UNIT I

1. What is UML? What are the ways and perspectives to apply UML?

The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems.

Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of software engineering. The standard is managed, and was created by, the Object Management Group. UML includes a set of graphic notation techniques to create visual models of software-intensive systems.

The Unified Modeling Language is commonly used to visualize and construct systems which are software intensive. Because software has become much more complex in recent years, developers are finding it more challenging to build complex applications within short time periods. Even when they do, these software applications are often filled with bugs, and it can take programmers weeks to find and fix them. This is time that has been wasted, since an approach could have been used which would have reduced the number of bugs before the application was completed.

Three ways to apply UML:

1. UML as sketch:
   - Informal and incomplete diagrams
   - Created to explore difficult parts of the problem

Prepared By: Mrs.R.BARONA, AP/IT
2. UML as blueprint:
   - Detailed design diagram
   - Used for better understanding of code

3. UML as programming language:
   - Complete executable specification of a software system in UML

Three perspectives to apply UML:

1. Conceptual perspective:
   - Diagrams describe the things of real world.
     UML diagrams are used to describe things in situations of real world.
     Raw UML object diagram notation used to visualize.

2. Specification perspective:
   - Diagrams describe software abstractions or components with specifications and interfaces.
     It describes the real world things, software abstraction and component with specification and interfaces. Raw UML class diagram notation used to visualize software components.

3. Implementation perspective:
   - Diagrams describe software implementation in a particular technology

2. What is UP? Explain are the phases of UP.

Prepared By: Mrs.R.BARONA, AP/IT
The Unified Process has emerged as a popular iterative software development process for building object oriented systems.

Unified process is a iterative process, risk driven process and architecture centric approach to software development. It comes under software development process.

The **Unified Software Development Process** or **Unified Process** is a popular iterative and incremental software development process framework. The best-known and extensively documented refinement of the Unified Process is the Rational Unified Process (RUP).

**UP Phases:**

I. **Inception:**

Inception is the initial stage of project. It deals with approximate vision, business case, scope of project and vague estimation.

- Initial stage of project
- Approximate vision
- Business case and scope
- Vague estimate

Inception is the smallest phase in the project, and ideally it should be quite short. If the Inception Phase is long then it may be an indication of excessive up-front specification, which is contrary to the spirit of the Unified Process.

**The following are typical goals for the Inception phase.**

Prepared By: Mrs.R.BARONA, AP/IT
Establish a justification or business case for the project
Establish the project scope and boundary conditions
Outline the use cases and key requirements that will drive the design tradeoffs
Outline one or more candidate architectures
Identify risks
Prepare a preliminary project schedule and cost estimate

The Lifecycle Objective Milestone marks the end of the Inception phase.

Advantages of inception.

- Estimation or plans are expected to be reliable.
- After inception, design architecture can be made easily because all the use cases are written in detail.

II. Elaboration:

During the Elaboration phase the project team is expected to capture a healthy majority of the system requirements. However, the primary goals of Elaboration are to address known risk factors and to establish and validate the system architecture. Common processes undertaken in this phase include the creation of use case diagrams, conceptual diagrams (class diagrams with only basic notation) and package diagrams (architectural diagrams).

- Refined vision
- Core architecture
- Resolution of high risk
- Identification of most requirement and scope

Prepared By: Mrs.R.BARONA, AP/IT
III. **Construction:**

Construction is the largest phase in the project. In this phase the remainder of the system is built on the foundation laid in Elaboration. System features are implemented in a series of short, timeboxed iterations. Each iteration results in an executable release of the software. It is customary to write full text use cases during the construction phase and each one becomes the start of a new iteration.

Common UML (Unified Modelling Language) diagrams used during this phase include Activity, Sequence, Collaboration, State (Transition) and Interaction Overview diagrams. The Initial Operational Capability Milestone marks the end of the Construction phase.

- Design the elements
- Preparation for deployment

IV. **Transition:**

The final project phase is Transition. In this phase the system is deployed to the target users. Feedback received from an initial release (or initial releases) may result in further refinements to be incorporated over the course of several Transition phase iterations. The Transition phase also includes system conversions and user training. The Product Release Milestone marks the end of the Transition phase.

- Beta tests
- Deployments

3. **Explain Case Study For NextPOS System**
The case study is the NextGen point-of-sale (POS) system. In this apparently straightforward problem domain, we shall see that there are very interesting requirement and design problems to solve. In addition, it is a realistic problem; organizations really do write POS systems using object technologies.

A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control.

These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).

A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.

Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.

Using an iterative development strategy, we are going to proceed through requirements, object-oriented analysis, design, and implementation.

Prepared By: Mrs.R.BARONA, AP/IT
Architectural Layers and Case Study Emphasis

A typical object-oriented information system is designed in terms of several architectural layers or subsystems.

The following is not a complete list, but provides an example:

- **User Interface**: Graphical interface; windows.
- **Application Logic and Domain Objects**: Software objects representing domain concepts (for example, a software class named *Sale*) that fulfill application requirements.
- **Technical Services**: General purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging. These services are usually application-independent and reusable across several systems.

OOA/D is generally most relevant for modeling the application logic and technical service layers.

The NextGen case study primarily emphasizes the problem domain objects, allocating responsibilities to them to fulfill the requirements of the application.

Object-oriented design is also applied to create a technical service subsystem for interfacing with a database.

In this design approach, the UI layer has very little responsibility; it is said to be thin. Windows do not contain code that performs application logic or processing. Rather, task requests are forwarded on to other layers.

4. **Explain about Inception Phase**

Prepared By: Mrs.R.BARONA, AP/IT
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.

Prepared By: Mrs.R.BARONA, AP/IT
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 changes 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. The overriding goal of the inception phase is to achieve concurrence among all stakeholders on the lifecycle objectives for the project.

The inception phase is of significance primarily for new development efforts, in which there are significant business and requirements risks which must be addressed before the project can proceed.

For projects focused on enhancements to an existing system, the inception phase is more brief, but is still focused on ensuring that the project is both worth doing and possible to do.
Objectives

The primary objectives of the Inception phase include:

- Establishing the project’s software scope and boundary conditions, including an operational vision, acceptance criteria and what is intended to be in the product and what is not.

- Discriminating the critical use cases of the system, the primary scenarios of operation that will drive the major design tradeoffs.

- Exhibiting, and maybe demonstrating, at least one candidate architecture against some of the primary scenarios.

- Estimating the overall cost and schedule for the entire project (and more detailed estimates for the elaboration phase that will immediately follow).

- Estimating potential risks (the sources of unpredictability).

- Preparing the supporting environment for the project.

Essential Activities

The essential activities of the Inception include:

Formulating the scope of the project. This involves capturing the context and the most important requirements and constraints to such an extent that you can derive acceptance criteria for the end product.
Planning and preparing a business case. Evaluating alternatives for risk management, staffing, project plan, and cost/schedule/profitability tradeoffs.

Synthesizing a candidate architecture, evaluating tradeoffs in design, and in make/buy/reuse, so that cost, schedule and resources can be estimated. The aim here is to demonstrate feasibility through some kind of proof of concept. This may take the form of a model which simulates what is required, or an initial prototype which explores what are considered to be the areas of high risk.

Preparing the environment for the project, assessing the project and the organization, selecting tools, deciding which parts of the process to improve.
5. **Explain about Usecase Modeling**

The Use Case Model describes the proposed functionality of the new system. A Use Case represents a discrete unit of interaction between a user (human or machine) and the system. A Use Case is a single unit of meaningful work; for example, login to system, register with system, and create order are all Use Cases. Each Use Case has a description which describes the functionality that will be built in the proposed system. A Use Case may 'include' another Use Case's functionality or 'extend' another Use Case with its own behavior.

Use Cases are typically related to 'actors'. An actor is a human or machine entity that interacts with the system to perform meaningful work.

![Use Case Diagram]

**Actors**
Prepared By: Mrs.R.BARONA, AP/IT
An Actor is a user of the system. This includes both human users and other computer systems. An Actor uses a Use Case to perform some piece of work which is of value to the business.

The set of Use Cases an actor has access to defines their overall role in the system and the scope of their action.

Constraints, Requirements and Scenarios

The formal specification of a Use Case includes:

1. **Requirements:**

   These are the formal functional requirements that a Use Case must provide to the end user. They correspond to the functional specifications found in structured methodologies. A requirement is a contract that the Use Case will perform some action or provide some value to the system.

2. **Constraints:**

   These are the formal rules and limitations that a Use Case operates under, and includes pre- post- and invariant conditions. A pre-condition specifies what must have already occurred or be in place before the Use Case may start. A post-condition documents what will be true once the Use Case is complete. An invariant specifies what will be true throughout the time the Use Case operates.

3. **Scenarios:**

Prepared By: Mrs.R.BARONA, AP/IT
Scenarios are formal descriptions of the flow of events that occurs during a Use Case instance. These are usually described in text and correspond to a textual representation of the Sequence Diagram.
6. Describe use case relationships.

