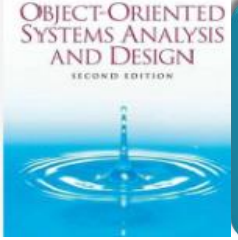


System and User documentation

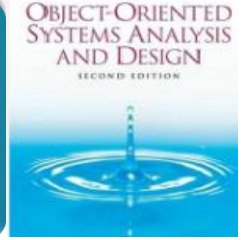


➢ User documentation

- **User documentation** refers to the software product or service which is used by the end user. Many software companies and software developers always try to develop **user documents** for the end user.
- Following are the most important user documentations.
  - Reference manual:- is a document which is designed alphabetically for experienced users
  - User guide
  - Support guide
  - Training material

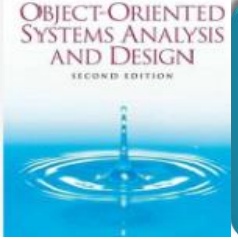


System and User documentation

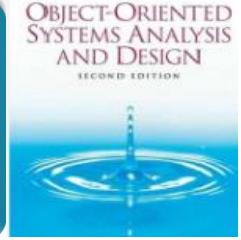


System documentation

- **The documentation of any agile software development process is a vital and valuable component** but the amount of time and resources that are given to it must be controlled and optimized not to overwhelm the software development process.
- **System documentation** is the process of arranging all the documents or information we achieved at the end of each phase.
- **System documentation** is to provide an overview of the system and actually to help the team members understand the system.
- This document is specifically for the developers and maintenance developers
- The information provided by this document includes **technical architecture showing the technical aspects of the system, like how its working and its vital components**, etc.

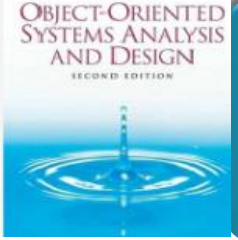


System and User documentation

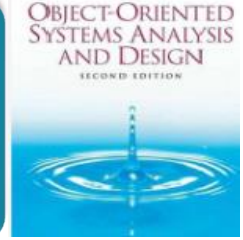


- **System documentation**

- It may also include **the business architecture of the system**, i.e. to provide an idea to the developers what the system is going to produce and how is it going to produce it.
- **The technical and business architecture** shows that the system documentation should at least include high level requirements for the system.
- And so the **detailed architecture and design models** (if there are) should also be presented here in this document.
- **System documentation** means to provide accurate information to the customers that allows them to effectively use and maintain the product.

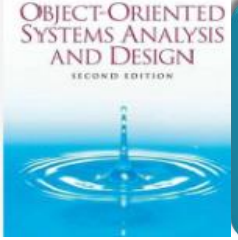


System and User documentation

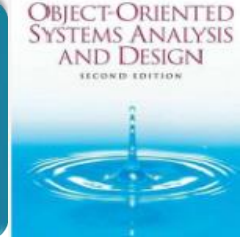


System documentation

- This type of documentation should be written that a person with at least knowledge about that product can understand easily.
- **System documentation** usually provides an overview of the system that the people can understand the system.
- It gives more benefit to the people who are new in the project so those people can understand more things from **system documentation**.



System and User documentation

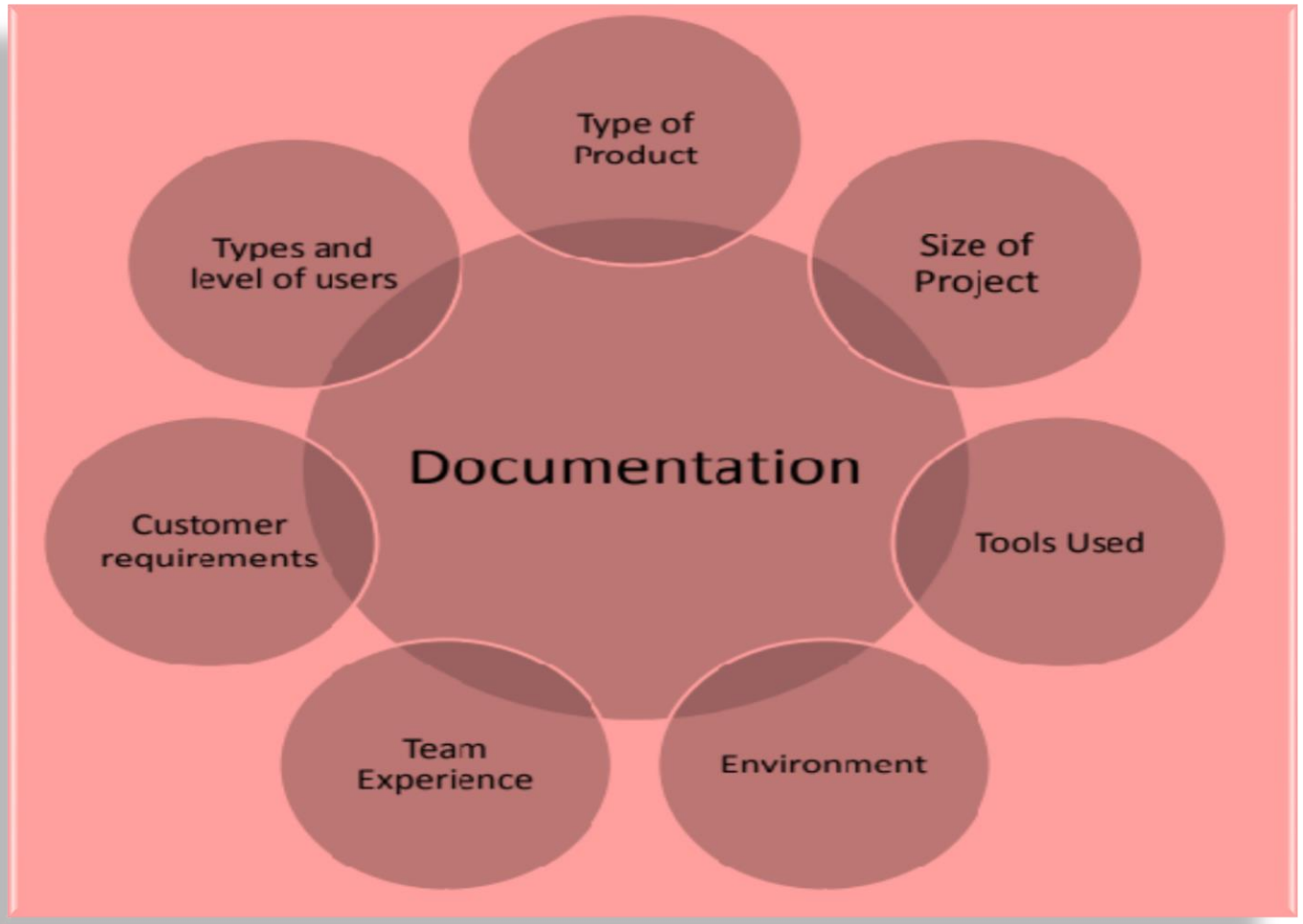


➢ System documentation

❖ Some desirable **characteristics of good system documentation** are given below.

1. **Created for intended audience:**
2. **Specific:**
3. **Relationship with another documentation**
4. **Up to date:**
5. **Sufficiently comprehensive**
6. **Accessible**

System and User documentation



Factors which affect documentation

Object Oriented System Implementation

The End of Unit Seven