

JOMO KENYATTA UNIVERSITY OF AGRICULTURE AND TECHNOLOGY

SCHOOL OF OPEN DISTANCE AND e LEARNING IN COLLABORATION WITH THE
DEPARTMENT OF INFORMATION TECHNOLOGY.

BIT 2321 SOFTWARE ENGINEERING II

BIT 2321 SOFTWARE ENGINEERING

Prerequisite: *ICS 2302 SOFTWARE ENGINEERING I*

Purpose: this course aims to polish a student for software development in the radical and distributed computing environments. Provide the student with the knowledge of building three tier and enterprise applications. It introduces current principles in software development and collaboration and standards of middleware.

CONTENTS

LESSON ONE.....	6
Introduction to software engineering II.....	6
1.1 What is software engineering.....	6
1.1.1 The software engineering process model has the following steps.....	6
1.2 Application of software techniques in today's changing economy.....	6
1.3 Sample Questions.....	6
LESSON TWO.....	7
Software design patterns and collaborative development.....	7
2.1 Introduction to software design patterns that can be used in software engineering.....	7
2.2 Java Design Pattern's.....	7
2.3 Introduction to the java framework (MVC, struts, spring, JSF).....	10
2.3.1 Benefits of the MVC model.....	10
2.4 How to implement selected design pattern in java.....	11
2.5 Chapter question.....	13
LESSON THREE.....	14
Component based technologies used in software engineering.....	14
3.1 Introduction to component based engineering and technologies.....	14
3.2 Importance of component based engineering in software development.....	14
3.3 Application of Component based technologies in software development.....	14
3.4 COM.....	15
3.5 EJB.....	16
3.6 CORBA.....	17
3.7 Component collaborative development process.....	18
3.8 Chapter Questions.....	19
LESSON FOUR.....	20
Middle ware enterprise development and service oriented programming.....	20
4.1 Introduction to middle ware enterprise development.....	20
4.2 Importance of middle ware development in enterprise system development.....	21
4.3 Model Integration Computing.....	21
4.4 Integrating MIC with component middleware.....	22
4.5 Service oriented programming and its application in software engineering.....	26
4.5.1 Service oriented programming elements.....	26
4.5.2 Aspects of service oriented programming.....	27
4.5.3 Benefits of Service Oriented Programming.....	27

4.6 Chapter Questions.....	28
LESSON FIVE.....	29
Software validation and verification.....	29
5.1 Introduction to software validation and verification.....	29
5.2 The difference between validation and verification of software.....	29
5.2.1 Where is each applicable in the development process of the software.....	29
5.3 Introduction to software testing.....	29
5.4 Testing procedures and strategies that can be used to test the software.....	29
5.5 Different stages of testing for the software.....	30
5.6 Importance of validating and verifying software.....	31
5.7 Chapter Questions.....	31
LESSON SIX.....	32
Computer software Project management.....	32
6.1 Introduction computer software project management.....	32
6.2 Activities in Software project management.....	32
6.3 Methodologies in software project management.....	33
6.3.1 Rational Unified Process (RUP).....	33
6.3.2 SSADM.....	34
6.4 Techniques in software project management.....	35
6.4.1 Project Management body of Knowledge (PMBOK).....	35
6.4.2 COCOMO.....	35
6.4.3 Milestone Trend Analysis.....	36
6.4.4 Earned value management (EV).....	37
6.5 Chapter Questions.....	38
LESSON SEVEN.....	39
Object Oriented Software development.....	39
7.1 Introduction to object oriented software development.....	39
7.2 Use of UML diagrams in O.O software development technique to model/design software..	40
7.3 TERMS AND CONCEIPTS.....	40
7.3.1 Class and object.....	40
7.3.2 Data Abstraction and Encapsulation.....	41
7.3.4 Access modifiers/rules of encapsulation.....	42
7.3.5 Polymorphism and dynamic binding.....	43
7.4 BENEFITS OF OOP.....	48

7.5 APPLICATIONS OF OOP.....	48
7.6 Chapter Questions.....	52
LESSON EIGHT.....	53
Computer software Cost estimation.....	53
8.1: Introduction to software cost estimation.....	53
8.1.1 Accurate cost estimation is important because:.....	53
8.2: Methods used to estimate software cost.....	53
8.2.1 Algorithmic methods.....	54
8.2.2 Non algorithmic methods.....	54
8.3 COCOMO I.....	54
8.4: Function point.....	55
8.5: Software Line of Code.....	56
8.6 Chapter questions.....	56
LESSON NINE.....	57
Quality management and software maintenance.....	57
9.1 Introduction to software maintenance.....	57
9.1.1. Software maintenance activities.....	57
9.1.2. Techniques for maintenance.....	59
9.3 Quality management and importance in software engineering.....	59
9.3.1 Classification of Quality requirements.....	60
9.3.2 Software Quality Architecture.....	61
9.4 Configuration management and its importance in software engineering.....	62
9.5 Chapter Question.....	62
LESSON TEN.....	63
Software prototype and formal specification.....	63
10.1 Introduction to software prototyping.....	63
10.1.1 Software prototyping strategies.....	64
10.2 Online rapid prototyping techniques.....	65
10.3 Offline rapid prototyping techniques.....	65
10.4 Chapter Questions.....	66

LESSON ONE

Introduction to software engineering II

Learning outcomes:

Upon completion of this lesson, the learner should be able to:

- Recap the definition of software engineering as learnt in software engineering one
- Appreciate the need for software engineering in developing high end and quality enterprise software's.

1.1 What is software engineering

Before even we define the term software engineering, we should ask ourselves to define the term software. Software is an aspect in computing that deals with providing an abstraction that helps the user interact with hardware on the machine. Therefore software helps us users accomplish a certain task. Software can be tailor made or general purpose software.

Software engineering is therefore the science and art of building significant software systems that are on time, within the budget, performing the correct operation and within the stipulated user requirements.

1.1.1 The software engineering process model has the following steps:

- **Specification**: Set out the requirements and constraints on the system.
- **Design**: Produce a model of the system.
- **Manufacture**: Build the system.
- **Test**: Check the system meets the required specifications.
- **Install**: Deliver the system to the customer and ensure it is operational.
- **Maintain**: Repair faults in the system as they are discovered.

1.2 Application of software techniques in today's changing economy.

Software engineering is mostly used in developing robust and agile applications and software's in our ever changing technological world. From military applications to basic home applications, software engineers apply the methods and principles from the software engineering practices that have evolved over the years.

1.3 Sample Questions

a) Define the term software engineering; in addition explain its importance in the software development process.

LESSON TWO

Software design patterns and collaborative development

Learning outcomes

Upon completion of the lesson, the learner should be able to:

- Be able to describe, recognize, apply and implement selected design patterns in java.
- Be able to implement these designs in software project
- To explain the link between a selected design pattern and the design methodology of choice in use.

2.1 Introduction to software design patterns that can be used in software engineering

Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. Patterns are used in software engineering to generalize a solution to a particular design problem which shall be a blue print in essence for the software.

The following are elements of a design pattern:

- a) Pattern name: useful vocabulary that is used to describe the pattern.
- b) Problem applicability and area of applicability: describes the area where the pattern can be applied and the problem it can solve.
- c) Solution: the solution to the problem to which it was designed for in that problem domain.
- d) Consequences of applying that design pattern: it should specify the course of the whole process, the pros and cons of applying that design pattern.

2.2 Java Design Pattern's.

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

In 1994, four authors Erich Gamma, Richard Helm; Ralph Johnson und John Vlissides published a book titled Design Patterns - Elements of Reusable Object-Oriented Software which initiated the concept of Design Pattern in Software development.

These authors are collectively known as Gang of Four (GOF). According to these authors design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favor object composition over inheritance

Design Patterns have two main usages in software development.

- Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

- **Best Practices**

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps un-experienced developers to learn software design in an easy and faster way.

As per the design pattern reference book **Design Patterns - Elements of Reusable Object-Oriented Software**, there are 23 design patterns. These patterns can be classified in three categories: Creational, Structural and behavioral patterns. As shown in the table below, the pattern name and description of its work.

Pattern & Description	Description
Creational Patterns	These design patterns provides way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.
Structural Patterns.	These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities
Behavioral Patterns.	These design patterns are specifically concerned with communication between objects
J2EE Patterns	These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

The patterns described in the table below are some of the most common, basic and important design patterns one can find in the areas of object-oriented design and programming. Some of these fundamental design patterns, such as the Interface, Abstract Parent, Private Methods, etc.,

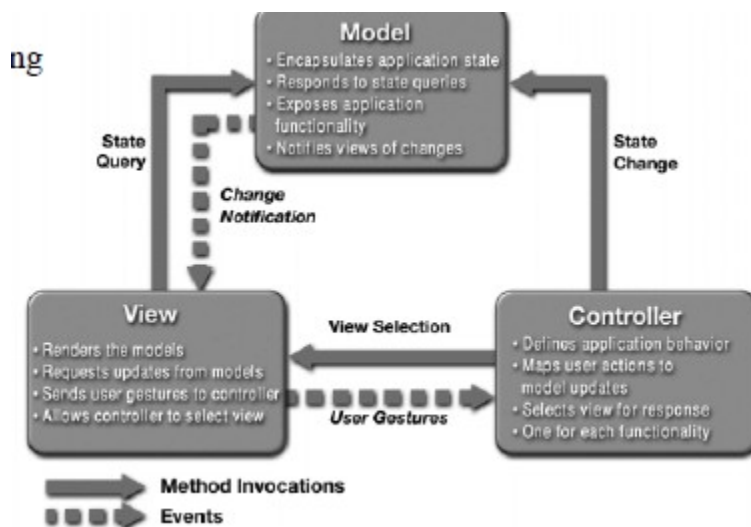
Pattern Name	Description of work done by the pattern
Interface	Can be used to design a set of service provider classes that offer the same service so that a client object can use different classes of service provider objects in a seamless manner without having to alter the client implementation.
Abstract Parent Class	Useful for designing a framework for the consistent implementation of the functionality common to a set of related classes.
Private Methods	Provide a way of designing a class behavior so that external objects are not permitted to access the behavior that is meant only for the internal use.
Accessor Methods	Provide a way of accessing an object's state using specific methods. This approach discourages different client objects from directly accessing the attributes of an object, resulting in a more maintainable class structure
Constant Data Manager	Useful for designing an easy to maintain, centralized repository for the constant data in an application.
Immutable Object	Used to ensure that the state of an object cannot be changed. May be used to ensure that the concurrent access to a data object by several client objects does not result in race conditions.

2.3 Introduction to the java framework (MVC, struts, spring, JSF)

MVC - Model-View-Controller - is a design pattern for the architecture of web applications. It is a widely adopted pattern, across many languages and implementation frameworks, whose purpose is to achieve a clean separation between three components of most any web application, these three components are:

- Model: business logic & processing
- View: user interface (UI)
- Controller: navigation & input

In general, J2EE applications follow the MVC design pattern



2.3.1 Benefits of the MVC model

MVC benefits fall into three categories:

1. Separation of concerns in the codebase

Separation of Model, View, and Controller:

- allows re-use of business-logic across applications
- allows development of multiple UIs without touching business logic codebase
- discourages "cut-&-paste" repetition of code, streamlining upgrade & maintenance tasks

2. Developer specialization and focus

UI (User Interface) developers can focus exclusively on UI, without getting bogged down in business logic rules or code.

Business logic developers can focus on business logic implementation and changes, without having to slog through sea of UI widgets

3. Parallel development by separate teams

Business logic developers can build "stub" classes that allow UI developers to forge ahead before business logic is fully implemented. UI can be reworked as much as the customer requires without slowing down the development of code that implements business rules.

Business rule changes are less likely to require revision of the UI (because rule changes don't always affect structure of data the user sees).

2.4 How to implement selected design pattern in java

Model Implementation Technologies

Within the J2EE framework, there are a range of technologies that may be applied to implementation of the model layer. These include:

- Object-Relational Mapping Frameworks (Java Objects <-> Relational Databases):
- Hibernate, an open-source persistence framework
- Oracle's TopLink
- Sun's EJB (Enterprise Java Beans)
- Sun's JDO (Java Data Objects)
- Hand-coded data accessors via the JDBC API

The attributes and merits of each of these solutions are outside the scope of this presentation. In general, however, Object-Relational mapping frameworks involve a layer of abstraction and automation between the application programmer and data persistence, which tends to make application development faster and more flexible.

Differences between specific O-R Mapping technologies turn on the array of services offered by each technology, vendor specificity, and how well these differences map to a particular application's development and deployment requirements.

View (Detail)

Elements of the View:

The application's View is what the user sees. In a web application, the View layer is made up of web pages; in a web application that can be accessed by non-traditional browsers, such as PDAs or cell-phones, the View layer may include user interfaces for each supported device or browser type.

In any case, the View includes:

- Core data - the subject of a page's business
- Business logic widgets (e.g., buttons to perform Edit or Save)
- Navigation widgets (e.g., navigation bar, logout button)
- Skin (standard look of the site: logos, colors, footer, copyright, etc.)

• View Technologies

In the context of a web application, view technologies are used to render the user interface (UI) in a way that properly and dynamically links it to the application's business-logic and data layer (i.e., to the model).

Java Server Pages (JSPs) are a widely-utilized technology for constructing dynamic web pages. JSPs contain a mix of static markup and JSP-specific coding that references or executes Java. In brief:

- JSP Tags call compiled Java classes in the course of generating dynamic pages.
- JSP Directives are instructions processed when the JSP is compiled. Directives set page-level instructions, insert data from external files, and specify custom tag libraries.
- Java code - in sections called "scriptlets" - can be included in JSP pages directly, but this practice is strongly discouraged in favor of using JSP Tags

XML Pipelining is another technique for rendering the User Interface. Apache's Cocoon and Orbeon's OXF are technologies that use this technique. (XML pipelining simply means executing a series of transformations of XML documents in a proscribed order. For example, a data structure pulled from a persistent store may be rendered as an XML document, then - in a pipeline may be augmented with data from a separate data structure; sorted; and rendered in HTML format using a series of XSLT transformations. A simple example of an XML pipeline using Apache's Ant tool can be viewed in this article.)