Use case relationships is divided into three types

1. Include relationship
2. Extend relationship
3. Generalization

1. Include relationship:
   - It is common to have some practical behavior that is common across several use cases.
   - It is simply to underline it or highlight it in some fashion

   Example:
   Paying by credit: Include Handle Credit Payment

2. Extend relationship:
   - Extending the use case or adding new use case to the process
   - Extending use case is triggered by some conditions called extension point.

3. Generalization:
   - Complicated work and unproductive time is spending in this use case relationship.
   - Use case experts are successfully doing use case work without this relationship.
Includes and Extends relationships between Use Cases

One Use Case may include the functionality of another as part of its normal processing. Generally, it is assumed that the included Use Case will be called every time the basic path is run. An example may be to list a set of customer orders to choose from before modifying a selected order in this case the <list orders> Use Case may be included every time the <modify order> Use Case is run.

A Use Case may be included by one or more Use Cases, so it helps to reduce duplication of functionality by factoring out common behavior into Use Cases that are re-used many times.

One Use Case may extend the behavior of another - typically when exceptional circumstances are encountered.

Relationships Between Use Cases
Use cases could be organized using following relationships:

- Generalization
- Association
- Extend
- Include

Generalization Between Use Cases
Generalization between use cases is similar to generalization between classes; child use case inherits properties and behavior of the parent use case and may override the behavior of the parent.
Notation: Generalization is rendered as a solid directed line with a large open arrowhead (same as generalization between classes).

**Association Between Use Cases**

Use cases can only be involved in **binary** Associations. Two use cases specifying the same subject cannot be associated since each of them individually describes a complete usage of the system.

**Extend Relationship**

*Extend* is a directed relationship from an extending use case to an extended use case that specifies how and when the behavior defined in usually supplementary (optional) extending use case can be inserted into the behavior defined in the use case to be extended.

**Note:** *Extended use case* is meaningful on its own, independently of the extending use case, while the *extending use case* typically defines behavior that is **not necessarily meaningful by itself**.
The extension takes place at one or more extension points defined in the **extended use case**.

The extend relationship is **owned** by the extending use case. The same extending use case can extend more than one use case, and extending use case may itself be extended.

**Extend** relationship between use cases is shown by a dashed arrow with an open arrowhead from the **extending use case** to the **extended (base) use case**. The arrow is labeled with the keyword **Registration** use case is meaningful on its own, and it could be extended with optional **Get Help On Registration** use case.

The **condition** of the extend relationship as well as the references to the extension points are optionally shown in a **Note** attached to the corresponding extend relationship.
Registration use case is conditionally extended by Get Help On Registration use case in extension point Registration Help

**Include Relationship**

An **include** relationship is a **directed relationship** between two use cases, implying that the behavior of the required (not optional) **included** use case is inserted into the behavior of the **including** (base) use case. Including use case **depends on** the addition of the included use case.

The include relationship is intended to be used when there are **common parts** of the behavior of two or more use cases. This common part is extracted into a separate use case to be included by all the base use cases having this part in common.

As the primary use of the include relationship is to **reuse common parts**, including use cases are usually **not complete** by themselves but dependent on the included use cases.

**Include** relationship between use cases is shown by a dashed arrow with an open arrowhead from the including (base) use case to the included (common part) use case. The arrow is labeled with the keyword **include**.
7. **What are the templates used to write a Fully-dressed Use case?**

<table>
<thead>
<tr>
<th>Use case section</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case name</td>
<td>Starts with a verb.</td>
</tr>
<tr>
<td>Scope</td>
<td>The system under design.</td>
</tr>
<tr>
<td>Level</td>
<td>Defines user goal.</td>
</tr>
<tr>
<td>Primary actor</td>
<td>Who calls on the system to deliver its services.</td>
</tr>
<tr>
<td>Stake holders and interests</td>
<td>Who cares about this use case and what do they do.</td>
</tr>
<tr>
<td>Precondition</td>
<td>What must be true on start and worth telling the reader.</td>
</tr>
<tr>
<td>Success guarantee</td>
<td>What must be true on successful condition.</td>
</tr>
<tr>
<td>Main success scenario</td>
<td>A typical, unconditional happy path scenario of success.</td>
</tr>
<tr>
<td>Extension</td>
<td>Alternate scenarios of success or failure.</td>
</tr>
<tr>
<td>Special requirements</td>
<td>Related non functional requirements.</td>
</tr>
<tr>
<td>Technology and data variation list</td>
<td>Varying I/O methods and data formats.</td>
</tr>
<tr>
<td>Frequency occurrence</td>
<td>Could be nearly continuous.</td>
</tr>
</tbody>
</table>
8. Explain in detail about Domain model refinement.

**Generalization:**

Generalization is the activity of identifying commonality among concepts and defining superclass and subclass relationships in detail about Domain model refinement.

**100% Rule:**

100% of the conceptual superclass definition should be applicable to the subclass. The subclass must conform to 100% of the superclass is attributes and associations.

**Guidelines to define conceptual subclasses:**

- The subclasses have additional attributes of interest.
- The subclass has additional associations of interest.

**Guidelines to define conceptual superclass:**

- The potential conceptual subclasses represent variations of a similar concept.
- The subclasses will conform to the 100% and Is-a rule.
UNIT II

1. **Explain in detail about Elaboration Phase.**

   During the Elaboration phase the project team is expected to capture a healthy majority of the system requirements.

   However, the primary goals of Elaboration are to address known risk factors and to establish and validate the system architecture.

   Common processes undertaken in this phase include the creation of use case diagrams, conceptual diagrams (class diagrams with only basic notation) and package diagrams (architectural diagrams).

   The architecture is validated primarily through the implementation of an Executable Architecture Baseline.

   This is a partial implementation of the system which includes the core, most architecturally significant, components.

   It is built in a series of small, time boxed iterations. By the end of the Elaboration phase the system architecture must have stabilized and the executable architecture baseline must demonstrate that the architecture will support the key system functionality and exhibit the right behavior in terms of performance, scalability and cost.

   The final Elaboration phase deliverable is a plan (including cost and schedule estimates) for the Construction phase.
At this point the plan should be accurate and credible, since it should be based on the Elaboration phase experience and since significant risk factors should have been addressed during the Elaboration phase.

The Lifecycle Architecture Milestone marks the end of the Elaboration phase.

During the elaboration phase the majority of the Use Cases are specified in detail and the system architecture is designed.

This phase focuses on the "Do-Ability" of the project. We identify significant risks and prepare a schedule, staff and cost profile for the entire project.
2. Explain in detail about Domain model.

Domain model:

Domain model means a representation of real-situation or conceptual classes, not of software objects.

A domain model, or Domain Object Model (DOM) in problem solving and software engineering can be thought of as a conceptual model of a domain of interest (often referred to as a problem domain) which describes the various entities, their attributes and relationships, plus the constraints that govern the integrity of the model elements comprising that problem domain.

Symbols used for Domain model:

1. Domain object/ conceptual classes
2. Association between conceptual classes
3. Attributes and methods

Is a domain model a Picture of Software Business Objects:

A UP Domain Model is a visualization of things in a real-situation domain of interest, not of software objects such as Java or C# classes, or software objects with responsibilities.

Therefore, the following elements are not suitable in a domain model:

- Software artifacts
- Responsibilities or methods.

Prepared By: Mrs.R.BARONA, AP/IT
Conceptual classes: It is considered in terms of symbols, intension, extension.

Are Domain and Data Models the Same Thing:
- Data Model: It shows the persistent data to be stored somewhere else.
- It has relation database design. It has some attributes and methods.
- Domain Model: Domain model is not a data model because it does not have attributes and methods for a class.

Overview
The domain model is created in order to represent the vocabulary and key concepts of the problem domain. The domain model also identifies the relationships among all the entities within the scope of the problem domain, and commonly identifies their attributes.

A domain model that encapsulates methods within the entities is more properly associated with object oriented models. The domain model provides a structural view of the domain that can be complemented by other dynamic views, such as Use Case models.

An important benefit of a domain model is that it describes and constrains the scope of the problem domain. The domain model can be effectively used to verify and validate the understanding of the problem domain among various stakeholders. It is especially helpful as a communication tool and a focusing point both amongst the different members of the business team as well as between the technical and business teams.

Prepared By: Mrs.R.BARONA, AP/IT
Usage

A well-thought domain model serves as a clear depiction of the conceptual fabric of the problem domain and therefore is invaluable to ensure all stakeholders are aligned in the scope and meaning of the concepts indigenous to the problem domain.

A high fidelity domain model can also serve as an essential input to solution implementation within a software development cycle since the model elements comprising the problem domain can serve as key inputs to code construction, whether that construction is achieved manually or through automated code generation approaches.

It is important, however, not to compromise the richness and clarity of the business meaning depicted in the domain model by expressing it directly in a form influenced by design or implementation concerns.

The domain model is one of the central artifacts in the project development approach called Feature Driven Development (FDD).

In UML, a class diagram is used to represent the domain model. In Domain-driven design, the domain model (Entities and Value objects) is a part of the Domain layer which often also includes other concepts such as Services.
Sample domain model for a health insurance plan
3. **Explain in detail about Conceptual Data Modeling**

**Introduction**

Conceptual data modeling represents the initial stage in the development of the design of the persistent data and persistent data storage for the system. In many cases, the persistent data for the system are managed by a relational database management system (RDBMS).

The business and system entities identified at a conceptual level from the business models and system requirements will be evolved through the use-case analysis, use-case design, and database design activities into detailed physical table designs that will be implemented in the RDBMS. Note that the Conceptual Data Model discussed in this concept document is not a separate artifact.

Instead it consists of a composite view of information contained in existing Business Modeling, Requirements, and Analysis and Design Disciplines artifacts that is relevant to the development of the Data Model.

The Data Model typically evolves through the following three general stages:

**Conceptual:**

This stage involves the identification of the high level key business and system entities and their relationships that define the scope of the problem to be addressed by the system.
These key business and system entities are defined using the modeling elements of the UML profile for business modeling included in the Business Analysis Model and the Analysis Class model elements of the Analysis Model.

**Logical:**

This stage involves the refinement of the conceptual high level business and system entities into more detailed logical entities. These logical entities and their relationships can be optionally defined in a Logical Data Model using the modeling elements of the UML profile for database design as described in Guidelines: Data Model.

This optional Logical Data Model is part of the Artifact: Data Model and not a separate RUP artifact.

**Physical:**

This stage involves the transformation of the logical class designs into detailed and optimized physical database table designs. The physical stage also includes the mapping of the database table designs to table spaces and to the database component in the database storage design.

The activities related to database design span the entire software development lifecycle, and the initial database design activities might start during the inception phase. For projects that use business modeling to describe the business context of the application, database design may start at a conceptual level with the identification of Business Actors and Business Use Cases in the Business Use-Case Model, and the Business Workers and Business Entities in the Business Analysis Model.

Prepared By: Mrs.R.BARONA, AP/IT
For projects that do not use business modeling, the database design might start at the conceptual level with the identification of System Actors and System Use Cases in the Use-Case Model, and the identification of Analysis Classes in the Analysis Model from the Use-Case Realizations.

The figure below shows the set of Conceptual Data Model elements that reside in the Business Models, Requirements Models, and the Analysis Model.

The following sections describe the elements of the Business Models, Use-Case Model, and Analysis Model that can be used to define the initial Conceptual Data Model for persistent data in the system.

Prepared By: Mrs.R.BARONA, AP/IT
4. **Explain about Conceptual Data Modeling Elements**

**Business Models**

**Business Use-Case Model**

The Business Use-Case Model consists of Business Actors and Business Use Cases. The Business Use Cases represent key business processes that are used to define the context for the system to be developed. Business Actors represent key external entities that interact with the business through the Business Use Cases. The figure below shows a very simple example Business Use-Case Model for an online auction application.

![Business Use-Case Model Diagram](image)

As entities of significance to the problem of space for the system, Business Actors are candidate entities for the Conceptual Data Model. In the example above, the Buyer and Seller Business Actors are candidate entities for which the online auction application must store information.

**Business Analysis Model**

The Business Analysis Model contains classes that model the Business Workers and Business Entities identified from analysis of the workflow in the Business Use Case. Business Workers represent the participating workers that perform the actions needed to carry out that workflow.
Business Entities are "things" that the Business Workers use or produce during that workflow. In many cases, the Business Entities represent types of information that the system must store persistently.

The figure below shows an example sequence diagram that depicts Business Workers and Business Entities from one scenario of the Business Use Case titled "Provide Online Auction" for managing an auction.

In this simplified example, the Auction Manager object represents a Business Worker role that will likely be performed by the online auction management system itself. The Auction and Auction Item objects are Business Entities that are used or produced by the Auction Manager worker acting as an agent for the Seller and Buyer.
Business Actors. From a database design perspective, the Auction and Auction Item Business Entities are candidate entities for the Conceptual Data Model.

**Requirements and Analysis Models**

For projects that do not perform business modeling, the Requirements (System Use Case) and Analysis Models contain model elements that can be used to develop an initial Conceptual Data Model. For projects that use business modeling, the business entities and relationships identified in the Business Analysis Models are refined and detailed in the Analysis Model as Entity Classes.

**System Use-Case Model**

The System Use-Case Model contains System Actors and System Use Cases that define the primary interactions of the users with the system. The System Use Cases define the functional requirements for the system.

From a conceptual data modeling perspective, the System Actors represent entities external to the system for which the system might need to store persistent information. This is important in cases where the System Actor is an external system that provides data to and/or receives data from the system under development. System Actors can be derived from the Business Actors in the Business Use-Case Model and the Business Workers in the Business Analysis Model.

The figure below depicts the Business Use-Case Model for the online auction system. In this model, the Buyer and Seller Business Actors are now derived from a generic User Business Actor.
A new System Actor named Credit Service Bureau has been added to reflect the need to process payments through an external entity. This new System Actor is another candidate entity for the Conceptual Data Model.

**Analysis Model**

The Analysis Model contains the Analysis Classes identified in the Use-Case Realizations for the System Use Cases. The types of Analysis Classes that are of primary interest from a conceptual data modeling perspective are the Entity Analysis Classes.

As defined in Guidelines: Analysis Class, Entity Analysis Classes represent information managed by the system that must be stored in a persistent manner. The Entity Analysis Classes and their relationships form the basis of the initial Data Model for the application.

The conceptual Entity Analysis Classes in the Analysis Model might be refined and detailed into logical Persistent Design Classes in the Design Model. These design classes represent candidate tables in the Data Model. The attributes of the classes are candidate columns for the tables and also represent candidate keys for them. See Guidelines: Forward-Engineering Relational Databases for a description of how elements in the Design Model can be mapped to Data Model elements.
5. **Explain in detail about Association.**

**Association:**

Associations are defined as “the semantic relationship between two or more classifiers that involve connections among their instances.”

**How to name an association in UML:**

Association names should start with a capital letter, since an association represents a classifier of links between instances; in the UML, classifiers should start with a capital letter.

Association defines the relationship between two or more classes in the System. These generally relates to the one object having instance or reference of another object inside it. This article discusses on how we can implement Association in UML.

**Associations in UML can be implemented using following ways:**

1) Multiplicity
2) Aggregation
3) Composition

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Only One Instance</td>
</tr>
<tr>
<td>0..1</td>
<td>Zero or One Instance</td>
</tr>
<tr>
<td>*</td>
<td>Many Instance</td>
</tr>
<tr>
<td>0..*</td>
<td>Zero or Many Instance</td>
</tr>
<tr>
<td>1..*</td>
<td>One or Many Instance</td>
</tr>
</tbody>
</table>
Role of association: Each end of an association is called a role.

Roles may optionally have:
- Multiplicity expression
- Name
- Navigability

Multiplicity:
Multiplicity defines how many instances of a class A can be associated with one instance of a class B.
6. **Explain about Aggregation and Composition**

**Aggregation**

Aggregation is a specialize form of Association where all object have their own lifecycle but there is a ownership like parent and child. Child object can not belong to another parent object at the same time. We can think of it as "has-a" relationship.

**Implementation details:**

1. Typically we use pointer variables that point to an object that lives outside the scope of the aggregate class
2. Can use reference values that point to an object that lives outside the scope of the aggregate class
3. Not responsible for creating/destroying subclasses

Let's take an example of Employee and Company.

A single Employee can not belong to multiple Companies (legally!!), but if we delete the Company, Employee object will not destroy. Here is respective Model and Code for the above example.

**Composition**

Composition is again specialize form of Aggregation. It is a strong type of Aggregation. Here the Parent and Child objects have coincident lifetimes.

Prepared By: Mrs.R.BARONA, AP/IT
Child object does not have its own lifecycle and if parent object gets deleted, then all of its child objects will also be deleted.

1. Open a new Document name it as test.txt
2. Write this sentence inside this document "This is a composition".
3. Save the document.
4. Now, delete this document.

This is what is called composition, you can't move the sentence "This is a composition" from the document because its lifecycle is linked to the parent (i.e. the document here !!)

**Implementation details:**

1. Typically we use normal member variables
2. Can use pointer values if the composition class automatically handles allocation/deallocation
3. Responsible for creation/destruction of subclasses Let's take an example of a relationship between House and its Rooms.

House can contain multiple rooms there is no independent life for room and any room can not belong to two different house. If we delete the house room will also be automatically deleted. Here is respective Model and Code for the above example.

**Room class has Composition Relationship with House class**

```c++
myHse_p : House
name_p : char *

<<constructor>> Room()
<<destructor>> ~Room()
<<static>> initList_v()
<<static>> createRoom_v()
disp()
```

```c++
roomsList_p : Room
name_p : char *

<<constructor>> House()
<<destructor>> ~House()
```

Prepared By: Mrs. R. BARONA, AP/IT
This is what is called composition, you can't move the sentence "This is a composition" from the document because its lifecycle is linked to the parent (i.e. the document here !!)