• Templates

In order to maintain a consistent look-and-feel on a site, pages are often in template form.

The web page's <body> is where core business content of a page can be found. Using a template to render a consistent header, footer, and navigation UI around the core business content standardizes a site's graphic presence, which tends very strongly toward making users' experience smoother and less prone to error

• Styling

Cascading Style Sheets are a critical component of rendering a consistent look and feel in a web site (cf. this CSS Tutorial for more on Cascading Style Sheets). CSS is used to specify colors, backgrounds, fonts and font-size, block-element alignment, borders, margins, list-item markers, etc. By specifying the designer's choices in a single file (or group of files), and simply referring to the style sheet(s) in each web page, changes in style can be accomplished without altering the pages themselves.

• The Decorator Pattern

Templating and styling are aspects of an important "Gang of Four" design pattern, called the Decorator Pattern. Using the Decorator Pattern simply means wrapping some core object in something that gives it additional functionality. In the context of building a user interface for the web - the View layer of an MVC-architected application - this might mean wrapping:

- core business information in a
- template (e.g., header + footer + navigation), which includes (in the common HTML header) reference to a
- CSS style sheet (where background, colors, fonts, etc. are specified)

Controller Detail

- Responsibilities of the Controller:

The Controller will do the following:

- Parse a user request (i.e., "read" it)
- Validate the user request (i.e., assure it conforms to application's requirements)
- Determine what the user is trying to do (based on URL, request parameters, and/or form elements)
- Obtain data from the Model (if necessary) to include in response to user
- Select the next View the client should see The sequencing of calls to the Model (business-logic layer), and/or the sequencing of views and required input from the user defines the application's workflow. Workflow is thus defined in the Controller layer of the application.

- **Controller Technologies**

There are numerous application frameworks that can be used to provide the control layer in an MVC-architected application.

These include:

- Struts
- Java Server Faces
- WebWork
- Spring

2.5 Chapter question

- a) Explain why design patterns are important while designing software's.
- b) Explain the java design pattern; in addition explain the various types of java design patterns that can be used.
- c) Explain how the MVC model is used in the java framework

LESSON THREE

Component based technologies used in software engineering

Learning outcomes: at the end of the lesson the learner should be able to:

- To compare and contrast different software components and their importance in interfacing development.
- To understand how the COM, EJB, XML components are used in developing components in software development
- Be able to understand the CORBA architecture and the ORB layer
- To show how applications can be run on the CORBA architecture.
- The importance of the component in collaborative development

3.1 Introduction to component based engineering and technologies.

A *component* is a reusable unit of deployment and composition that is accessed through an interface. Component is an independent entity with a complete functionality that can be upgraded and distributed separately.

Components are high level aggregations of smaller software pieces, and provide a 'black box' building block approach to software development. Moreover, it encapsulates the implementation detail from the environment and can be reused by their interfaces.

The CBSD (Component Based Software development) focuses on assembling pre-existing software components for building large software systems. This approach not only enhances the flexibility and maintainability of systems but reduces overall software development costs and assembles systems rapidly.

3.2 Importance of component based engineering in software development

- It accelerates the software development process
- Reduces cost of developing the said software
- Enhances software quality for it uses already pre-established components in the design and development of these software.

3.3 Application of Component based technologies in software development

The component models define the standards forms and standard interfaces between the components. They make it possible to components to being deployed and to communicate. The communication can be established between components on the same node (computer) or between different nodes. For the later we are talkies about component distribution. Component models are the most important step to lift the component software off the ground. If components are developed independent of each other then it is highly unlikely that components developed under such conditions are able to cooperate usefully. The primary goal of component technology, independent deployment and assembly of components is not achieved. A component model supports components by forcing them to conform to certain standards and allows instances of these components to cooperate with other components in this model.

In CBSD, the notion of building a system by writing code has been replaced with building a system by assembling and integrating existing software components. There are four activities in component-based development approach:

- **Component qualification**

It is a process of determining components suitability of reusing the previously developed components in the next system. This process also can be used to select the components when a market place for competing products exists. Among the many factors that should be considered during the components qualification includes the functionality and services that a component is providing and some other aspects like the standards that are used and some quality aspects such as component reliability, predictability, and usability etc.

- **Component adaptation**

The components are developed to meet different requirements, and based on different assumptions about their context. Thus, for their reusability it must be adapted based on rules that ensure to minimize the conflicts among components in new systems. To mitigate against these conflicts, an adaptation technique called component wrapping [5] is often used. Rogers S. Pressman defined different wrapping approaches for components adoptions like: *White box wrapping* examines the internal processing details of the component and makes code level modifications to remove any conflict. *Gray-box wrapping* is applied when the component library provides a component extension language or API that enables conflicts to be removed or masked. *Black-box wrapping* requires the introduction of pre and post processing at the component interface to remove or mask conflicts.

- **Assembling components**

Components are assembled or integrated through some well-defined infrastructure, which provides the binding that forms a system from disparate components. COTS components, for example, are usually written to some component model defined by e.g., Enterprise Java Beans (EJB), COM, CORBA, and .NET etc.

- **System evolution**

To meet the business needs, the system evaluation is very important. In CBS, system evolution is based around the replacing of outdated components by new ones, or, at least, ideally. The treatment of components as plug-replaceable units is a simplistic view of system evolution. In practice, replacing a component may be a nontrivial task, especially when there is a mismatch between the new component and old one, triggering another stage of adaptation with the new component.

3.4 COM

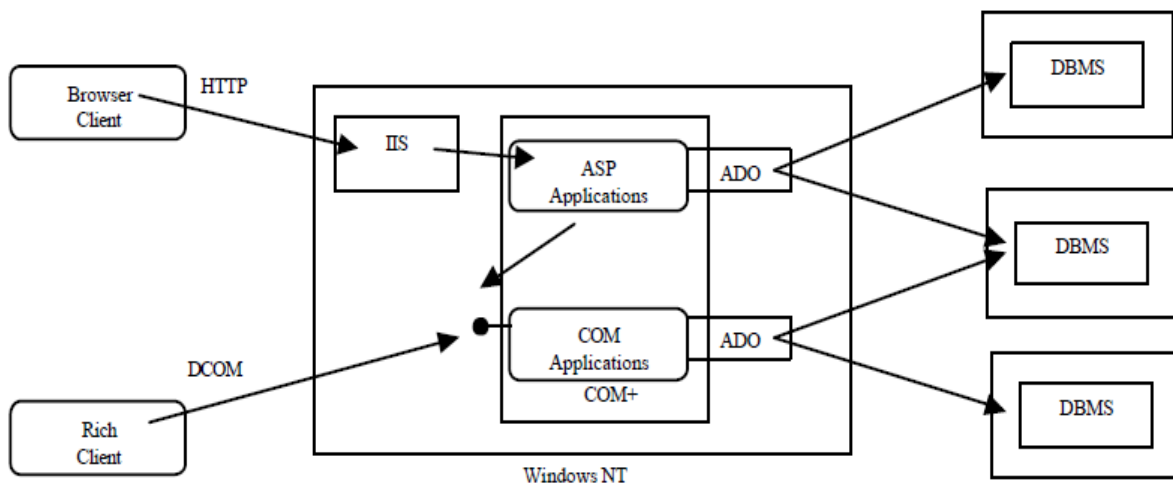
Component Object Model (COM) is a component technology which is a specification and set of services that allows us to create software components and then linking these components together to form reusable, object-oriented, language-independent and upgradable applications. Since COM components are language independent therefore, these can be written in any programming language. COM components are similar to normal objects and it promotes the Object Oriented technique of polymorphism, encapsulation and inheritance. COM components consist of COM interfaces, a system for registering and passing messages between components via these interfaces. COM interfaces are defined in Microsoft Interface Description Language (MIDL).

A COM component can implement and expose multiple interfaces. A client uses COM to locate the server components and then it queries for the wanted interfaces.

By defining interfaces as unchangeable units, COM solves the interface versioning problem. Each time a new version of the interface is created a new interface will be added instead of changing the older version. A basic COM rule is that you cannot change an interface when it has

been released. This makes couplings between COM components very loose and it is easy to upgrade parts of the system indifferent from each other.

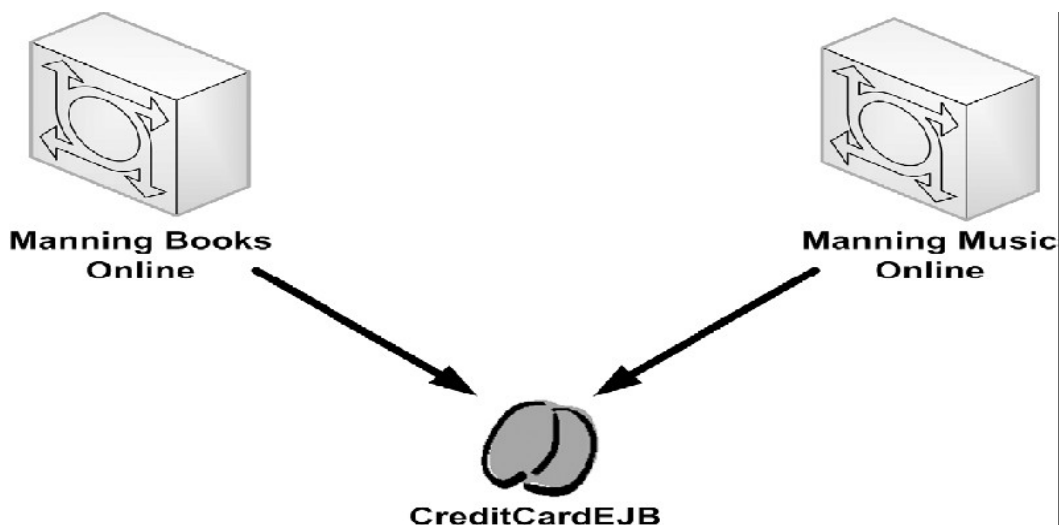
DCOM is the protocol that is used to make COM location transparent. A client talks to a proxy, which looks like the server and manages the real communication with the server. COM+ is an extension to COM with technologies that support among others: transactions, directory service, load balancing and message queuing. Figure below shows how clients can connect, through an Internet Information Server (IIS) or DCOM, to the business logic, which is implemented with COM+. The business logic uses ActiveX Data Objects (ADOs) to access the data in the



databases.

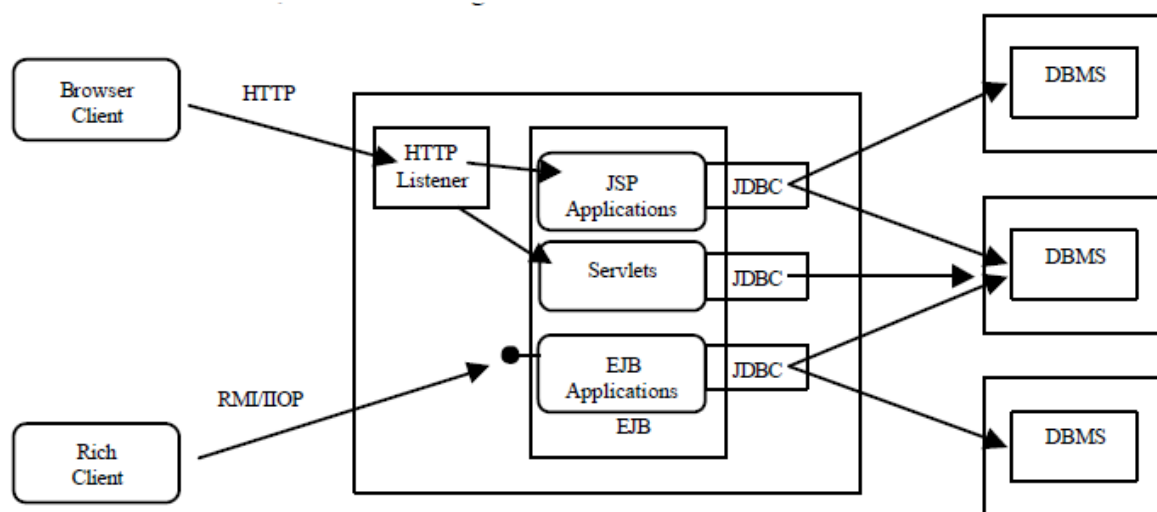
3.5 EJB

Enterprise JavaBeans (EJB) is a component model for building portable, reusable and scalable business components using the Java programming language for distributed environment. In the industry, Enterprise JavaBeans has been applied for component development environment as best solution. EJB allows development of reusable components. For example, to implement the credit card charging module as an EJB component that may be accessed by multiple applications



as shown
in the
figure
below

EJB is part of the Java 2 Platform Enterprise Edition (J2EE) which includes many other technologies remote method invocation (RMI), naming and directory interface (JNDI), database connectivity (JDBC), Server Pages (JSPs) and Messaging services (JMS). Figure below shows the architectural style of EJB used in a three-tier application. The clients connect to the server components through either a web server or directly using remote method invocation (RMI). The server components that implement the business logic reside within an EJB container with the support for transactions and security. The data is stored in databases, which are managed with some database management service (DBMS) and is accessed through the data base connectivity component (JDBC). Java server pages (JSP) or servlets are used when the thin web clients access the system through the Internet.