**Aggregation:**

1. Create a file called file.txt

2. Make a simple Application to open the File.txt (rw mode), but don't program it close the connection.

3. Run an instance of this application

4. Keep the first instance, and run another instance of this application (In theory it should complain that it can't open the file in rw mode because it is already used by other application).

5. Close the 2 instances (make sure you close the connection).

From the above application, we understand that the Application and the File has a separate lifecycles, however this file can be opened only by one application simuletanously (there is only one parent at the same time, however, this parent can move the child to another parent or can make it orphan).
7. Explain in detail about Attributes.

Attributes:

An attribute is a logical data value of an object. It is useful to identify those conceptual classes that are needed to satisfy the information requirements of the current scenarios under development.

When to show attributes?

Include attributes that the requirements suggest or imply a need to remember information.

The full syntax for an attribute in the UML is:

visibility name : type multiplicity = default { property-string}.

Derived attribute:

The total attribute in the Sale can be calculated or derived from the information in the SalesLineItem. It is derivable attribute, we use convention: a / symbol before the attribute name.

Datatype attribute:

It is a primitive datatype such as numbers, character, Boolean, string and enumerations.

Attribute

A logical data value of an object
In UML:
- Attributes are shown in the second compartment of the class box.
- The type of an attribute may optionally be shown.

In a domain model, attributes and data types should be simple. Complex concepts should be represented by an association to another conceptual class.

An attribute should be what the UML standard calls a data type: a set of values for which unique identity is not meaningful. Numbers, strings, Booleans, dates, times,
phone numbers, and addresses are examples of data types. Values of these types are called **value objects**.

**Relating Types**

Conceptual classes in a domain model should be related by associations, not attributes.

In particular, an attribute should not be used as a kind of **foreign key**.

**Quantities and Units**

Quantities with associated units should be represented either as conceptual classes or as attributes of specialized types that imply units (e.g., *Money* or *Weight*).
Derived Attributes

A quantity that can be calculated from other values, such as role multiplicities, is a derived attribute, designated in UML by a leading slash symbol.
8. **Explain in detail about UML activity diagram.**

This diagram shows sequential and parallel activities in a process.

**UML activity notations:**

The UML activity notations are,

- Rake symbol
- Merge symbol
- Decision symbol

**Guidelines for writing activity diagram:**

- This is used for very complex processor.
- If your business process, then use the rake notation for sub activity system.
- On the first overview “level0” diagram, keep all the actions at a very high level of abstraction, so that the diagram is short and sweet.

**Overview:**

Activity diagram is another important diagram in UML to describe dynamic aspects of the system.

Activity diagram is basically a flow chart to represent the flow from one activity to another activity. The activity can be described as an operation of the system. So the control flow is drawn from one operation to another.

This flow can be sequential, branched or concurrent. Activity diagrams deals with all type of flow control by using different elements like fork, join etc.
**Purpose:**

The basic purposes of activity diagrams are similar to other four diagrams. It captures the dynamic behaviour of the system.

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing dynamic nature of a system but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in activity diagram is the message part.

It does not show any message flow from one activity to another. Activity diagram is some time considered as the flow chart. Although the diagrams looks like a flow chart but it is not. It shows different flow like parallel, branched, concurrent and single. So the purposes can be described as:

- Draw the activity flow of a system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.

**How to draw Component Diagram?**

Activity diagrams are mainly used as a flow chart consists of activities performed by the system. But activity diagram are not exactly a flow chart as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane etc.

Before drawing an activity diagram we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities we need to understand how they are associated with constraints and conditions.

Prepared By: Mrs.R.BARONA, AP/IT
So before drawing an activity diagram we should identify the following elements:

- Activities
- Association
- Conditions
- Constraints

Once the above mentioned parameters are identified we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

The following is an example of an activity diagram for order management system. In the diagram four activities are identified which are associated with conditions.

One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and mainly used by the business users.

The following diagram is drawn with the four main activities:

- Send order by the customer
- Receipt of the order
- Confirm order
- Dispatch order

After receiving the order request condition checks are performed to check if it is normal or special order. After the type of order is identified dispatch activity is performed and that is marked as the termination of the process.

Prepared By: Mrs.R.BARONA, AP/IT
UNIT-3

1. Explain about Logical Architecture And Uml Package Diagram

Logical architecture:

It’s called the logical architecture because there’s no decision about how these elements are deployed across different operating system processes or across physical computers in a network.

Layer:

A layer is a very coarse grained grouping of classes, packages or subsystems that has a cohesive responsibility for a major aspect of the system.

Typically layers in an OO system include:

- User Interface
- Application logic and domain objects
- Technical Services

Layered architecture is divided into

- Strict Layered architecture
- Relaxed Layered architecture

Uml package diagram:

It is used for designing logical architecture of the system. Using this package we can group anything. Eg: classes, other packages (usecases) Nesting of packages is common in UML package diagram.

Prepared By: Mrs. R.BARONA, AP/IT
Uml tool reverse engineering package from code:

First sketch a UML package diagram then organize code according to this package sketches. This process is done in earlier stage. The code base grows and we spend more time in programming and less time in modeling.

Guidelines to design a layer:

- Organize the large scale logical structure of a system into discrete layers of distinct related responsibilities, with clean and cohesive separation of higher layer and lower layer. Higher layer is called as more application specific service. Lower layer is called as general service.
- Collaboration and coupling is from higher to lower layer. Lower to higher layer coupling is avoided.

Benefits of using layers:

- There is a separation of concern that means separation of higher level from lower level services. Lower level services are called as general services. Higher level services is called as more application specific services. This reduces coupling and dependency and improves cohesion.
- By using layers complexity is also reduced.
- Some layers can be replaced with new implementation. This is generally not possible for lower level such as technical services, foundation etc. It can be only possible for UI, application and domain layer. This layer architecture can be developed or maintained by team because of logical segmentation.
Separation of concern:

It is the responsibility of objects in the layer should be strongly related to each other and should not be mixed with the responsibilities of other layer. UI object should not be an application logic.

Reverse engineering package diagram from code:

The great use if UML case tool is to reverse engineers. The source code and generate a package diagram automatically.
2. Explain about UML Interaction Diagram

UML includes interaction diagram to show how two objects interact through Messages.

Two types of interaction diagram are

- Sequence diagram
  - It is the important and commonly used diagram.
- Communication interaction diagram

Sequence and communication interaction diagram express similar interaction. It provides a big picture overview of how asset of interaction diagram are related in terms of logical flow and process flow. The most important diagram among the now is sequence diagram.

Sequence diagram:

- It is in the fence format. New objects are added in the right side. Messages are passed in the time ordering.

Communication interaction diagram:

- Messages are passed in number ordering.

Strength and weakness of sequence vs communication diagram:

<table>
<thead>
<tr>
<th>TYPES</th>
<th>STRENGTH</th>
<th>WEAKNESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Clearly shows sequence or time ordering messages.</td>
<td>Forced to extend to the right when adding new objects. Consumes horizontal spaces.</td>
</tr>
<tr>
<td></td>
<td>It has large set of deatailed</td>
<td></td>
</tr>
</tbody>
</table>
### Mark Questions with Answers

<table>
<thead>
<tr>
<th>notations.</th>
<th>Communication</th>
<th>More difficult to see sequence of messages. Only fewer notations are present.</th>
</tr>
</thead>
</table>

**Singleton object:**

A class with only one instance is called singleton object and instance is called as singleton instance.

**Basic sequence diagram notations:**

**Lifeline boxes and lifeline:**

The lifeline boxes including a vertical line extended below. These are the actual lifeline of the objects. The lifeline is indicated as dashed line. The lifeline boxes are used for indicating boxes and objects.

**Messages:**

Each message between the objects is represented with a message expression on a filled arrowed solid line between the vertical lifelines.

**Diagram frames in uml sequence diagram:**

Frames are also called as interaction frame or diagram frames. It is a region or fragmentation of the diagram. Frame has label, operator and guard.

**Types of frames:**

<table>
<thead>
<tr>
<th>Frame operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt</td>
<td>It is alternative fragment for mutual exclusion conditional logic expressed in guards.</td>
</tr>
</tbody>
</table>

Prepared By: Mrs. R. BARONA, AP/IT
Loop | Loop fragment while guard is true can also write loop(n) to indicate looping n times
---|---
Opt | Optional fragment that executes if guard is true
Par | Parallel fragment that executes if guard is true.
Region | Critical region within which only one thread can run.

**Basic communication diagram notations:**

**Link:**
It is the path of communication between two objects. It indicate form of navigation and visibility between the object. It is an instance of an association.

**Messages:**
Each message between object is represented with a message expression and a small arrow indicating the direction of messages.

**Messages to “self” or “this”:**
A message can be send from an object to itself with messages flowing along the link.
3. **Explain about the Uml Class Diagram**

**Class diagram:**

Class diagram is used to define classes interface and their associate.

**Design class diagram:**

Class diagram has 2 perspectives

1. It can be design as a domain model.
2. It can be create as a design model.

**Classifiers:**

Classifiers are a model that described the behavior and structure. It is also a specialized thing in a class diagram. They is the generalization of many of the elements including classes, interfaces and use case.