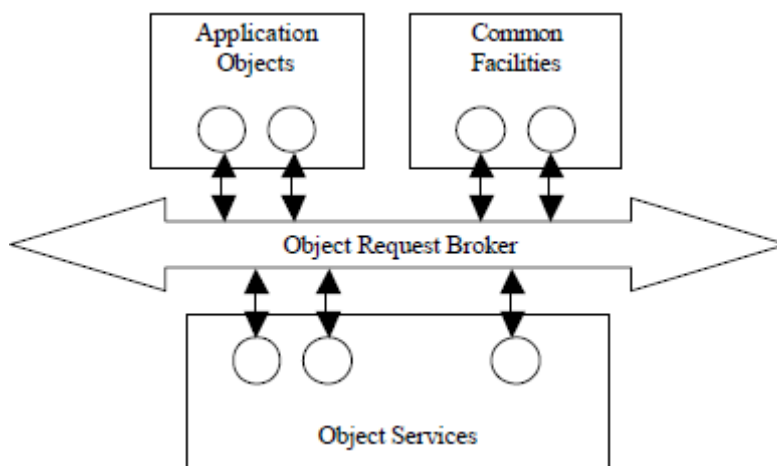
To make a JavaBean an Enterprise bean the JavaBean has to conform to the specification of EJB by implementing and expose a few required methods. These methods allow the EJB container to manage beans in a uniform way for creation, transactions etc. A client to an enterprise bean can virtually be anything, for example a servlet, applet or another enterprise bean. Since enterprise beans may call each other, and then a complex bean task might be divided into smaller tasks and handled by a hierarchy of beans. This is a powerful way of “divide and conquer”. There are two different kinds of enterprise beans: Session and entity beans. Session beans live as long as the client code that calls it. Session beans represent the business process and are used to implement

business logic, business rules and workflow. EJB is designed so it can run together with CORBA and access CORBA objects easily.

3.6 CORBA

The Common Object Request Broker Architecture (CORBA) is a standard that has been developed by the Object Management Group (OMG) in the beginning of the nineties. The OMG provides industry guidelines and object management specifications to supply a common framework for integrating application development. Primary requirements for these specifications are reusability, portability and interoperability of object based software components in a distributed environment. CORBA is part of the Object Management Architecture (OMA) which covers object services, common facilities and definitions of terms.

Object services are for instance naming, persistency, events, transactions and relationships. These can be used when implementing applications. Common facilities provide general-purpose services like information, task and system management. All services and facilities are specified in IDL. An object request broker (ORB) provides the basic mechanism for transparently making requests and receiving responses from objects located locally or remotely. Requests can be made through the ORB without regard to the service location or implementation. Objects publish their interfaces using the Interface Definition Language (IDL) as defined in the CORBA specification. Structure of CORBA is as shown below.



3.7 Component collaborative development process

This section describes the differences of how to develop components and how to develop with components. It is important to do this distinction to make it clear how to use different methods. The developer of a component has to think about how to make the component open for integration with other components and not so much about how to integrate other components.

The development cycle of a component-based system is different from the traditional ones. For instance the waterfall, iterative, spiral and prototype based models. Development with components differs from traditional development. There is a new development process for CBSE and it differs from the traditional waterfall model.

Gathering requirements and design in the waterfall process corresponds to finding and selecting components. Implementation, test and release correspond to create, adapt, deploy and replace.

The different steps in the component development process are:

1. Finding components that may be used in the product. Here all possible components are listed for further investigation.
2. Select the components that fit the requirements of the product.
3. Create a proprietary component that will be used in the product. We do not have to find these types of components since we develop them ourselves.
4. Adapt the selected components so that they suit the existing component model or requirement specification. Some component needs more wrapping than others.
5. Compose or deploy the product. This is done with a framework or infrastructure for components.
6. Replace old versions of the product with new ones.

This is also called maintaining the product. There might be bugs that have been fixed or new functionality added. All the different steps have different technologies to support the developers.

When developing and designing components, we recommend the following advises:

- Always document all the features of the component. Do not restrict the documentation to functionality and document all other properties as well.
- Provide test-suites with the component so that the customer can use it, test it in their environment. It is extremely important to test an imported component in the environment it will operate.
- Provide source code if possible, it might help the application developer to understand the semantics of your component.
- Make the components so they easily integrate into existing component frameworks. Describe what frameworks the component work with and describe how to make it work with other frameworks as well.
- Components need to be carefully generalized to enable reuse in a variety of contexts.

However, solving a general problem rather than a specific one takes more work. Make sure that the application developers can adopt the component to their requirements. This can be done with sink interfaces where the user adds its own interface to the component so that the component can use that interface to communicate with the user.

3.8 Chapter Questions

1. Explain the importance of component based technologies in the development process for software production.
2. Explain the following component based technologies: COM, EJB, CORBA; in addition explain how each used in the software development process

LESSON FOUR

Middle ware enterprise development and service oriented programming

Learning outcomes: at the end of the lesson, the learner should be able to:

- Define what is middle ware enterprise
- To explain the importance of software engineering in middle ware enterprise development
- Differentiate between enterprise logic and business logic used in middle ware development
- Explain the processes and procedures used in middle ware enterprise development.
- To explain how service oriented programming is achieved and the various ways to implement service oriented programming in software engineering.

4.1 Introduction to middle ware enterprise development.

The term *enterprise application* applies to a large class of applications that perform important business functions, such as planning enterprise resource usage, automating key business functions, and managing supply chains and customer relationships. Examples of enterprise applications include airline reservation systems, bank asset management systems, and just-in-time inventory control systems.

Middleware is reusable software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware. Its primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure to coordinate how parts of applications are connected and how they interoperate.

In particular, component middleware offers enterprise application developers the following reusable capabilities:

- Horizontal infrastructure services, such as request brokers.
- Vertical models of domain concepts, such as common semantics for higher-level reusable component services.
- Connector mechanisms between components, such as remote method invocations or message passing.

Various technologies, such as OSF's Distributed Computing Environment (DCE) (www.opengroup.org/dce),

IBM's MQ Series (www-3.ibm.com/software/ts/mqseries/), and CORBA, emerged over the past two decades to alleviate complexities associated with developing software for enterprise applications. Their successes have added the middleware paradigm to the familiar operating system, programming language, networking, and database offerings used by previous generations of software developers. By decoupling application-specific functionality and logic from the accidental complexities inherent in the infrastructure, middleware enables application developers to concentrate on programming application-specific functionality, rather than wrestling repeatedly with lower-level infrastructure challenges.

4.2 Importance of middle ware development in enterprise system development

Despite advances in the ubiquity and quality of component middleware, however, developers of enterprise applications still face the following challenges:

Proliferation of middleware technologies: Large-scale, long-lived enterprise applications require component middleware platforms to work with heterogeneous platforms and languages, interface with legacy code written in different languages, and interoperate with multiple technologies from many suppliers. However, COTS component middleware technologies do not yet provide complete end-to-end solutions that support enterprise application development in diverse environments.

Satisfying multiple quality's of service requirements simultaneously.

An increasing number of enterprise applications, such as high-volume e-commerce systems and automated stock trading systems, have stringent quality of service (QoS) demands, such as efficiency, scalability, dependability, and security, that must be satisfied simultaneously and that cross-cut multiple layers and require end-to-end enforcement.

Accidental complexities in assembling components: To reduce lifecycle costs and time-to-market, application developers are attempting to assemble and deploy enterprise applications by selecting the right set of compatible COTS components, which in itself is a daunting task. The problem is further exacerbated by the existence of myriad strategies for configuring and deploying the underlying component middleware. Application developers therefore spend non-trivial amounts of time debugging problems associated with the selection of incompatible strategies and components.

A promising way to address the challenges described above is to apply *Model-Integrated Computing* (MIC) technologies. MIC is a paradigm for expressing application functionality and QoS (Quality of Service) requirements at higher levels of abstraction than is possible with programming languages like Visual Basic, Java, C++, or C#.

In the context of enterprise applications, MIC tools can be applied to

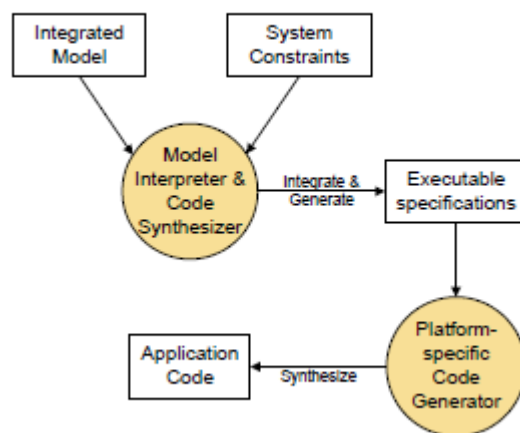
1. Analyze different—but interdependent—characteristics of system behavior, such as scalability, safety, and security. Tool-specific model interpreters translate the information specified by models into the input format expected by analysis tools. These tools check whether the requested behavior and properties are feasible given the constraints.
2. Synthesize platform-specific code that is customized for specific component middleware and enterprise application properties, such as isolation levels of a transaction, recovery strategies to handle various runtime failures, and authentication and authorization strategies modeled at a higher level of abstraction.

4.3 Model Integration Computing

Model-Integrated Computing (MIC) is a development paradigm that systematically applies domain-specific modeling languages to engineer computing systems ranging from small-scale real-time embedded systems to large-scale enterprise applications. MIC provides rich, domain-specific modeling environments, including model analysis and model-based program synthesis tools. In the MIC paradigm, application developers model an integrated, end-to-end view of the

entire application, including the interdependencies of its components. Rather than focusing on a single, custom application, therefore, MIC models capture the essence of a class of applications. MIC also allows the modeling languages and environments themselves to be modeled by so-called *meta-models*, which help to synthesize domain-specific modeling languages that can capture the nuances of domains they are designed to model.

As shown in Figure below MIC uses a set of tools to analyze the interdependent features of the system captured in a model and determine the feasibility of supporting different QoS requirements in the context of the specified constraints. Another set of tools then translates models into executable specifications that capture the platform behavior, constraints, and interactions with the environment. These executable specifications in turn can be used to synthesize application software.



4.4 Integrating MIC with component middleware

MIC and component middleware have evolved independently from different perspectives. Although these two paradigms have achieved good success independently, each has the following limitations:

- **Complexity due to heterogeneity.** Conventional component middleware is developed using separate tools and interfaces written and optimized manually for each middleware specification, such as CORBA, J2EE, and .NET, and for each target deployment, such as the various OS, network, and hardware configurations. Developing, assembling, validating, and evolving *all* this middleware manually is costly, time-consuming, tedious, and error-prone, particularly for runtime platform variations and complex application use-cases. This problem is getting worse as more middleware, target platforms, and complex enterprise applications continue to emerge.
- **Lack of sophisticated modeling tools:** Previous efforts at model-based development and code synthesis attempted by CASE tools generally failed to deliver on their potential for the following reasons [10]:
 - __ They attempted to generate entire applications, including the infrastructure and the application logic, which often lead to inefficient, bloated code that was hard to optimize, validate, evolve, or integrate with legacy code.
 - __ Due to the lack of sophisticated domain-specific languages and associated modeling tools, it was

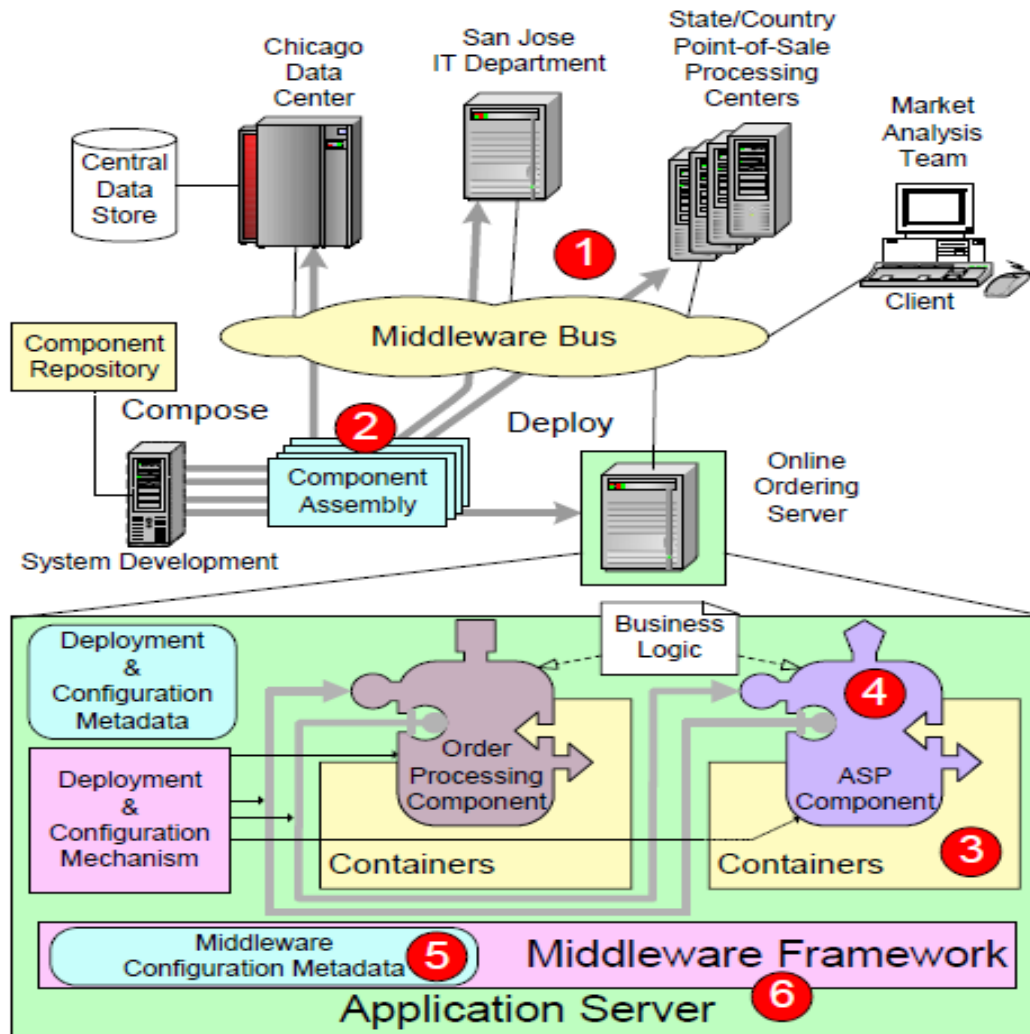
hard to achieve round-trip engineering, *i.e.*, moving back and forth seamlessly between model representations and the synthesized code.

- Since CASE tools and modeling languages dealt primarily with a restricted set of platforms (such as mainframes) and legacy programming languages (such as COBOL) they did not adapt well to the distributed computing paradigm that arose from advances in PC and Internet technology and newer object-oriented programming languages, such as Java, C++, and C#.

The limitations with Model-Integrated Computing and component middleware outlined above can largely be overcome by integrating them as follows:

- Combining MIC with component middleware helps to overcome problems with earlier-generation CASE tools since it does not require the modeling tools to generate all the code. Instead, large portions of applications can be *composed* from reusable, pre-validated middleware components, as shown in the figure below
- Combining MIC and component middleware helps address environments where business procedures and rules change at rapid pace, by synthesizing and assembling newer extended components that conform to new business rules.
- Combining component middleware with MIC helps to make middleware more flexible and robust by automating the configuration of many QoS-critical aspects, such as concurrency, distribution, transactions, security, and dependability. Moreover, MIC-synthesized code can help bridge the interoperability and portability problems between different middleware for which standard solutions do not yet exist.
- Combining component middleware with MIC helps to model the interfaces among various components in terms of standard middleware, rather than language-specific features or proprietary APIs.
- Changes to the underlying middleware or language mapping for one or many of the components modeled can be handled easily as long as they interoperate with other components. Interfacing with other components can be modeled as constraints that are validated by model checkers.

Figure below illustrates six points at which Model-Integrated Computing can be integrated into component middleware architectures. We describe each of these six integration points below:



- 1 Configuring and deploying an application services end-to-end
- 2 Composing components into application server components
- 3 Configuring application component containers
- 4 Synthesizing application component implementations
- 5 Synthesizing middleware-specific configurations
- 6 Synthesizing middleware implementations

1. Configuring and deploying application services end-to end: Developing large-scale enterprise applications requires application developers to handle a variety of configuration and deployment challenges, such as

- Locating the appropriate existing services

- _Partitioning and distributing business processes and
- _Provisioning the QoS required for each service that comprises an application end-to-end.

Integrating MIC and component middleware to deploy application services end-to-end can help application developers configure the right set of services into the right part of an application in the right way.

2. Composing components into application servers. Integrating MIC with component middleware provides capabilities that help application developers to compose components into application servers by

- _Selecting a set of suitable, semantically compatible components from reuse repositories.
- _Specifying the functionality required by new components to isolate the details of business systems that
 - Operate in environments where business processes change periodically and/or
 - Interface with third-party software associated with external information systems.
- Determining the interconnections and interactions between components in metadata.
- Packaging the selected components and metadata into an assembly that can be deployed into the application server.

3. Configuring application component containers. Application components use containers to interact with the application servers in which they are configured. Containers provide many policies that enterprise applications can use to fine-tune underlying component middleware behavior, such as its security, transactional, and quality of service properties.

4. Synthesizing application component implementations: Developing enterprise applications today involves programming new components that add application-specific functionality. Likewise, new components must be programmed to interact with external information systems, such as supplier ordering systems that are not internal to the application. Since these components involve substantial knowledge of application domain concepts, such as government regulations, business rules, organizational structure, and legacy systems, it would be ideal if they could be developed in conjunction with end-users or business domain experts, rather than programmed manually in isolation by software developers.

5. Synthesizing middleware-specific configurations. The infrastructure middleware technologies used by component middleware provide a wide range of policies and options to configure and tune their behavior. For example, CORBA ORBs often provide the following options and tuning parameters:

- _Various types of transports and protocols
- _Various levels of fault tolerance
- _Middleware initialization options
- _Efficiency of (de)marshaling event parameters
- _Efficiency of de-multiplexing incoming method calls
- _Threading models and thread priority settings and
- _Buffer sizes, flow control, and buffer overflow handling

Certain combinations of the options provided by the middleware may be semantically incompatible when used to achieve multiple QoS properties.

6. Synthesizing middleware implementations. Model Integrated Computing can also be integrated with component middleware by using MIC tools to generate custom middleware implementations. This is a more aggressive use of modeling and synthesis than integration point 5 described above since it affects middleware *implementations*, rather than their configurations. Application integrators could use these capabilities to generate highly customized implementations of component middleware so that _ it only includes the features actually needed for a particular application and _ It is carefully fine-tuned to the characteristics of particular programming languages, operating systems, and networks.

4.5 Service oriented programming and its application in software engineering.

A service is a contractually defined behavior that can be implemented and provided by any component for use by any component, based solely on the contract.

Service-Oriented Programming is a paradigm for distributed computing that supplements Object Oriented Programming. Whereas OOP focuses on what things are and how they are constructed, SOP (service oriented programming) focuses on what things can do. The table below gives examples of some service oriented technologies and their dependencies

Initiative	Focus	Dependencies
NET™	Language-Independent Services	SOAP, IP, SCL, XML, DISCO, WINTEL, HTTP
Cooltown™	User / Service Interaction	HTTP, IP
Jini™	Platform-Independent Service Discovery	Java, HTTP, IP
Openwings™	Service-Oriented Programming	Java, HTTP, IP

4.5.1 Service oriented programming elements

The analysis of several Service-Oriented technologies has yielded a set of common architectural elements that make up Service-Oriented Programming:

- Contract – An interface that contractually defines the syntax and semantics of a single behavior.
- Component – A third-party deployable computing element that is reusable due to independence from platforms, protocols, and deployment environments.
- Connector – An encapsulation of transport-specific details for a specified contract. It is an individually deployable element.

- Container – An environment for executing components that manages availability and code security.
- Context – An environment for deploying plug and play components, that prescribes the details of installation, security, discovery, and lookup.

4.5.2 Aspects of service oriented programming

There are also several architectural aspects that characterize service-oriented computing:

- Conjunctive - This refers to the ability to use or combine services in ways not conceived by their originators. This implies that components have published interfaces for the services they provide
- Deployable – This refers to the ability to deploy or reuse a component in any environment. This requires transport independence, platform independence, and environment independence.
- Mobile – This refers to the ability to move code around the network. This is used to move proxies, user interfaces, and even mobile agents. This is the key enabler for interoperability.
- Available – One of the premises of Service-Oriented Programming is that redundant networked resources can provide high availability. It is the goal of SOP to handle the failures that plague distributed computing.
- Secure – The concepts of mobile code and network discoverable services provide new challenges for security. While SOP allows services to have a much broader range of use, it cannot succeed without protecting these services from misuse.
- Interoperable – Interoperability is the ability of components from different sources to use each other's services. SOP supports this through two key features: code mobility and contracts. Interoperability is achieved by sending code across the network that complies with the contract. This code could be a proxy or even an entire application.

4.5.3 Benefits of Service Oriented Programming

Advantages of using service oriented programming can be summarized in the following table below

<i>Key benefits of Service-Oriented Computing</i>	
For developers	For users
<ul style="list-style-type: none"> • Truly reusable components • Simplified system integration • Extensible Systems 	<ul style="list-style-type: none"> • Available systems • Secure data • Plug and Play • Zero-administration • Accessibility of services

4.6 Chapter Questions

1. Explain how middleware enterprise development is accomplished in the software engineering process.
2. Explain how MIC is accomplished in software engineering and how one can't integrate it with component based technologies like EJB, CORBA or COM.

LESSON FIVE

Software validation and verification

Lesson objectives: at the end of the lesson, the learner should be able to:

- Differentiate between software validation and software verification
- State the importance/need of validating software and verifying it too.
- Explain the procedure undertaken to validate and verify software plus the various tools that can be used to do the two tasks.
- Software testing techniques that can be employed to verify and validate your developed software

5.1 Introduction to software validation and verification

Software validation and verification are terms used in software engineering; they do mean different things when they are looked at a whole. Software validation means that the software is checked to see whether it can perform up to standards in the environment for which it was developed for. While software verification means that the software is checked to ensure that all the user requirements are captured correctly and are implemented in the software.

There are two main techniques that maybe used to do both software validation and verification and these are inspections and tests.

5.2 The difference between validation and verification of software.

The difference between these two is that validation checks for the overall performance of the software and if it is able to operate in the environment of the business while verification is to check whether the user's requirements were captured.

5.2.1 Where is each applicable in the development process of the software

The purpose with both inspections and tests is to identify defects in the software. Inspections, or static verification, are used to check and analyze representations of the software component or system such as specifications, models and source code. Testing, or dynamic verification and validation, involves executing and examining an implementation of the software component or system. Verification and validation should be performed continuously during the development of a software component or system.

5.3 Introduction to software testing

Testing of software components (and component based software systems) is possibly the single most demanding aspect of component technology. When a software component is developed all future usage of it cannot be known and thereby not tested, and when the software component is integrated into a system it should already be tested and ready for use. The possibility to reuse pre-made and tested components is the core of the component technology.

Software testing is the activity of executing software to determine whether it matches its specifications and executes in its intended environment.

5.4 Testing procedures and strategies that can be used to test the software

Testing can be grouped depending of the focus and level of details of the test. *White box tests* verify the details of the software, how it is designed and implemented. *Black box tests* verify the functionality and quality of the software without consideration of its implementation. *Live tests* validate the software behavior under operational conditions.

White and black box tests are primarily done to identify faults while live tests are primarily done to prove that the software works as intended during operation.

A **testing strategy** is a general approach to the testing process rather than a method of devising particular system or component tests. Different testing strategies may be adopted depending on the type of system to be tested and the development process used. There are two different strategies available: **Top-Down Testing** and **Bottom-Up Testing**.

In **Top-Down Testing**, high levels of a system are tested before testing the detailed components. The application is represented as a single abstract component with sub-components represented by **stubs**. Stubs have the same interface as the component but very limited functionality.

After the top-level component has been tested, its **sub-components** are implemented and tested in the same way. This process continues recursively until the bottom - level components are implemented. The whole system may then be completely tested. Top-down testing should be used with **top-down program development** so that a system component is tested as soon as it is coded. Coding and testing are a single activity with no separate component or module testing phase.

If top-down testing is used, unnoticed **design errors** may be detected at an early stage in the testing process. As these errors are usually structural errors, early detection means that extensive re-design re-implementation may be avoided. Top-down testing has the further advantage that we could have a **prototype system** available at a very early stage, which itself is a psychological boost. Validation can begin early in the testing process as a demonstrable system can be made available to the users.

Bottom-Up Testing is the opposite of Top-Down. It involves testing the modules at the lower levels in the hierarchy, and then working up the hierarchy of modules until the final module is tested. This type of testing is appropriate for **object-oriented systems** in that individual objects may be tested using their own test drivers. They are then integrated and the object collection is tested.

5.5 Different stages of testing for the software

The most widely used process of testing software consists of **five stages**:

1. **Unit Testing:** Individual components are tested to ensure that they operate correctly. Each component is tested independently without other system components.
2. **Module Testing:** This involves the testing of independent components such as procedures and functions. A module encapsulates related components so it can be tested without other system modules.
3. **Subsystem Testing:** This phase involves testing collections of modules which have been integrated into sub-systems. Sub-systems may be independently designed. The most common problems which arise in large software systems are sub-system interface mismatches. The sub-system test process should therefore concentrate on the detection of interface errors by rigorously exercising the interfaces.
4. **System Testing:** Sub systems are integrated to make up the entire system. The testing process is concerned with finding errors that result from unanticipated interactions between sub-systems and system components. It is also concerned with validating that the system meets its functional and non-functional requirements.
5. **Acceptance Testing:** This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system procurer rather than simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition because the real data exercises the system in different ways from the test data. It may also reveal requirements problems where the system's facilities do not really meet the user's needs or the system's performance is not acceptable.

5.6 Importance of validating and verifying software.

Developing software components that can be reused, commercialized requires a high focus on software quality. In mature software development organizations high quality can be ensured by means of an appropriate defined development process and appropriate methods and tools, that is defect prevention. High quality can also be ensured by means of thorough verification and validation, that is defect identification. The quality of the component should be certified in a, for the component purchasers, trusted way.

5.7 Chapter Questions

1. Explain the software validation process, why is it necessary in the software development process.
2. Differentiate between software validation and software verification.
3. Explain the software testing process; in addition explain the software testing strategies.

LESSON SIX

Computer software Project management

Lesson objective: at the end of the lesson, the learner should be able to:

- Explain how to manage the processes involved in developing a new software
- Explain various terminologies and definitions as used in computer software project management
- Explain techniques used in software project management
- Explain the various CASE tools that one can use in software development project.

6.1 Introduction computer software project management

Project management is the discipline of defining and achieving targets while optimizing the use of resources (time, money, people, materials, energy, space, etc) over the course of a project (a set of activities of finite duration). Thus software project management is the discipline of combining project management principles and activities in the designing and development of computer software's.

A project is a temporary endeavor undertaken to create a unique product or service. It's progressively elaborated with repeating activities within it.

A project can be broken down into *phases*, or *tasks* to be done. Each *phase* is defined by its entry criteria, exit criteria, resources, deliverables and reports.

Difference between an activity and a phase are:

Activity: a set of tasks performed towards a specific purpose.

Phase: A set of tasks performed over time; defined by its start and end points

6.2 Activities in Software project management

Activities involved in software project management are as follows:

- 🕒 Requirements Analysis
- 🕒 Product Design
- 🕒 Programming
- 🕒 Test Planning
- 🕒 Verification & Validation
- 🕒 Project Office functions
- 🕒 Configuration management and quality assurance
- 🕒 Manuals

Software Lifecycle Model examples are as listed below:

- 🕒 Waterfall model
- 🕒 Boehm's Risk Spiral model
- 🕒 Prototyping Paradigm
- 🕒 Evolutionary (Incremental) Development
- 🕒 Extreme Programming

6.3 Methodologies in software project management

6.3.1 Rational Unified Process (RUP)

The Rational Unified Process (RUP) is a software design methodology created by the Rational Software Company. The Rational Software Company was acquired by IBM in 2003. RUP is a thick methodology; the whole software design process is described with high detail. RUP is hence particularly applicable on larger software projects. The RUP methodology is general enough to be used out of the box, but the modular nature of RUP is designed and documented using Unified Modeling Language (UML) also takes it easy to adapt the methodology to the special needs of a single project or company. One of the major differences between RUP and other methodologies like SSADM (see Section 3.2) is that RUP doesn't use a waterfall approach for software development. The phases of requirements, analysis, design, implementation, integration and testing are not done in strict sequence. In RUP, an iterative approach is used: a software product is designed and built in a succession of incremental iterations. Each iteration includes some, or most, of the development disciplines (requirements, analysis, design, and implementation, testing, and so on).