**Two types of classifiers**

1. Regular classes
2. Interface.

**Uml Attributes:**

Attributes of a classifier are shown in several ways

- Attribute text notation such as currentsale:sale
- Association line
- Both together

Prepared By: Mrs. R. BARONA, AP/IT
Syntax:

visibility name:type multiplicity=default{property- string}

Guidelines for attribute:

- Attribute are assumed private if no visibility is given
- Attributes as association line has the following style
  - A navigability arrow pointing from source object to target object
  - Multiplicity should be specified at the target end and not at the source end
  - Role name which is also called as attribute name is given at the target end
  - No association name should be specified
- The UML meta data model allows multiplicity and role names at the source end and also allows association name
- As the domain model is not software perspective
  - Anoid using navigation arrow
  - Show association name

Guideline: when to use attribute text versus association lines for attributes

Use attribute text notation for datatype object and association line notation for others. These two representations are semantically equal. A datatype refers to object for which unique identity is not important.

Operations and methods:

operation:

Third compartment of uml class diagram shows the operations

Syntax:

visibility name(parameter list){property-string}
Here there is no return type element. With return type element, the operation syntax

```
Visibility name(parameter list): return type {property-string}
```

**Qualified association:**

A qualified association has a qualifier that is used to select an object from a large set of related objects based upon the qualifier key.

**Association class:**

Association class can create an association itself as a class and model it with attributes, operations and other features.

**Singleton class:**

A class with only one instance is known as singleton class.

**Active class:**

An active object runs and controls its own thread of execution then the class of an active object is active class.
4. Explain about System Sequence Diagram

SSD is a fast and easily created artifact that illustrate input and output events related to the system.

Sequence Diagrams

UML provides a graphical means of depicting object interactions over time in Sequence Diagrams. These typically show a user or actor, and the objects and components they interact with in the execution of a use case. One sequence diagram typically represents a single Use Case 'scenario' or flow of events.

Sequence diagrams are an excellent way to document usage scenarios and to both capture required objects early in analysis and to verify object usage later in design. Sequence diagrams show the flow of messages from one object to another, and as such correspond to the methods and events supported by a class/object.

The diagram illustrated below shows an example of a sequence diagram, with the user or actor on the left initiating a flow of events and messages that correspond to the Use Case scenario. The messages that pass between objects will become class operations in the final model.

Prepared By: Mrs.R.BARONA, AP/IT
SSD:

Use case describes how external actors interact with the system. During interaction an actor generates system events to a system representing some system operation to handle the system event. Sequence diagram also says the actor interaction and the operations initiated by them.

SSD shows one particular scenario of usecase

- Events that actors generate
- Their order
- Inter system event

Why draw SSD?

We have to design the software to handle these events and execute a response. These works are done by SSD.

Basically software system reacts three things

- External events from actor
- Time events
- Fault or exceptions

External events:

- It affects the system events.
- Used for analyzing the system behavior.
- It explains what a system does and it does not explain what a system doing

Applying UML: Sequence Diagram

Interaction frame is used to show loops in sequence diagram.
Relationship between ssd and usecase:

SSD shows the system event for one scenario of usecase. Usecase generates the system the system events.

How to name system events and operations:

Scanitem(item ID)  enteritem(item Id)

From this which one is better?

System event should be expressed at the abstract level of abstraction and not in terms of physical input devices. Hence enteritem(item Id) is better than scanitem(item Id) because it captures the intent of the operation. scanitem(itemId) explains about what interface is used to capture the system event.

How to model ssd involving other external system:

SSD is used to illustrate collaboration between system such as collaboration between pos and external credit payment.

What ssd information to be placed in the glossary:

The elements shown in SSD are terse it may be operation name, parameters and written data. We need glossary because we need proper explanation so that during software design what it is coming in and what is going out Eg: change dues, receipt.

Prepared By: Mrs.R.BARONA, AP/IT
5. Explain about the Basic Sequence Diagram And Notations

Sequence diagram:

It is in the fence format. New objects are added in the right side. Messages are passed in the time ordering.

Basic sequence diagram notations:

Lifeline boxes and lifeline:

The lifeline boxes including a vertical line extended below. These are the actual lifeline of the objects. The lifeline is indicated as dashed line. The lifeline boxes are used for indicating boxes and objects.

Messages:

Each message between the objects is represented with a message expression on a filled arrowed solid line between the vertical lifelines. Starting message is called as the found message which is shown with the opening solid ball and sender will not be specified and it is not shown where the message is coming from.

Focus of control and execution specification:

Execution specification shows the focus of control. It is automatic in a UML case tools.

Illustrating reply or return:

Two ways to show the return result from a messages

1. Using the message syntax return var=message(parameter).

2. Using a reply(or return)messages line at the end of an activation bar.

Prepared By: Mrs.R.BARONA, AP/IT
Object lifeline and destruction:

<<destroy>> stereotype message with a large x and it also explains the short lifeline of an object.

Diagram frames in uml sequence diagram:

Frames are also called as interaction frame or diagram frames. It is a region or fragmentation of the diagram. Frame has label, operator and guard.

Types of frames:

<table>
<thead>
<tr>
<th>Frame operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt</td>
<td>It is alternative fragment for mutual exclusion conditional logic expressed in guards.</td>
</tr>
<tr>
<td>Loop</td>
<td>Loop fragment while guard is true can also write loop(n) to indicate looping n times</td>
</tr>
<tr>
<td>Opt</td>
<td>Optional fragment that executes if guard is true</td>
</tr>
<tr>
<td>Par</td>
<td>Parallel fragment that executes if guard is true.</td>
</tr>
<tr>
<td>Region</td>
<td>Critical region within which only one thread can run.</td>
</tr>
</tbody>
</table>

Conditional messages:

Opt frame is placed around one or more message. The guard is placed over the object lifeline. It is the one-conditional message. To show a single conditional message with an opt frame box is not necessary and second diagram is very simple and is probably popular for years.
Mutually exclusive conditional messages:

For this we are using alt frame. For mutually exclusion conditional messages we use the alt frame and it is placed around where the conditional arises.

Iteration over a collection:

Action Box:

Action box may containing arbitrary statements that is incrementary or decrementary selector expression line item[i] in lifeline is used to select one object from the group.

Guidelines:

Any sequence diagram can be surrounded with an sd frame to name it. It is referenced by using the ref frame by another sequence diagram.
6. Explain Logical Architecture Refinement

Introduction:

Logical architecture and the layers pattern was introduced. There are other types in these packages, only a few are shown to indicate noteworthy aspects. The foundation layer was not shown in this view, the architect decided it did not add interesting information, even though the development team will certainly be adding some information classes, such as more advanced string manipulation utilities. For now, a separate application layer is not used. The responsibilities of control or session objects in the application are handled by the register object. The architect will add an application layer in a later iteration as the behavior grows in complexity and alternative client interface are introduced.

Inter Layer And Inter Package Coupling:

Applying UML

- Observe that dependency lines can be used to communicate coupling between packages or types in a package. Plain dependency lines are excellent when the communication does not care to be more specific on the exact dependency but just wants to highlight general dependencies.

- Note also the use of a dependency line emitting from a package rather than a particular type such as from the sales package to posrule engine facade class and the domain package to the log4J package. This is useful when either the specific dependent type is not interesting or the communication wants to suggest that many elements of the package may share that dependency.
Inter Layer And Inter Package Scenarios:

Package diagram show static information. In spirit architectural view which hides uninteresting details and emphasizes what the architect wants to convey an interaction diagram in the logical view of the architecture focuses on the collaborations as they cross layer and package boundaries, A set of interaction diagrams that illustrate architecturally significant scenario.

Collaborations with layers pattern:

Two design decisions at an architecture level are

- What are the big parts?
- How are they connected?

Simple Packages Versus Subsystems:

Some packages or layers are not just conceptual group of things that are true subsystems with behavior and interface. The pricing package is not a subsystem; it simply groups the factory and strategies used in pricing. Likewise with foundation packages such as java.util On the other hand, the persistence POSR ule Engine and Jess packages are subsystems. They are discrete engines with cohesive responsibilities that do work.

Facade:

For packages that represent subsystems, the most common pattern of access is facade, a GOF design pattern. The façade should not normally expose many lowlevel operations. Rather it is desirable for the façade to expose a small number of high level operations, the coarse grained services when a faced does expose many low level operations it tends to became in cohesive.

Prepared By: Mrs.R.BARONA, AP/IT
Relaxed Layered Coupling:

The layers in the most layered architecture are not coupled in the same limited sense as a network protocol based on the OSI layer model. In the protocol model, there is strict restriction that elements of layer N only access the services of the immediate lower layer N-1.

Comments on typical coupling between layers

- All higher layers have dependencies on the technical service and foundation layer
- It is primarily the domain layer that has dependency on the Business Infrastructure Layer
- The presentation layer makes calls on the application layer, which makes service calls on the domain

Terminology: Tiers, Layers And Partitions

The original notion of a tier in architecture was a logical layer, not a physical node but the word has become widely used to mean a physical processing node such as the “client tiers”.

Information Systems: The Classic Three Tier Architecture

A classic description of the vertical tiers in a three tier architecture is

- Interface - windows, reports and so on
- Application logic - tasks and rules that govern the process
- Storage - persistent storage mechanism.

Prepared By: Mrs. R. BARONA, AP/IT
7. **What can be modeled using sequence diagrams?**

Sequence diagrams are particularly useful for modeling.

**Complex interactions between components**

Sequence diagrams are often used to design the interactions between components of a system that need to work together to accomplish a task. They are particularly useful when the components are being developed in parallel by different teams (typical in wireless and telephony systems) because they support the design of robust interfaces that cover multiple scenarios and special cases.

**Use case elaboration**

Usage scenarios describe a way the system may be used by its actors. The UML sequence diagram can be used to flesh out the details of one or more use cases by illustrating visually how the system will behave in a particular scenario. The use cases along with their corresponding sequence diagrams describe the expected behavior of the system and form a strong foundation for the development of system architectures with robust interfaces.

**Distributed & web-based systems**

When a system consists of distributed components (such as a client communicating with one or more servers over the Internet), sequence diagrams can be used to document and validate the architecture, interfaces and logic of each of these components for a set of usage scenarios.

Prepared By: Mrs.R.BARONA, AP/IT
Complex logic

UML sequence diagrams are often used to model the logic of a complex feature by showing the interactions between the various objects that collaborate to implement each scenario. Modeling multiple scenarios showing different aspects of the feature helps developers take into account special cases during implementation.

State machines

Telecom, wireless and embedded systems make extensive use of state machine based designs where one or more state machines communicate with each other and with external entities to perform their work. For example, each task in the protocol stack of a cellular phone goes through a series of states to perform actions such as setup a call or register with a new base station.

Similarly the call processing components of a Mobile Switching Center use state machines to control the registration and transfer of calls to roaming subscribers. Sequence diagrams (or call flows as they are commonly referred to in the telecom and wireless industry) are useful for these types of applications because they can visually depict the messages being exchanged between the components and their associated state transitions.

Benefits of using UML sequence diagrams

These are some of the main benefits of using UML sequence diagrams.

1. Help you discover architectural, interface and logic problems early.
Because they allow you to flesh out details before having to implement anything, sequence diagrams are useful tools to find architectural, interface and logic problems.
early on in the design process. You can validate your architecture, interfaces, state machine and logic by seeing how the system architecture would handle different basic scenarios and special cases.

This is particularly true for systems involving the interaction of components that are being implemented in parallel by different teams. In the cell phone example, each task would typically be implemented by a separate team. Having a set of sequence diagrams describing how the interfaces are actually used and what messages/actions are expected at different times gives each team a consistent and robust implementation plan.

You can also document how special cases should be handled across the entire system. The very act of creating the sequence diagrams and making them work with your architecture is valuable because it forces you to think about details such as interfaces, states, message order, assignment of responsibilities, timers/timeouts and special/error cases ahead of time.

2. **Collaboration tool.** Sequence diagrams are valuable collaboration tools during design meetings because they allow you to discuss the design in concrete terms. You can see the interactions between entities, various proposed state transitions and alternate courses/special cases on paper as you discuss the design. In our experience, having a concrete design proposal during design meetings greatly enhances the productivity of these meetings even if the proposed design has problems.

You can narrow down the problems and then make corrections to solve them. The proposal serves as a concrete starting point for the discussion and as a place to capture proposed changes. **Sequence diagram editor** makes it so easy to edit your sequence
diagrams that you could even make the corrections in real time during the meeting and instantly see the result of the changes as you make them.

3. **Documentation.** Sequence diagrams can be used to document the dynamic view of the system design at various levels of abstraction, which is often difficult to extract from static diagrams or even the complete source code. The diagrams can abstract much of the implementation detail and provide a high level view of system behavior. Below is a sequence diagram for making a hotel reservation. The object initiating the sequence of messages is a Reservation window.
8. **Explain about UML Package Diagram?**

A package diagram is a UML diagram composed only of packages and the dependencies between them. A package is a UML construct that enables you to organize model elements, such as use cases or classes, into groups. Packages are depicted as file folders and can be applied on any UML diagram. Create a package diagram to:

- Depict a high-level overview of your requirements (overviewing a collection of UML Use Case diagrams)
- Depict a high-level overview of your architecture/design (overviewing a collection of UML Class diagrams).
- To logically modularize a complex diagram.
- To organize Java source code.

There are guidelines for:

a. Class Package Diagrams
b. Use Case Package Diagrams
c. Packages
I. Class Package Diagrams

A class package diagram

Create UML Component Diagrams to Physically Organize Your Design.

Place Subpackages Below Parent Packages.

Vertically Layer Class Package Diagrams.

Create Class Package Diagrams to Logically Organize Your Design.

**Error! Reference source not found.** Depicts a UML Class diagram organized into packages. In addition to the package guidelines presented below, apply the following heuristics to organize UML.
Class diagrams into package diagrams:

- Place the classes of a framework in the same package.
- Classes in the same inheritance hierarchy typically belong in the same package.
- Classes related to one another via aggregation or composition often belong in the same package.
- Classes that collaborate with each other a lot, information that is reflected by your UML Sequence diagrams and UML Collaboration diagrams, often belong in the same package.

II. Use Case Package Diagrams

Use cases are often a primary requirement artifact in object-oriented development methodologies, this is particularly true of instantiations of the Unified Process, and for larger projects package diagrams are often created to organize these usage requirements.
1. Create Use Case Package Diagrams to Organize Your Requirements
2. Include Actors on Use Case Package Diagrams
3. Horizontally Arrange Use Case Package Diagrams

III. Packages

The advice presented in this section is applicable to the application of packages on any UML diagram, not just package diagrams.

1. Give Packages Simple, Descriptive Names
2. Apply Packages to Simplify Diagrams
3. Packages Should be Cohesive
4. Indicate Architectural Layers With Stereotypes on Packages
5. Avoid Cyclic Dependencies Between Packages
6. Package Dependencies Should Reflect Internal Relationships
UNIT-IV

1. **Explain Creator Pattern with an example.**

**Creator:**

The creation of objects is one of the most common activities in an object-oriented system. Consequently, it is useful to have a general principle for assignment of creation responsibilities. Assigned well, the design can support low coupling increased clarity, encapsulation, and reusability.

**Problem:**

Who should be responsible for creating a new instance of some class?

**Solution:**

Assign class B the responsibility to create an instance of class A if one or more of the following is true:

- B aggregates A objects.
- B contains A objects.
- B records instances of A objects.
- B closely uses A objects.
- B has the initializing data that will be passed to A when it is created.
- B is a creator of A objects.

**Example:** NextGen POS Example

**Problem:** Who should create a SalesLineItem

Prepared By: Mrs.R.BARONA, AP/IT
Solution:

Since a Sale contains (in fact, aggregates) many SalesLineItem objects, the Creator pattern suggests that Sale is a good candidate to have the responsibility of creating SalesLineItem instances.

Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes. In general, a class B should be responsible for creating instances of class A if one, or preferably more, of the following apply:

- Instances of B contains or compositely aggregates instances of A
- Instances of B record instances of A

Prepared By: Mrs.R.BARONA, AP/IT
Instances of B closely use instances of A
Instances of B have the initializing information for instances of A and pass it on creation.

Discussion of Creator pattern
- Responsibilities for object creation are common
- Connect an object to its creator when:
  - Aggregator aggregates Part
  - Container contains Content
  - Recorder records
  - Initializing data passed in during creation

Creator benefits
- Low coupling is supported, which implies lower maintenance dependencies and higher opportunities for reuse.
- Coupling is properly not increased because the created class is likely already visible to the creator class due to the existing associations that motivated its choice as creator.

Related Patterns
- Low coupling
- Concrete Factory & Abstract Factory

Prepared By: Mrs.R.BARONA, AP/IT
2. Explain ‘Information Expert’ with an example.

Information Expert

By Information expert we should look for the class of objects that has the information needed to determine the total.

Information Expert is a principle used to determine where to delegate responsibilities. These responsibilities include methods, computed fields and so on. Using the principle of Information Expert a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored. Information Expert will lead to placing the responsibility on the class with the most information required to fulfill it.

Problem: What is a general principle of assigning responsibilities to objects?

Solution: Assign a responsibility to the information expert – a class that has the information necessary to fulfill the responsibility.

Example:

Problem: Who should be responsible for knowing the total price for a sale?

Solution: Let Sale do it because it knows all of its SaleLineItems. For example, to get the sale total, Sale gets subtotals from SaleLineItem which get prices from ProductDescription.
We see that Sale already has the SalesLineItem objects needed to calculate the total.

But to get the price of a SalesLineItem, we also need the ProductDescription's price attribute. SalesLineItem has the needed information.
To fulfill the responsibility to know its subtotal, the SalesLineItem needs to collaborate with ProductDescription.

Expert benefits

- Information encapsulation is maintained since obj. use their own info to fulfill tasks. This usually supports low coupling, which leads to more robust and maintainable systems.
Behavior is distributed across the classes that have the required info, thus encouraging more cohesive “lightweight” class definitions that are easier to understand and maintain.