Due to the modular nature of RUP, it can be used for all sorts of software projects. It is even possible to use RUP for non-software projects. However, because of the complexity of the RUP methodology, it is used mostly for larger software projects.

ADVANTAGES

- The iterative approach leads to higher efficiency. Testing takes place in each iteration, not just at the end of the project life cycle. This way, problems are noticed earlier, and are therefore easier and cheaper to resolve. When using a waterfall approach, it can happen that, for example, software programmers have to wait for the completion of the design phase before starting implementing and integrate the design.
- Managing changes in software requirements will be made easier by using RUP. Unless a software project is very small, it is nearly impossible to define all the software requirements at the beginning of a project. It will almost always take more than one step to know what the final software product will look like, for the customer as well as for the project members.
- RUP itself is software, too, and is distributed in an electronic and online form. Team members don't need to leave their computers for RUP related activities. No more searching in big, dusty books. All information about the software development methodology is available at the project members' fingertips.

Disadvantages

- RUP is a commercial product, no open or free standard. Before RUP can be used, the RUP has to be bought from IBM, as an electronic software and documentation package (a trial version can be downloaded from the IBM website, however). The RUP only exists in an electronic form, which can sometimes limit its use.
- RUP, as said before, describes the whole software design process with high detail; it is a very complex methodology, difficult to comprehend for both project managers and project members. Therefore, it is not the most appropriate software design methodology for most small projects.
- Starting to use RUP as software development methodology is difficult. Everyone participating in the project will have to learn working with RUP. For details about applying RUP on projects, see [RUP-TRANS].

6.3.2 SSADM

SSADM is an open standard, which means that it is freely available for use in industry and many companies offer support, training and Computer Aided Software Engineering (CASE) tools for it. In detail, SSADM sets out a cascade or waterfall view of systems development.

SSADM revolves around the use of three key techniques:

1. Logical Data Modeling (LDM): This is the process of identifying, modelling and documenting the data requirements of a business information system. A LDM consists of a LDS and the associated documentation. LDSs represent Entities (things about which a business needs to record information) and relationships (necessary associations between entities).
2. Data Flow Modeling (DFM): This is the process of identifying, modeling and documenting how data flows around a business information system. A Data Flow Model consists of a set of integrated DFDs supported by appropriate documentation. DFDs represent processes (activities which transform data from one form to another), data stores (holding areas for data), external entities (things which send data into a system or receive data from a system and finally data flows (routes by which data can flow).
3. Entity/Event Modeling (EM): This is the process of identifying, modeling and documenting the business events which affect each entity and the sequence in which these events occur. An EM consists of a set of Entity Life History's (ELHs) (one for each entity) and appropriate supporting documentation.

Advantages

- SSADM divides an application development project into modules, stages, steps, and tasks, and provides a framework for describing projects in a fashion suited to managing the project.
- SSADM can reduce the chances of initial requirements being misunderstood and of the systems functionality straying from the requirements through the use of inadequate analysis and design techniques.

Disadvantages

- SSADM assumes that the requirements (in the form of an agreed requirements specification) will not change during the development of a project. Following each step of SSADM rigorously can be time consuming and there may be a considerable delay

between inception and delivery (which is typically the first time the users see a working system). The longer the development time the more chance of the system meeting the requirements specification but not satisfying the business requirements at the time of delivery.

6.4 Techniques in software project management

These techniques can be used as an aid to estimate, track and evaluate different aspects of the project. They include COCOMO, critical path, MTA, EV among others

6.4.1 Project Management body of Knowledge (PMBOK)

The full Project Management Body of Knowledge (PMBOK) includes knowledge of proven traditional practices that are widely applied, as well as knowledge of innovative and advanced practices that have seen more limited use, and includes both published and unpublished material [PMBOK-PMI]. The PMBOK framework splits the project processes into five distinct process groups: initiating, planning, executing, controlling and closing. Note that these groups do not imply that the project has to go through each one in this order; they are only provided in order to be able to structure and categorize the different project processes. PMBOK also identifies several project knowledge areas: integration management, scope management, time management, cost management, quality management, human resource management, communications management, risk management and procurement management.

Advantages

- PMBOK provides a general project management framework in the form of process groups and knowledge areas.
- PMBOK gives a concise summary of and reference to generally accepted project management principles.
- PMBOK proposes a unified project management terminology.

Disadvantages

- PMBOK is only a framework; the actual needs of the project in question should be determined by a knowledgeable managerial team.
- PMBOK provides minimal coverage of various project management methodologies and techniques. One definitely needs to consult specialized texts on these subjects in order to learn the ins and outs.
- PMBOK only covers those aspects of the project management process that are profession independent.

6.4.2 COCOMO

COCOMO is an empirical, algorithmic model for estimating the effort, schedule and costs of a software project. It was derived by collecting relevant data from a large number of software projects, then analyzing the data to discover the formulae that were the best-fit to the observations.

The first version of the COCOMO model (now known as COCOMO 81) was a three-level model where the levels showed the detail of the analysis of the cost estimate. The first level (basic)

provided an initial, rough estimate; the second level modified this using a number of project and process multipliers and the most detailed level produced estimates for different phases of the project

Advantages

- Although it's hard to pinpoint the exact cost of any given project, one can still obtain usable data by calculating optimistic and pessimistic estimates.
- Implementation and execution of the model is very simple and efficient. As a result, it is supported by public as well as commercial tools.
- COCOMO is a well-known and well-documented technique.

Disadvantages

- It is quite difficult to come up with satisfactory estimates for the size of a project when the latter still in an early stage of development.
- The use of the number of lines of source code as a measure of complexity is highly disputable. Even though COCOMO tries to take this into account by providing different tables for all major programming languages, there are still lots of inconsistencies such as: expressivity differences between programmers, usage of subroutines, general code reuse, etcetera.
- Several input parameters in the model cannot be determined quantitatively; they need to be estimated as well.

6.4.3 Milestone Trend Analysis

MTA is a software engineering technique for evaluating the actual progress of a project in relation to its planning. This relatively simple technique consists of recording the dates of the milestone deadlines at the times they are changed, i.e. when they are postponed or advanced. This way one gets a matrix of data: the columns of the matrix delimit the project milestones, the rows the dates on which the deadlines were reevaluated, while an actual cell contains the new deadline estimate for the milestone in question.

MTA can be applied to every project that uses milestones as the major indicators of progress. It is in essence a very simple and elegant technique that can easily be applied to assess progress.

Advantages

- MTA is a simple, elegant and effective technique.
- MTA is widely used and supported.
- MTA has a large application area.

Disadvantages

- MTA in itself does not keep track of inter-package dependencies. Therefore, when a certain milestone completion date is altered, one needs to make sure its dependencies are altered as well. This does not prove to be much of a problem in practice however, since MTA is available as a plugin for more comprehensive project management tools that can keep track of dependencies.

- The inputs of the MTA technique are of course estimates of milestone completion deadlines. As such, it is imperative these estimates are made by knowledgeable and experienced engineers. MTA will not be of much use if these estimates are not reasonably accurate.

6.4.4 Earned value management (EV)

In earned value management the progress of a project is estimated by comparing what already has been done with the estimates that were made at the beginning of a project. By extrapolating these measurements, a project manager can judge how much resources will be used at the end of a project.

Some common acronyms that are used in the EV management:

- BCWS Budgeted Cost for Work Scheduled
- BCWP Budgeted Cost for Work Performed
- ACWP Actual Cost of Work Performed
- BAC Budget At Completion
- EAC Estimate At Completion

EV indicators

- Cost Variance: $CV = BCWP - ACWP$: This shows the difference between the budgeted cost for a certain amount of work (BCWP) and the real cost of an amount of work (ACWP). A negative number indicates that the cost has been underestimated, while a positive number indicates that the cost has been overestimated
- Schedule Variance: $SV = BCWP - BCWS$: This shows the difference between what has been earned at a certain and what should have been earned.
- Budget Remaining: $BR = BAC - ACWPCumulative$. This shows the amount of budget that is still available to complete the project.
- Work Remaining: $BCWR = BAC - BCWPCumulative$: This shows the amount that still has to be earned in this project, thus the work that remains to be done.
- Variance at Completion: $V_{AC} = BAC - EAC$: This shows the difference between the planned cost at the end and the estimated cost at the end. A negative value indicates that the project is costing more than planned and a positive value indicates that it's costing less.
- Cost Performance Index (Efficiency): $CPI_e = BCWP/ACWP$: This shows how efficient the project is being done in terms of cost. For example a value of 2 shows that the project is currently costing half of the amount planned. Or in other words it's being done twice as efficiently as estimated.
- Schedule Performance Index (Efficiency): $SPI_e = BCWP/BCWS$: This shows how efficient the project is being done in terms of time. For example a value of 2 shows that the project is going twice as fast as estimated.
- Estimate At Completion: $EAC = BAC/CPI_e$: This gives a prediction what the cost will at the end of the project. The equivalent but longer version $EAC = ACWP + (BAC - BCWP)/CPI_e$ is often used in the literature. It's important to note that CPI_e is a moving

target and changes during the course of the project. Instead of using the CPIe of the whole project, the CPIe of for example the last three months can be used. This takes the current performance of the project better into account. Of course choosing shorter timespans for the CPIe will increase the influence of short periods of peak performance.

Advantages

During a project a manager can judge if the projects are on schedule/budget. If that's not the case an estimate can be made how far the project is over budget.

Disadvantages

It's very difficult to estimate the real Earned Value at a certain point in time. Wrong estimates of this value can make a project look like it's doing a lot better then it really is (or the other way around).

6.5 Chapter Questions

1. Explain the activities involved in the software project management process
2. Why is it necessary to incorporate project management activities in the software development process?

LESSON SEVEN

Object Oriented Software development

Lesson objectives: the learner at the end of the lesson should be able to:

- Use object oriented software development methodology to develop a software
- Know the various diagrams used in modeling a software based on object oriented software methodology

7.1 Introduction to object oriented software development

Object-oriented software design is a design strategy where system designers think in terms of ‘things’ instead of operations or functions. The executing system is made up of interacting objects that maintain their own local state and provide operations on that state information (Figure below). They hide information about the representation of the state and hence limit access to it. An object-oriented design process involves designing the object classes and the relationships between these classes. When the design is realized as an executing program, the required objects are created dynamically using the class definitions.

Figure: System with interacting objects (IAN Sommerville-2005)

Object-oriented analysis and design is more cost-effective and a faster way to develop software and systems. This technology cuts development time, overhead and enables software engineers to make reusable, reliable and easily maintainable applications.

The benefits of object-oriented analysis and design specifically include:

- Required changes are localized and unexpected interactions with other program modules are unlikely;
- Inheritance and polymorphism make OO systems more extensible, contributing thus to more rapid development;
- Object-based design is suitable for distributed, parallel or sequential implementation;

- Objects correspond more closely to the entities in the conceptual worlds of the designer and user, leading to greater seamlessness and traceability;
- Shared data areas are encapsulated, reducing the possibility of unexpected modifications or other update anomalies.

Object-oriented analysis and design methods share the following basic steps although the details and the ordering of the steps vary quite a lot:

- Find the ways that the system interacts with its environment (use cases);
- Identify objects and their attribute and method names;
- Establish the relationships between objects;
- Establish the interface(s) of each object and exception handling;
- Implement and test the objects;
- Assemble and test systems.

Analysis is the decomposition of problems into their component parts. In computing it is understood as the process of specification of system structure and function independently of the means of implementation or physical decomposition into modules or components. Analysis was traditionally done top-down using structured analysis, or an equivalent method based on functional decomposition, combined with separate data analysis.

Object-oriented analysis is analysis, but also contains an element of synthesis. Abstracting user requirements and identifying key domain objects are followed by the assembly of those objects into structures of a form that will support physical design at some later stage. The synthetic aspect intrudes precisely because we are analyzing *a system*, in other words imposing a structure on the domain. This is not to say that refinement will not alter the design; a well-decoupled design can be considerably different from a succinct specification model.

7.2 Use of UML diagrams in O.O software development technique to model/design software.

In OOP paradigm, from the programmer's point of view, an object-oriented language must support three very important explicit characteristics. We use these concepts extensively to model the real-world problems when we are trying to solve with our object-oriented programs. These three concepts are:

- encapsulation
- inheritance
- Polymorphism.

7.3 TERMS AND CONCEPTS

7.3.1 Class and object.

A **class** is a category or group of things that have similar attributes and common behaviors. In the programming context, a class is a template data type from which many copies can be made.

Each of these copies is known as an **object**. An object, therefore, is an instance of a class. Each class has got attributes and member functions **encapsulated** within it.

An object is a thing, an entity, a noun, something you can pick up or kick, anything you can imagine that has its own identity. Some objects are living, some aren't. Examples from the real world include a car, a person, a house, a table, a dog, a pot plant, a check book or a raincoat. All objects have **attributes**: for example, a car has a manufacturer, a model number, a color and a price; a dog has a breed, an age, a color and a favorite toy. Objects also have **behavior**: a car can move from one place to another and a dog can bark.

A data type together with the operations to be performed on instantiation of the data type. Such as construct is called an abstract data type.

An **object** is an instance of a class.

An object is an instantiation (instance) of an abstract data type. Individual objects created from each actual thing.

Characteristics of objects.

Objects have the following characteristics.

- **State**- this is all the data which the object encapsulates(instance variables or data members attributes) each which has a unique value
- **Behavior** – the way an object acts and reacts in terms of state changes and message passing. Objects can receive messages and act on them.
- **Identity**- an object is referred by name.
-

7.3.2 Data Abstraction and Encapsulation

Encapsulation is defined as gathering into one unit all aspects of the real world entity modeled by that unit (conceptual independent).

The wrapping up of data and functions into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions, which are wrapped in the class, can access it. These functions provide the interface between the object's data and the rest of the program. This insulation of the data from direct access by the program is called data hiding or information hiding.