**Related Patterns**

- Low coupling
- High cohesion
3. **Explain ‘Low Coupling’ with an example.**

**Low Coupling**

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other element. If there is coupling or dependency, then when the depended-upon element changes, the dependent may be affected.

**Problem: **How to support a low dependency, low change impact, and increased reuse?

**Solution:**
- Assign a responsibility so that coupling remains low.
- Evaluate design alternatives
- Choose option that minimizes coupling
- “Simple chain of command”

**Example: **NextGen POS Example

**Problem: **Who should create a Payment and associate it with a Sale?

**Solution:**
- By Expert, let Register create Payment info, then associate it with the Sale
  Register & Sale now coupled to Payment
- By Low Coupling, Register delegates Payment creation to Sale Only Sale coupled to Payment
Design: 1

On the one hand, we might give Register the responsibility to create the Payment:

```
makePayment()  : Register  1: create()  p : Payment
          ↓                     ↓
          2: addPayment(p)     : Sale
```

Design: 2

On the other hand, we might give Sale the responsibility to create the Payment:

```
makePayment()  : Register  1: makePayment()  : Sale
          ↓                             ↓
          1. create()                 : Payment
```
Which is better? It is unavoidable that Sale is coupled to Payment. But the first design creates an unnecessary coupling of Sale to Register. All else being equal, we prefer the second.

**Low Coupling benefits**
- Not affected by changes in other components
- Simple to understand in isolation
- Convenient to reuse
- Classes are
  - More independent
  - Easier to reuse
  - Easier to understand
  - Easier to maintain

**Related Patterns**
- Protected Variation
4. Explain pattern ‘Controller’

Controller is the first obj. (class) beyond the UI layer that is responsible for receiving or handling a system operation message.

The Controller pattern assigns the responsibility of dealing with system events to a non-UI class that represent the overall system or a use case scenario. A Controller object is a non-user interface object responsible for receiving or handling a system event.

A use case controller should be used to deal with all system events of a use case, and may be used for more than one use case (for instance, for use cases Create User and Delete User, one can have one User Controller, instead of two separate use case controllers).

It is defined as the first object beyond the UI layer that receives and coordinates ("controls") a system operation. The controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. It should not do much work itself.

The GRASP Controller can be thought of as being a part of the Application/Service layer (assuming that the application has made an explicit distinction between the App/Service layer and the Domain layer) in an object-oriented system with common layers.
**Problem:**

The UI & domain layers should be loosely coupled. What is the 1st object should receive and coordinate messages (system operation) between the UI layer and other domain objects?

**Solution :**

Assign responsibility to an object representing one of these choice:

- The overall ‘system’ or a ‘root’ object (Store, Bank)
- A device that the SW is running within (BankATM)
- A major subsystem (AccountingSystem)
- A major use case scenario (GameHandler)
- ‘Single channel of communication between layers’

**Example:** NextGen POS Example

**Problem :** Which object in the domain layer should be responsible for the system operations?

**Solution:**

- Select the ‘root’ object in the POS domain model (Store)
- Select an object that represents a key device (Register)
- Create an object that handles the use case (SaleHandler)
**Example:** for the enterItem operation, we have a situation like this:

![Diagram of enterItem operation](image)

Which class of object should be responsible for receiving this system event message?

It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.

The controller is a kind of "facade" onto the domain layer from the interface layer.

Some possible design alternatives:

- `enterItem(id, quantity)` : Register
- `enterItem(id, quantity)` : ProcessSaleHandler

Prepared By: Mrs.R.BARONA, AP/IT
Combining with the other system operations, we can see the consequences of each design alternative:

Some tips about controllers:

- The controller should be seen as a facade into the domain layer.
- Use the same controller for all system events involved in a use case.
- Controllers will often be stateful, maintaining information about the current progress of the use case.
- For purposes of cohesion, controllers should primarily delegate, (forward responsibility to other objects) wherever possible.
- In Java EE, controllers will oftentimes be stateful session Enterprise Java Beans.
- The controller may delegate directly to domain layer objects, or if it is in a separate application logic layer, it may make requests to other controllers in the pure domain layer.

Prepared By: Mrs.R.BARONA, AP/IT
Some controller antipatterns:

- A single controller with "too many" system events.
- No delegation.
- Duplication of information found elsewhere in the domain layer.

Don't allow direct manipulation of domain objects in the UI layer!

Example (good design):
**Example (bad design):**

**Controller benefits**
- Increased potential for reuse and pluggable interfaces
- Opportunity to reason about the state of the UC

**Related Patterns**
- Command: In a msg handling sys., each msg may be represented and handled by a separate Command obj
- Façade
- Layers (POSA pattern) placing domain logic in the domain layer.
- Pure Fabrication

Prepared By: Mrs.R.BARONA, AP/IT
5. Define pattern ‘High Cohesion’

Cohesion measures how functionality related the operations of a software elements are, and also measures how much work a software element is doing.

Problem: How to keep objects focused, understandable, manageable, maintainable and a side effect?. How to support low coupling?

Solution:

- Assign responsibilities so that cohesion remains high and object’s responsibilities are closely related
- Evaluate alternatives to optimize cohesion
- ‘Don’t spread yourself too thin’
- ‘Teamwork’

Example:

For the POS system, when we design for the makePayment() system operation:
We already saw that this creates an unnecessary coupling between Register and Payment. But it also reduces the cohesion of Register.

By delegating Payment creation to Sale, we not only decrease coupling, but we also increase the cohesion of Register.

Since Register has fewer responsibilities, it is more cohesive.

**High Cohesion benefits**

- Clarity and ease of comprehension of the design is increased
- Maintenance and enhancements are simplified
- Low coupling is often supported
- Reuse of fine-grained, highly related functionality is increased because a cohesive class can be used for a very specific purpose.
6. **Explain Designing for Visibility**

In object-oriented design, there is a notation of visibility for attributes and operations. UML identifies four types of visibility: public, protected, private, and package. The UML specification does not require attributes and operations visibility to be displayed on the class diagram, but it does require that it be defined for each attribute or operation.

To display visibility on the class diagram, you place the visibility mark in front of the attribute's or operation's name. Though UML specifies four visibility types, an actual programming language may add additional visibilities, or it may not support the UML-defined visibilities. Table 4 displays the different marks for the UML-supported visibility types.

- during the creation of interaction diagrams (especially when considering the coupling pattern:
  - It is obvious that for an object to call a method of another object the caller object needs a reference (we say must have visibility) to it;
  - Visibility is the ability of one object to see or have reference to another;
  - Hence, when creating a design of interacting objects, it is necessary to ensure that the necessary visibility is present to support message interaction.
- For example, the following interaction diagram implies that the ProductCatalog instance is visible to the Register instance:

Prepared By: Mrs.R.BARONA, AP/IT
This visibility requirement was reflected in the design class diagram for the POS application.

- Visibility: the ability of one object to “see” or have a reference to another object. Visibility is required for one object to message another.
Visibility is the ability of an object to “see” or have a reference to another object.

In order for a sender object to send a message to a receiver object, it must have visibility to the receiver.

Visibility must be established; it doesn’t happen automatically.

There are four common types of visibility:

- Attribute visibility.
- Parameter visibility.
- Local visibility.
- Global visibility.

**Attribute visibility**

- When a client object defines an attribute that references an instance of a server object.
- Required if the client needs a permanent memory of the server object.
- Attribute visibility from A to B exists when B is an attribute of A.
  - It is a relatively permanent visibility because it persists as long as A and B exist.
  - This is a very common form of visibility in object-oriented systems.

Prepared By: Mrs.R.BARONA, AP/IT
For example:

```java
class Register {
    ...
    private ProductCatalog catalog;
    ...
}
```

```java
public void enterItem(itemId, qty) {
    ...
    desc = catalog.getProductDesc(itemId);
    ...
}
```

**Parameter visibility**

- When a client object receives a parameter that refers to an instance of a server object.
- Required if the client needs a temporary connection to the server object.
- Parameter visibility from A to B exists when B is passed as a parameter to a method of A.
  - It is a relatively temporary visibility because it persists only within the scope of the method.
For example:

```
makeLineItem(ProductDescription desc, int qty)
{
    ...
    sl = new SalesLineItem(desc, qty);
    ...
}
```

- Within the scope of the makeLineItem method, the Sale has parameter visibility to a ProductDescription; thereafter within the constructor of SalesLineItem, this parameter visibility is transformed into attribute visibility.

**Local visibility**

- In a method of the client object, when a local variable is assigned a new instance or an existing object.
- Required if the client needs a temporary connection to the server object.
- Local visibility from A to B exists when B is declared as a local object within a method of A.

  ✓ It is a relatively temporary visibility because it persists only within the scope of the method.