Encapsulation wraps attributes and operations into objects i.e. an object attributes and operations are intimately tied together

This insulation of the data from direct access by the program is called data **hiding or Abstraction**

Abstraction is simply a means of achieving stepwise refinement by suppressing unnecessary details and accentuating relevant details.

Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes.

They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called data members because they hold information. The functions that operate on these data are sometimes called methods or member functions.

Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT)

Data abstraction allows the designer to think at the level of data structures and the operations to be performed on it and only later to be concerned with details of how the data structures and operations are to be implemented.

- **Types of abstraction.**

- Data abstraction**

Data encapsulation (that is data structures together with the operation to be performed on that data structure is an example of data abstraction.

What you can do with the data is separated from *how* it is represented

- Procedural abstraction.**

The designer can conceptualize the product in terms of high level operation. These operations can be defined in terms of low-level operations until the lowest level is reached. At this level the operations are expressed in terms of the predefined constructs of the programming language.

- Information hiding**

Objects have the property of information hiding. This means that objects may know how to communicate with one another across well- defined interfaces but are not allowed to know how objects are implemented- the information details are hidden within the objects themselves.eg the brake pedal.

7.3.4 Access modifiers/rules of encapsulation

Private

Instance variables or modifiers declared with access modifiers is private are accessible only to methods of the class in which they are declared.

Declaring instance variables with access modifier private is known as data hiding or information hiding. This prevents data in a class from being modified accidentally by a class in another part of the program.

Public

A class's public members are accessible whenever the program has a reference to an object of that class or one of its subclasses.

Protected

Protected modifier offers an intermediary level of access between public and private. Protected members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the same package.



7.3.5 Polymorphism and dynamic binding.

Polymorphism

This is a run time characteristics whereby the OO environment select a specific form of behavior from a number of choices.

It is derived from Greek and it has to do with taking many shapes. Polymorphism enables you to program in the general rather than specific.

Relying on each object to know how to do the right thing (i.e. do what is appropriate for that type of object) in response to the same method call is the key concept of polymorphism (animal example)

Polymorphism, means the ability to take more than one form.

An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.

For e.g., consider the operation of addition of two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.

The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

Dynamic binding means attaching a message to a method at run time. This is the way that object-oriented languages cope with polymorphic variables and redefined methods.

- **Inheritance**

Inheritance is the process by which objects of one class acquire the properties of objects of another class. For example, the bird 'robin' is a part of the class "flying bird" which is again a part of the class 'bird'.

The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived.

Inheritance allows us to specify that a class gets some of its characteristics from a **parent class** and then adds unique features of its own – this leads to the description of whole families of objects.

Inheritance allows us to group classes into more and more general concepts, so that we can reason about larger chunks of the world that we live in.

From a programming point of view, we want inheritance because:

- It supports richer, more powerful, modeling. This benefits both the development team and other developers who might want to reuse code.
- It allows us to define information and behavior in one class and share the definitions in related subclasses. This means that we have less code to write.
- It's natural. This is one of the prime motivations for object orientation in the first place.

This is a form of software re-used in which a new class is created by absorbing existing class members and embellishing them with new or modified capabilities.

With inheritance you can save time during program development and also increase the likelihood that system will be implemented and maintained effectively

Inheritance is supposed by all OO languages and the idea behind it is that new data type can be defined as extension of previously defined data type rather having to be defined from scratch.

In OO a class can be defined in an abstract data type that supports inheritance.

An object is an instance of a class.

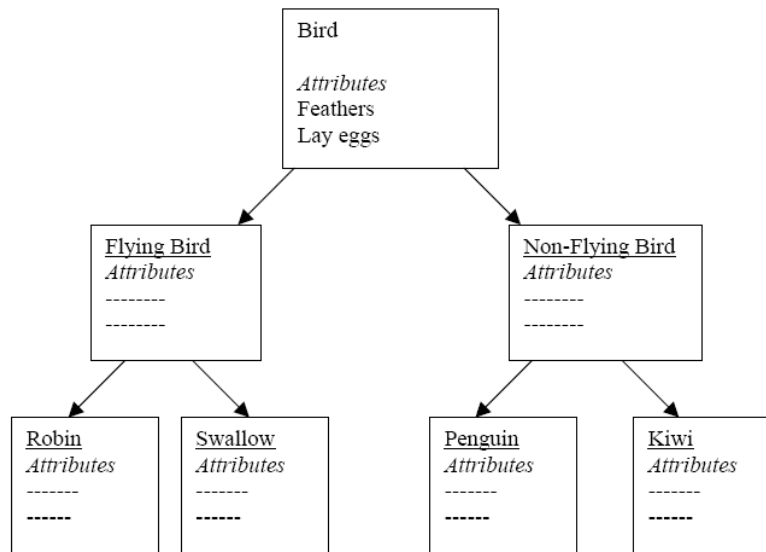


Fig. Property inheritance

Polymorphism

Polymorphism, means the ability to take more than one form.

An operation may exhibit different behaviors in different instances. The behavior depends upon the

Base class and derived class

Base class is the existing class and also called **super class** and the new class is called subclass or derived class

Direct super class is the super class from which the subclass explicitly inherits. Inherited explicitly (one level up hierarchy)

Indirect super class is the class Inherited two or more levels up hierarchy. i.e. the class which defines the inheritance relationship between classes in the inheritance hierarchy.

Superclass typically represents larger set of objects than subclasses

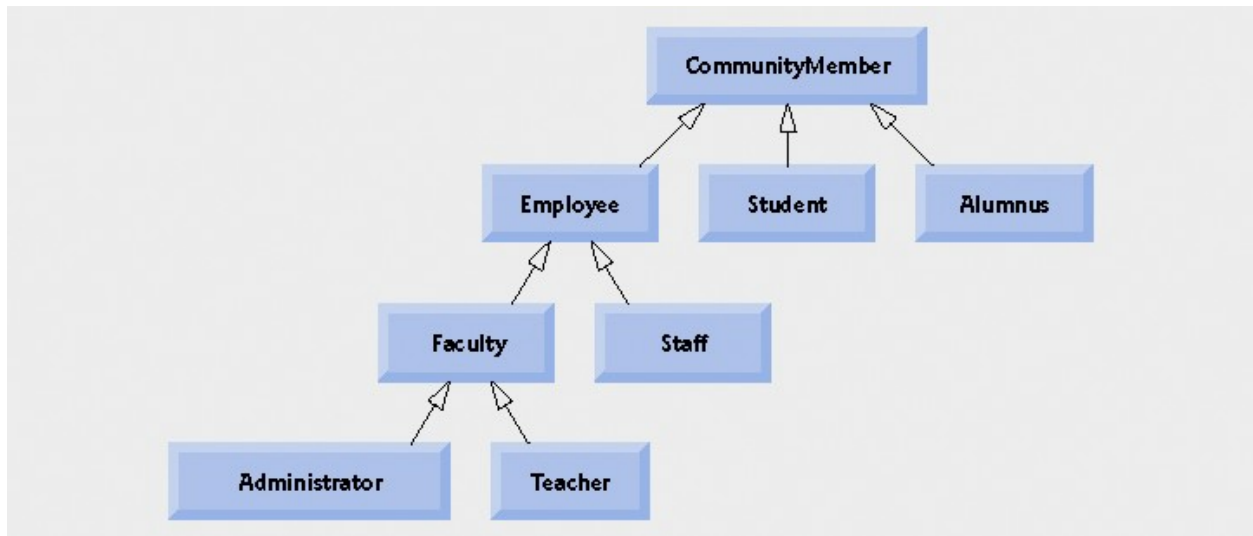
Example: superclass: Vehicle

Cars, trucks, boats, bicycles, ...

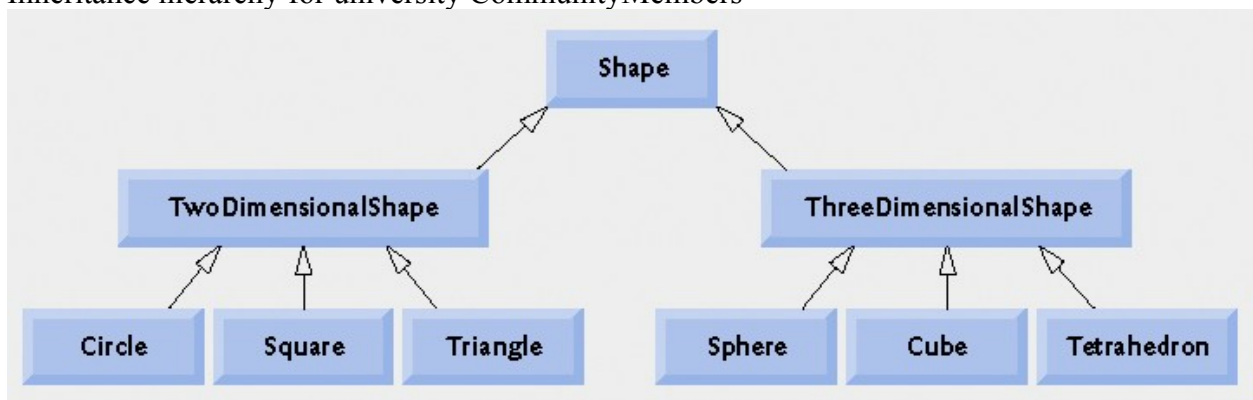
subclass: Car

Smaller, more-specific subset of vehicles

Inheritance hierarchy is an Inheritance relationships: tree-like hierarchy structure where each class becomes superclass when it supply members to other classes or subclass when it Inherit members from other classes



Inheritance hierarchy for university CommunityMembers

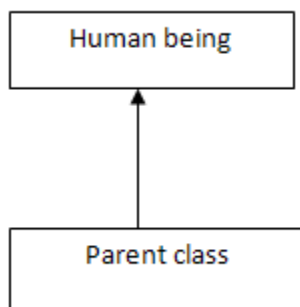


Inheritance hierarchy for Shapes

Each arrow represents is-a relationship- employee is a community member.

- **Generalization and specialization.**

A subclass is more specific than its superclass and represents more specialized group of objects. A subclass exhibits the behaviors of its superclass and can add behaviors that are specific to the subclass.



Parent class is a specialization of **human being** is a generalization of parent class.
Abstract and concrete classes

When we design an inheritance hierarchy, we generalize classes into higher-level abstractions (when moving up the hierarchy) and specialize them into lower-level abstractions or concrete classes (when moving down). As we move up or down, we often have to choose between alternative generalizations and specializations, even though they may all seem equally valid.

o **Abstract class.**

An **abstract class** is a class with at least one abstract method – the abstract method may be introduced on the class itself, or it may be inherited from a superclass.

An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon),

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract.

An abstract class has at least one method without code (an abstract method); in a concrete class, all the methods contain lines of code.

Abstract classes have the following advantages:

- They permit richer and more flexible modeling; for example, our List class has all three messages – contains, elementAt and numberOfElements – despite the fact that we can't provide concrete methods for all of them.
- They lead to more code sharing, because we can write concrete methods that use abstract methods; for example, the contains method for List invokes abstract methods.

abstract class GraphicObject

```
{
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

Each non-abstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize methods:

```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

```

}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}

```

7.4 BENEFITS OF OOP

OOP offers several benefits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

7.5 APPLICATIONS OF OOP

Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as windows. Hundreds of windowing systems have been developed, using the OOP techniques.

Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of applications because it can simplify a complex problem. The promising areas for application of OOP include:

- Real-time systems
- Simulation and modeling
- Object-oriented databases

- Hypertext, hypermedia and experttext
- AI and expert systems
- Neural networks and Parallel programming Decision support and office automation systems

7.6 THE UNIFIED PROCESS.(Unified Modeling Process)

7.6.1 The phases of the unified process.

The phases of unified process correspond to increments. The development consists of four phases (increments).

- Inception
- Elaboration
- Construction
- Transition

The UML Development process has four phases

UML has four project phases:

- Inception Phase - Approximately 20% of requirements determined.
- Elaboration Phase - Approximately 80% of requirements determined.
- Construction Phase
- Transition Phase

(Sometimes 2 other phases are added. Production & Retirement)

Inception Phase

This is the part of the project where the original idea is developed. The amount of work done here is dependent on how formal project planning is done in your organization and the size of the project. During this part of the project some technical risk may be partially evaluated and/or eliminated. This may be done by using a few throw away prototypes to test for technical feasibility of specific system functions. Normally this phase would take between two to six weeks for large projects and may be only a few days for smaller projects. The following should be done during this phase:

1. Project idea is developed.
2. Assess the capabilities of any current system that provides similar functionality to the new project even if the current system is a manual system. This will help determine cost savings that the new system can provide.
3. Utilize as many users and potential users as possible along with technical staff, customers, and management to determine desired system features, functional capabilities, and performance requirements. Analyze the scope of the proposed system.
4. Identify feature and functional priorities along with preliminary risk assessment of each system feature or function.
5. Identify systems and people the system will interact with.
6. For large systems, break the system down into subsystems if possible.

7. Identify all major use cases and describe significant use cases. No need to make expanded use cases at this time. This is just to help identify and present system functionality.
8. Develop a throw away prototype of the system with breadth and not depth. This prototype will address some of the greatest technical risks. The time to develop this prototype should be specifically limited. For a project that will take about one year, the prototype should take one month.
9. Present a business case for the project (white paper) identifying rough cost and value of the project. The white paper is optional for smaller projects. Define goals, estimate risks, and resources required to complete the project.
10. Set up some major project milestones (mainly for the elaboration phase). A rough estimate of the overall project size is made.
11. Preliminary determination of iterations and requirements for each iteration. This outlines system functions and features to be included in each iteration. Keep in mind that this plan will likely be changed as risks are further assessed and more requirements are determined.
12. Management Approval for a more serious evaluation of the project.

This phase is done once the business case is presented with major milestones determined (not cast in stone yet) and management approves the plan. At this point the following should be complete:

- Business case (if required) with risk assessment.
- Preliminary project plan with preliminary iterations planned.
- Core project requirements are defined on paper.
- Major use cases are defined.
- The inception phase has only one iteration. All other phases may have multiple iterations.

Elaboration Phase

This phase has several goals:

- To produce a proven, architectural baseline for your system
- To evolve your requirements model to the "80% completion point"
- To develop a coarse-grained project plan for the entire Construction phase
- To ensure that the critical tools, processes, standards, and guidelines have been put in place for the Construction phase
- To understand and eliminate the high-priority risks of your project

The primary purpose of this phase is to complete the most essential parts of the project that are high risk and plan the construction phase. This is the part of the project where technical risk is fully evaluated and/or eliminated by building the highest risk parts of the project. During this phase personnel requirements should be more accurately determined along with estimated man hours to complete the project. The complete cost and time frame of the project is more firmly determined. During this phase how the system will work must be considered. Use cases will help identify risks. Steps to take during this phase:

1. Complete project plan with construction iterations planned with requirements for each iteration.
2. 80% of use cases are completed. Significant use cases are described in detail.
3. The project domain model is defined. (Don't get bogged down)
4. Rank use cases by priority and risk. Do the highest priority and highest risk use cases first. Items that may be high risk:
 - Overall system architecture especially when dealing with communication between subsystems.
 - Team structure.
 - Anything not done before or used before such as a new programming language, or using the unified/iterative process for the first time.
5. Begin design and development of the riskiest and highest priority use cases. There will be an iteration for each high risk and priority use case.
6. Plan the iterations for the construction phase. This involves choosing the length of the iterations and deciding which use cases or parts of use cases will be implemented during each iteration. Develop the higher priority and risk use cases during the first iterations in the construction phase.

As was done on a preliminary level in the previous phase, the value (priority) of use cases and their respective risks must be more fully assessed in this phase. This may be done by either assigning an number to each use case for both value and risk. or categorize them by high, medium, or low value and risk. Time required for each use case should be estimated to the man week. Do the highest priority and highest risk use cases first.

Requirements to be completed for this phase include:

- Description of the software architecture. Therefore most use cases should be done, activity diagrams, state charts, system sequence diagrams, and the domain model should be mostly complete.
- A prototype that overcomes the greatest project technical risk and has minimal high priority functionality.
- Complete project plan.
- Development plan.

There may be an elaboration phase for each high-risk use case.

Construction Phase

Construction iterations are based on use cases. Small use cases may be done in one iteration or a larger use case may be worked on in sections. For each iteration, analysis, design, and creation of software is performed for each use case in that iteration. Again, much documentation indicates that the domain model (conceptual model) should be done for each construction iteration and be based on use cases being implemented during that specific iteration. The following should be done during the construction iterations:

1. Completion of expanded use case diagrams for use cases.
2. System sequence diagrams for major use cases.

3. Operation contracts based on domain model and use cases.
4. Collaboration diagrams.
5. Class diagrams.
6. Map class and collaboration diagrams to code.
7. Update the domain model but do not force it to the class diagrams.


It is worth performing tests to be sure each use case works properly at the end of the iteration. At the end of the construction phase there is a product that users can use. The following will be complete:

- Users manuals.
- Release version and description.

Transition Phase

During this phase, the finished product is brought to the user. Items to be addressed in this phase include:

- Final program debugging
- Code Optimization
- Beta testing
- Completion of manuals.
- User training

Possible operation in parallel with an existing system 

7.6 Chapter Questions

1. Explain the importance of object oriented programming in software development.
2. Explain the 7 diagrams used in object oriented system design process
3. Explain the UML modeling process, what are the steps involved.

LESSON EIGHT

Computer software Cost estimation.

Lesson objectives: the learner at the end of the lesson should be able to:

- To differentiate the various software cost estimation techniques that can be used to estimate the cost of software.
- Apply these techniques in estimating the cost of software.

8.1: Introduction to software cost estimation

Software cost estimation is the process of predicting the effort required to develop software system. Accurate software cost estimates are critical to both developers and customers. They can be used for generating request for proposals, contract negotiations, scheduling, monitoring and control. Underestimating the costs may result in management approving proposed systems that then exceed their budgets, with underdeveloped functions and poor quality, and failure to complete on time. Overestimating may result in too many resources committed to the project, or, during contract bidding, result in not winning the contract, which can lead to loss of jobs.

8.1.1 Accurate cost estimation is important because:

- It can help to classify and prioritize development projects with respect to an overall business plan.
- It can be used to determine what resources to commit to the project and how well these resources will be used.
- It can be used to assess the impact of changes and support re-planning.
- Projects can be easier to manage and control when resources are better matched to real needs.
- Customers expect actual development costs to be in line with estimated costs.

Software cost estimation involves the determination of one or more of the following estimates:

- Effort (usually in person-months)
- Project duration (in calendar time)
- Cost (in dollars)

8.2: Methods used to estimate software cost

Generally, there are many methods for software cost estimation, which are divided into two groups: Algorithmic and Non-algorithmic. Using of the both groups is required for performing the accurate estimation. If the requirements are known better, their performance will be better. In this section, some popular estimation methods are discussed.

8.2.1 Algorithmic methods

These models work based on the especial algorithm. They usually need data at first and make results by using the mathematical relations. Nowadays, many software estimation methods use these models. Algorithmic Models are classified into some different models. Each algorithmic model uses an equation to do the estimation:

Effort = $f(x_1, x_2, \dots, x_n)$ where, $(x_1 \dots x_n)$ is the vector of the cost factors. The Differences among the existing algorithmic methods are related to choosing the cost factors and function. All cost factors using in these models are:

Product factors: required reliability; product complexity; database size used; required reusability; documentation match to life-cycle needs;

Computer factors: execution time constraint; main storage constraint; computer turnaround constraints; platform volatility;

Personnel factors: analyst capability; application experience; programming capability; platform experience; language and tool experience; personnel continuity;

Project factors: multisite development; use of software tool; required development schedule.

Examples of methods in this category are like line of code, function point analysis and COCOMO

8.2.2 Non algorithmic methods

They are based on analytical comparisons and inferences. For using the Non Algorithmic methods some information about the previous projects which are similar the under estimate project is required and usually estimation process in these methods is done according to the analysis of the previous datasets. Example of methods in this group include machine learning, analogy and expert judgment.

8.3 COCOMO I

Stands for Constructive Cost Model(COCOMO)

This family of models was proposed by Boehm. The models have been widely accepted in practice. In the COCOMOs, the code-size S is given in thousand LOC (KLOC) and Effort is in person-month.

A) *Basic COCOMO*.

This model uses three sets of $\{a, b\}$ depending on the complexity of the software only:

- (1) For simple, well-understood applications, $a = 2.4$, $b = 1.05$;
- (2) For more complex systems, $a = 3.0$, $b = 1.15$;
- (3) For embedded systems, $a = 3.6$, $b = 1.20$.

The basic COCOMO model is simple and easy to use. As many cost factors are not considered, it can only be used as a rough estimate.

Two equations are used to estimate effort and schedule as below:

B) Intermediate COCOMO and Detailed COCOMO.

In the intermediate COCOMO, a nominal effort estimation is obtained using the power function with three sets of $\{a, b\}$, with coefficient a being slightly different from that of the basic COCOMO:

- (1) For simple, well-understood applications, $a = 3.2, b = 1.05$
- (2) For more complex systems, $a = 3.0, b = 1.15$
- (3) For embedded systems, $a = 2.8, b = 1.20$

Then, fifteen cost factors with values ranging from 0.7 to 1.66 are determined. The overall *impact factor* M is obtained as the product of all individual factors, and the estimate is obtained by multiplying M to the nominal estimate.

$$\begin{aligned}
 & \bullet PM_{NS} = A * Size^E * \prod_{i=1}^{17} EM_i \\
 & \text{Where } E = B + 0.01 * \sum_{j=1}^5 SF_j, A = 2.94, B = 0.91 \\
 & \bullet TDEV = C * (PM)^F \\
 & \text{Where } F = D + 0.2 * 0.01 * \sum_{j=1}^5 SF_j = D + 0.2 * (E - B) \\
 & C = 3.67, D = 0.28
 \end{aligned}$$

8.4: Function point

At first, Albrecht (1983) presented Function Point metric to measure the functionality of project. In this method, estimation is done by determination of below indicators:

- User Inputs,
- User Outputs,
- Logic files,
- Inquiries,
- Interfaces

A Complexity Degree which is between 1 and 3 is defined for each indicator. 1, 2 and 3 stand for simple, medium and complex degree respectively. Also, it is necessary to define a weight for each indicator which can be between 3 and 15.

At first, the number of each mentioned indicator should be tallied and then complexity degree and weight are multiplied by each other. Generally, the unadjusted function point count is defined as below:

$$UFC = \sum_{i=1}^5 \sum_{j=1}^3 N_{ij} W_{ij}$$

8.5: Software Line of Code

SLOC is an estimation parameter that illustrates the number of all commands and data definition but it does not include instructions such as comments, blanks, and continuation lines. This parameter is usually used as an analogy based on an approach for the estimation. After computing the SLOC for software, its amount is compared with other projects which their SLOC has been computed before, and the size of project is estimated. SLOC measures the size of project easily. After completing the project, all estimations are compared with the actual ones.

Thousand Lines of Code (KSLOC) are used for estimation in large scale. Using this metric is common in many estimation methods. SLOC Measuring seems very difficult at the early stages of the project because of the lack of information about requirements.

Since SLOC is computed based on language instructions, comparing the size of software which use different languages is too hard. Anyway, SLOC is the base of the estimation models in many complicated software estimation methods. SLOC usually is computed by considering SL as the lowest, SH as the highest and SM as the most probable size.

$$S = \frac{S_L + 4S_M + S_H}{6}$$

Where N_{ij} is the number of indicator i with complexity j and; W_{ij} is the weight of indicator i with complexity j . According to the previous experiences, function point could be useful for software estimations because it could be computed based on requirement specification in the early stages of project. To compute the FP, UFC should be multiplied by a Technical Complexity Factor (TCF) which is obtained from the components.

8.6 Chapter questions

1. Explain the categories software estimation methods are grouped into.
2. Explain the difference between COCOMO1 and COCOMO
3. How is COCOMO1 different from function point analysis and line of code methods used to estimate software cost.
4. Why software cost estimation is an important process in the software engineering process.

LESSON NINE

Quality management and software maintenance

Lesson objectives: at the end of the lesson should be able to:

- Define the term quality management, software maintenance, configuration management and software re-engineering.
- To explain the importance of quality management, software maintenance, configuration management of software.
- Explain processes of ensuring delivering of high quality software through the various software maintenance and configuration management

9.1 Introduction to software maintenance

Software maintenance can be defined as the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage. Pre-delivery activities include planning for post-delivery operations, supportability, and logistics determination. Post-delivery activities include software modification, training, and operating a help desk.

A maintainer is defined by ISO/IEC 12207 as an organization that performs maintenance activities [ISO12207].

ISO/IEC 12207 identifies the primary activities of software maintenance as: process implementation; problem and modification analysis; modification implementation; maintenance review/acceptance; migration; and retirement.

The categories of maintenance defined by ISO/IEC are as follows:

Corrective maintenance: Reactive modification of a software product performed after delivery to correct discovered problems.

Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.

Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability.

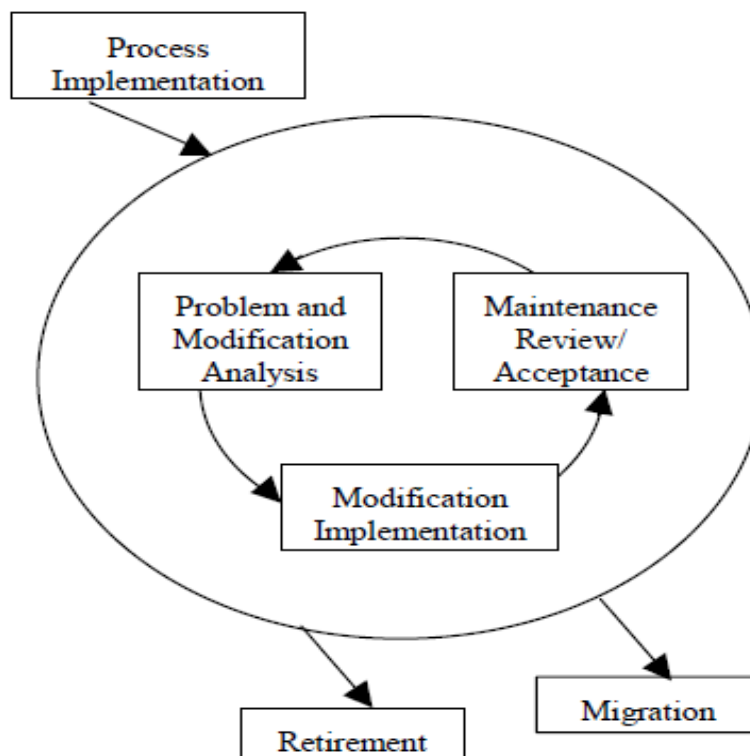
Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before turning into effective faults.

9.1.1. Software maintenance activities

As mentioned in the introduction part to this lesson, the primary activities of any software maintenance are process implementation, problem and modification analysis, modification implementation, maintenance review/acceptance, migration and retirement. They are depicted in the figure on the next page.

- Process Implementation tasks are:
 - Develop maintenance plans and procedures.
 - Establish procedures for Modification Requests.
 - Implement the CM (Configuration management) process.

- Problem and Modification tasks are:
 Perform initial analysis.
 Verify the problem.
 Develop options for implementing the modification.
 Document the results.
 Obtain approval for modification option.
- Modification Implementation tasks are:
 Perform detailed analysis.
 Develop, code, and test the modification.
- Maintenance Review/Acceptance tasks are:
 Conduct reviews.
 Obtain approval for modification.
- Migration tasks are:
 Ensure that migration is in accordance with ISO/IEC 12207.
 Develop a migration plan.
 Notify users of migration plans.
- Conduct parallel operations.
 Notify user that migration has started.
 Conduct a post-operation review.
 Ensure that old data is accessible.
- Software Retirement tasks are:
 Develop a retirement plan.
 Notify users of retirement plans.
 Conduct parallel operations.
 Notify user that retirement has started.
 Ensure that old data is accessible.



9.1.2. Techniques for maintenance

Effective software maintenance is performed using techniques specific to maintenance. The following provides some of the best practice techniques used by maintainers.

- *Program Comprehension:* Programmers spend considerable time in reading and comprehending programs in order to implement changes. Code browsers are a key tool in program comprehension. Clear and concise documentation can aid in program comprehension. Based on the importance of this subtopic, an annual IEEE Computer Society workshop is now held to address program comprehension.
- *Re-engineering:* Re-engineering is defined as the examination and alteration of the subject system to reconstitute it in a new form, and the subsequent implementation of the new form. Reengineering is often not undertaken to improve maintainability but is used to replace aging legacy systems. Refactoring, a program transformation that reorganizes a program without changing its behavior, is now being used in reverse engineering to improve the structure of object oriented programs.
- *Reverse engineering:* Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at higher levels of abstraction. Reverse engineering is passive; it does not change the system, or result in a new one. A simple reverse engineering effort may merely produce call graphs and control flow graphs from source code. One type of reverse engineering is re-documentation. Another type is design recovery. Date Reverse Engineering has gained great importance over the last few years. Reverse engineering topics are discussed at the annual Working Conference on Reverse Engineering (WCRE).
- *Impact Analysis:* Impact analysis identifies all systems and system products affected by a change request and develop an estimate of the resources needed to accomplish the change. It is performed after a change request enters the configuration management process. The objectives of impact analysis are: Determine the scope of a change in order to plan and implement work. Develop accurate estimates of resources needed to perform the work. Analyze the cost/benefits of the requested change. Communicate to others the complexity of a given change.

9.3 Quality management and importance in software engineering.

Software quality can be defined as the degree to which a system or a component meets specified requirements.

Software quality management activities are as follows:

- **Collecting requirements and defining scope** of IT project (focus on verification if defined requirements will be testable)
- **Designing** the solution (focus on planning test process e.g. what type of tests will be performed, how they will be performed in context of test environments and test data.
- **Solution implementation** supported by creating test cases and scenarios, executing them and registering defects including coordination of fixing them.

•**Change management**, supported by verification how planned changes can influence the quality of created solution and eventual change of test plan.

•**Closing project**, supported by realization number of tests focused on complex verification of overall quality of created solution. It can include System Integration Tests, User Acceptance Tests and Operational Acceptance Tests.

Causes of software errors

Software errors can be classified into the following categories:

(1) Faulty definition of requirements: faulty definition of requirements, usually prepared by the client, is one of the main causes of software errors, erroneous definition of requirements, absence of vital requirements, incomplete definition of requirements, inclusion of unnecessary requirements, functions that are not expected to be needed in the near future

(2) Client–developer communication failures: Misunderstandings resulting from defective client–developer communication are additional causes for the errors that prevail in the early stages of the development process

3) Deliberate deviations from software requirements : The developer reuses software modules taken from an earlier project without sufficient analysis of the changes and adaptations needed to correctly fulfill all the new requirements; due to time or budget pressures, the developer decides to omit part of the required functions in an attempt to cope with these pressures

(4) Logical design errors

(5) Coding errors

(6) Non-compliance with documentation and coding instructions

(7) Shortcomings of the testing process

(8) Procedure errors

(9) Documentation errors

9.3.1 Classification of Quality requirements

Classifications of software requirements into software quality factors are as follows

:

•**Product operation factors**: Correctness, Reliability, Efficiency, Integrity, Usability.

•**Product revision factors**: Maintainability, Flexibility, Testability.

•**Product transition factors**: Portability, Reusability, Interoperability.

Product operation software quality factors

•**Correctness** requirements are defined in a list of the software system's required outputs

•**Reliability** requirements deal with failures to provide service. They determine the maximum allowed software system failure rate, and can refer to the entire system or to one or more of its separate functions.

- Efficiency* requirements deal with the hardware resources needed to perform all the functions of the software system in conformance to all other requirements.

- Integrity* requirements deal with the software system security, that is, requirements to prevent access to unauthorized persons, to distinguish between the majority of personnel allowed to see the information and a limited group who will be allowed to add and change data.

- Usability* requirements deal with the scope of staff resources needed to train a new employee and to operate the software system.

Product revision software quality factors

- Maintainability* requirements determine the efforts that will be needed by users and maintenance personnel to identify the reasons for software failures, to correct the failures, and to verify the success of the corrections.

- The capabilities and efforts required to support adaptive maintenance activities are covered by the *flexibility* requirements. These include the resources (i.e. in man-days) required to adapt a software package to a variety of customers of the same trade, of various extents of activities, of different ranges of products and so on.

- Testability* requirements deal with the testing of an information system as well as with its operation.

Product transition software quality factors

- Portability* requirements tend to the adaptation of a software system to other environments consisting of different hardware, different operating systems, and so forth.

- Reusability* requirements deal with the use of software modules originally designed for one project in a new software project currently being developed.

- Interoperability* requirements focus on creating interfaces with other software systems or with other equipment firmware (for example, the firmware of the production machinery and testing equipment interfaces with the production control software). Interoperability requirements can specify the name(s) of the software or firmware for which interface is required. They can also specify the output structure accepted as standard in a specific industry or applications area.

9.3.2 Software Quality Architecture

SQA system always combines a wide range of SQA components, all of which are employed to challenge the multitude of sources of software errors and to achieve an acceptable level of software quality

SQA system components can be classified into six classes:

- Pre-project components:** To assure that (a) the project commitments have been adequately defined considering the resources required, the schedule and budget; and (b) the development and quality plans have been correctly determined.

- Components of project life cycle activities assessment:** The project life cycle is composed of two stages: the development life cycle stage and the operation–maintenance stage.

•**The development life cycle stage components detect design and programming errors.** Its components are divided into the following sub-classes: Reviews, Expert opinions, Software testing.

•**Components of infrastructure error prevention and improvement.** : Eliminate or at least reduce the rate of errors, based on the organization's accumulated SQA experience.

•Components of software quality management : Control of development and maintenance activities and the introduction of early managerial support actions that mainly prevent or minimize schedule and budget failures and their outcomes

•Components of standardization, certification, and SQA system assessment

•Organizing for SQA – the human components

9.4 Configuration management and its importance in software engineering

Configuration management (CM) is the discipline of controlling the evolution of complex systems; software configuration management (SCM) is its specialization for computer programs and associated documents.

SCM differs from general CM in the following two ways. First, software is easier to change than hardware and it therefore changes faster. Even relatively small software systems, developed by a single team, can experience a significant rate of change, and in large systems, such as telecommunications systems, the update activities can totally overwhelm manual configuration management procedures. Second, SCM is potentially more automatable this is because all components of a software system are easily stored on-line. CM for physical systems is hampered by having to handle objects that are not within reach of programmable controls.

Software configuration management is important as it:

- Manages simultaneous update of the same component by different persons in the configuration management process.
- Keeping track of what has been done. Called change tracking by the different persons involved in the process.

9.5 Chapter Question

1. Explain the controls used in the quality management process of a software.
2. Explain the techniques used to maintain software's.
3. What is the importance of configuration management, quality management in the software engineering process?

LESSON TEN

Software prototype and formal specification

Lesson objectives: at the end of the lesson the student should be able to:

- State/ explain the importance of prototyping in software development procedures.
- Explain the procedure that is followed when one is developing a prototype.
- Explain the advantages and disadvantages associated with prototyping.
- Explain scenarios where prototyping is an added advantage in software development procedures

10.1 Introduction to software prototyping.

We define a prototype as a *concrete representation* of part or all of an interactive system. A prototype is a tangible artifact, not an abstract description that requires interpretation. Designers, as well as managers, developers, customers and end users, can use these artifacts to envision and reflect upon the final system.

Hardware and software engineers often create prototypes to study the feasibility of a technical process. They conduct systematic, scientific evaluations with respect to pre-defined benchmarks and, by systematically varying parameters, fine-tune the system. Designers in creative fields, such as typography or graphic design, create prototypes to express ideas and reflect on them. This approach is intuitive, oriented more to discovery and generation of new ideas than to evaluation of existing ideas.

We can analyze prototypes and prototyping techniques along four dimensions:

- *Representation* describes the form of the prototype, e.g., sets of paper sketches or computer simulations;
- *Precision* describes the level of detail at which the prototype is to be evaluated; e.g., informal and rough or highly polished;
- *Interactivity* describes the extent to which the user can actually interact with the prototype; e.g., watch-only or fully interactive
- *Evolution* describes the expected life-cycle of the prototype, e.g. throwaway or iterative.

Prototypes serve different purposes and thus take different forms. A series of quick sketches on paper can be considered a prototype; so can a detailed computer simulation. Both are useful; both help the designer in different ways. We distinguish between two basic forms of representation: off-line and on-line.

Off-line prototypes (also called *paper prototypes*) do not require a computer. They include paper sketches, illustrated story-boards, cardboard mock-ups and videos. The most salient characteristics of off-line prototypes (of interactive systems) is that they are created quickly, usually in the early stages of design, and they are usually thrown away when they have served their purpose.

On-line prototypes (also called *software prototypes*) run on a computer. They include computer animations, interactive video presentations, programs written with scripting languages, and applications developed with interface builders. The cost of producing on-line prototypes is usually higher, and may require skilled programmers to implement advanced interaction and/or visualization techniques.

10.1.1 Software prototyping strategies

Designers must decide what role prototypes should play with respect to the final system and in which order to create different aspects of the prototype. This section presents four strategies: *horizontal*, *vertical*, *task-oriented* and *scenario-based*, which focus on different design concerns.

Horizontal prototypes

The purpose of a horizontal prototype is to develop one entire layer of the design at the same time. This type of prototyping is most common with large software development teams, where designers with different skill sets address different layers of the software architecture. Horizontal prototypes of the user interface are useful to get an overall picture of the system from the user's perspective and address issues such as consistency (similar functions are accessible through similar user commands), coverage (all required functions are supported) and redundancy (the same function is/is not accessible through different user commands).

Vertical prototypes

The purpose of a vertical prototype is to ensure that the designer can implement the full, working system, from the user interface layer down to the underlying system layer.

Vertical prototypes are generally high precision, software prototypes because their goal is to validate an idea at the system level. They are often thrown away because they are generally created early in the project, before the overall architecture has been decided, and they focus on only one design question. For example, a vertical prototype of a spelling checker for a text editor does not require text editing functions to be implemented and tested. However, the final version will need to be integrated into the rest of the system, which may involve considerable architectural or interface changes.

Task based prototypes

Task-based prototypes are organized as a series of tasks, which allows both designers and users to test each task independently, systematically working through the entire system.

Task-oriented prototypes include only the functions necessary to implement the specified set of tasks. They combine the breadth of horizontal prototypes, to cover the functions required by those tasks, with the depth of vertical prototypes, enabling detailed analysis of how the tasks can be supported.

Scenario-based prototypes

Scenario-based prototypes are similar to task-oriented ones, except that they do not stress individual, independent tasks, but rather follow a more realistic scenario of how the system would be used in a real-world setting. Scenarios are stories that describe a sequence of events and how the user reacts. We find it useful to begin with *use scenarios* based on observations of or interviews with real users. Ideally, some of those users should participate in the creation of the

specific scenarios, and other users should critique them based on how realistic they are. Use scenarios are then turned into *design scenarios*, in which the same situations are described but with the functionality of the new system. Design scenarios are used, among other things, to create scenario-based video prototypes or software prototypes. Like task-based prototypes, the developer needs to write only the software necessary to illustrate the components of the design scenario. The goal is to create a situation in which the user can experience what the system would be like in a realistic situation, even if it addresses only a subset of the planned functionality.

10.2 Online rapid prototyping techniques

Off-line prototyping techniques range from simple to very elaborate. Because they do not involve software, they are usually considered a tool for thinking through the design issues, to be thrown away when they are no longer needed.

They include the following methods: simple paper and pencil sketches, three-dimensional mock-ups, wizard-of-oz simulations and video prototypes.

10.3 Offline rapid prototyping techniques

The goal of on-line rapid prototyping is to create higher-precision prototypes than can be achieved with off-line techniques. Such prototypes may prove useful to better communicate ideas to clients, managers, developers and end users. They are also useful for the design team to fine tune the details of a layout or an interaction. They may exhibit problems in the design that were not apparent in less precise prototypes. Finally they may be used early on in the design process for low precision prototypes that would be difficult to create off-line, such as when very dynamic interactions or visualizations are needed. They include the following techniques:

a) Non-interactive simulations

A non-interactive simulation is a computer-generated animation that represents what a person would see of the system if he or she were watching over the user's shoulder. Non-interactive simulations are usually created when off-line prototypes, including video, fail to capture a particular aspect of the interaction and it is important to have a quick prototype to evaluate the idea. It's usually best to start by creating a storyboard to describe the animation, especially if the developer of the prototype is not a member of the design team.

b) Interactive simulations

Designers can also use tools such as Adobe Photoshop to create Wizard-of-Oz simulations. For example, the effect of dragging an icon with the mouse can be obtained by placing the icon of a file in one layer and the icon of the cursor in another layer, and by moving either or both layers. The visibility of layers, as well as other attributes, can also create more complex effects. Like Wizard-of-Oz and other paper prototyping techniques, the behavior of the interface is generated by the user who is operating the Photoshop interface.

c) Scripting languages

Scripting languages are the most advanced rapid prototyping tools. As with the interactive-simulation tools described above, the distinction between rapid prototyping tools and development tools is not always clear. Scripting languages make it easy to quickly develop throw-away quickly (a few hours to a few days), which may or may not be used in the final system, for performance or other technical reasons.

A scripting language is a programming language that is both lightweight and easy to learn. Most scripting languages are interpreted or semi-compiled, i.e. the user does not need to go through a compile-link-run cycle each time the script (program) is changed. Scripting languages can be forbidding: they are not strongly typed and non-fatal errors are ignored unless explicitly trapped by the programmer. Scripting languages are often used to write small applications for specific purposes and can serve as glue between pre-existing applications or software components.

10.4 Chapter Questions

1. What is the importance of prototyping software?
2. Explain two categories/techniques of prototyping that can be used in the software engineering process.
3. Explain the prototyping process.