Prepared By: Mrs.R.BARONA, AP/IT
For example:

```java
enterItem(id, qty) {
    ...
    // local visibility via assignment of returning object
    ProductDescription desc = catalog.getProductDesc(id);
    ...
}
```

Global visibility

- Global visibility from A to B exists when B is global to A.
  - It is a relatively permanent visibility because it persists as long as A and B exist;
  - It greatly increases the potential coupling within a design;

It can be achieved by using global variables or better, by using the singleton pattern.

Permanent visibility to an object via:

- Global variable
- Singleton pattern
7. **Explain Applying GOF Design patterns**

Design patterns represent common software problems and the solutions to those problems in a formal manner. They were inspired by a book written by architect Christopher Alexander. Patterns were introduced in the software world by another book: "Design Patterns: Elements of Reusable Object-Oriented Software", by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

These people were nicknamed the "Gang of Four" for some mysterious reason. The Gang of Four describes 23 design patterns. With patterns you don't have to reinvent the wheel and get proven solutions for frequently encountered problems. Many books and articles have been written on this subject.

This means that design patterns are becoming common knowledge, which leads to better communication. To summarize design patterns save time, energy while making your life easier. Singleton The singleton pattern deals with situations where only one instance of a class must be created. Take the case of a system administrator or superuser.

This person has the right to do everything in a computer system. In addition we will also have classes representing normal users. Therefore we must ensure that these classes have no access to the super user constructor. The solution to this problem in C++ and Java is to declare the superuser constructor private. The superuser class itself has a private static attribute sysadmin, which is initialized using the class constructor. Now we get an instance of the super user class with a public static method that returns sysadmin. Here is the class diagram:

Prepared By: Mrs.R.BARONA, AP/IT
Factory

The Factory Method pattern deals with situations where at runtime one of several similar classes must be created. Visualize this as a factory that produces objects. In a toy factory for instance we have the abstract concept of toy. Every time we get a request for a new toy a decision must be made - what kind of a toy to manufacture. Similarly to the Singleton pattern the Factory Method pattern utilizes a public static access or method. In our example the abstract Toyfactory class will have a getInstance() method, which is inherited by its non abstract subclasses.

Adapter

Sometimes you will have two classes that can in principle work well together, but they can't interface with each other for some reason. This kind of problem occurs when travelling abroad and you carry an electric shaver with you. Although it will work perfectly when you are at home. There can be problems in a foreign country, because of a different standard voltage.

The solution is to use an adapter. Let's turn our attention back to the software domain. Here we will have an interface which defines new methods for example getElectricity2. An adapter class will wrap around the Shaver class. The adapter class will implement the interface with the new method.

Prepared By: Mrs.R.BARONA, AP/IT
Proxy

The Proxy pattern deals with situations where you have a complex object or it takes a long time to create the object. The solution to this problem is to replace the complex object with a simple 'stub' object that has the same interface. The stub acts as a body double. This is the strategy used by the Java Remote Method Invocation API. The reason to use the proxy pattern in this case is that the object is on a remote server causing network overhead. Other reasons to use the proxy can be restricted access (Java applets for example) or time consuming calculations.

Decorator

Prepared By: Mrs.R.BARONA, AP/IT
The Decorator is usually a subclass, that is a body double for its superclass or another class with identical interface. The goal of the Decorator pattern is to add or improve the capabilities of the super class.

**Composite**

The composite is often encountered in GUI packages like for instance the Java Abstract Windwing Toolkit (AWT) or Microsoft Foundation (MFC) classes. All objects in this pattern have a common abstract superclass that describes basic object conduct. The base class in the MFC hierarchy is CObject. It provides functions for debugging and serialization. All the MFC classes even the most basic ones inherit these facilities.

**Observer**

An application with Model - View - Controller setup usually uses the Observer

Prepared By: Mrs.R.BARONA, AP/IT
Pattern. In a Java webserver environment the model will be represented by Java classes encompassing the business logic, the view is represented by Java Server Pages which display HTML in the client's browser and we will have a Servlets as Controllers.

The observer pattern strategy is to have view components take subscriptions for their model. This ensures that they will get notified if the model changes.

![Observer and MVC Class Diagram](image)

**Template**

In the good old days before OOP writing functions was the recommended thing to do. A sort algorithm would be implement by half dozen of functions, one sort function for integers, one sort function for floating points, one sort function for doubles etc. These functions are so similar that nobody in their right mind will type them letter by letter.

Instead a programmer will write a template and copy the template several times. After that it's just a matter of writing down datatypes as appropriate. Thanks to OOP and the Template Design Pattern less code is required for this task. First we need to define an abstract Template class let's call it SortTemplate and it will have methods sort, compare and process (performs one cycle of the algorithm).
Then we define concrete classes for each datatype. These classes are subclasses of SortTemplate and implement the compare and process methods.

**Strategy**

The Strategy Design Pattern makes it possible choose an implementation of an algorithm at run time. The implementation classes implement the same interface. In the case of the Java AWT Layout classes, the common interface is LayoutManager.
Summary

Design patterns are a hot research item. New patterns are emerging every day. In the future design patterns will be integrated in development tools.

The main advantages of design patterns:

- Provide proven solutions
- Simplify complex problems
- Improve communication
8. **Explain General Responsibility Assignment Software Patterns**

**General Responsibility Assignment Software Patterns** (or **Principles**), abbreviated **GRASP**, consists of guidelines for assigning responsibility to classes and objects in object-oriented design.

The different patterns and principles used in GRASP are: Information Expert, Creator, Controller, Low Coupling, High Cohesion, Polymorphism, Pure Fabrication, Indirection, Protected Variations. All these patterns answer some software problem, and in almost every case these problems are common to almost every software development project. These techniques have not been invented to create new ways of working but to better document and standardize old, tried-and-tested programming principles in object-oriented design.

It has been said that "the critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology". Thus, GRASP is really a mental toolset, a learning aid to help in the design of object-oriented software.

**Creator**

Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.
In general, a class B should be responsible for creating instances of class A if one, or preferably more, of the following apply:

- Instances of B contains or compositely aggregates instances of A
- Instances of B record instances of A
- Instances of B closely use instances of A
- Instances of B have the initializing information for instances of A and pass it on creation.

**Information Expert**

Information Expert is a principle used to determine where to delegate responsibilities. These responsibilities include methods, computed fields and so on. Using the principle of Information Expert a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored. Information Expert will lead to placing the responsibility on the class with the most information required to fulfill it.

**Controller**

The **Controller** pattern assigns the responsibility of dealing with system events to a non-UI class that represent the overall system or a use case scenario. A Controller object is a non-user interface object responsible for receiving or handling a system event. A use case controller should be used to deal with all system events of a use case, and may be used for more than one use case (for instance, for use cases Create User and Delete User, one can have one User Controller, instead of two separate use case controllers).

Prepared By: Mrs.R.BARONA, AP/IT
It is defined as the first object beyond the UI layer that receives and coordinates ("controls") a system operation. The controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. It should not do much work itself.

The GRASP Controller can be thought of as being a part of the Application/Service layer (assuming that the application has made an explicit distinction between the App/Service layer and the Domain layer) in an object-oriented system with common layers.

**Low Coupling**

**Low Coupling** is an evaluative pattern, which dictates how to assign responsibilities to support:

- low dependency between classes;
- low impact in a class of changes in other classes;
- high reuse potential

**High Cohesion**

**High Cohesion** is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of Low Coupling. High cohesion means that the responsibilities of a given element are strongly related and highly focused.

Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system. Alternatively, low cohesion is a situation in which a given element has too many unrelated responsibilities. Elements with low
cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and adverse to change.

**Polymorphism**

According to **Polymorphism**, responsibility of defining the variation of behaviors based on type is assigned to the types for which this variation happens. This is achieved using polymorphic operations.

**Pure Fabrication**

A **pure fabrication** is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived (when a solution presented by the Information Expert pattern does not). This kind of class is called "Service" in Domain-driven design.

**Indirection**

The **Indirection** pattern supports low coupling (and reuse potential) between two elements by assigning the responsibility of mediation between them to an intermediate object. An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the Model-view-controller pattern.

**Protected Variations**

The **Protected Variations** pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

**Visibility**

In object-oriented design, there is a notation of visibility for attributes and
operations. UML identifies four types of visibility: public, protected, private, and package.

The UML specification does not require attributes and operations visibility to be displayed on the class diagram, but it does require that it be defined for each attribute or operation.

To display visibility on the class diagram, you place the visibility mark in front of the attribute's or operation's name. Though UML specifies four visibility types, an actual programming language may add additional visibilities, or it may not support the UML-defined visibilities.

Visibility: The ability of one object to “see” or have a reference to another object. Visibility is required for one object to message another
UNIT-V

1. Explain UML state diagrams with an example?

State machine diagrams:

A UML statechart diagram illustrates the interesting events and states of an object, and the behavior of an object in reaction to an event. Transitions are shown as arrows, labeled with their event. States are shown in rounded rectangles.

When to Use: State Diagrams

Use state diagrams to demonstrate the behavior of an object through many use cases of the system. Only use state diagrams for classes where it is necessary to understand the behavior of the object through the entire system. Not all classes will require a state diagram and state diagrams are not useful for describing the collaboration of all objects in a use case. State diagrams are other combined with other diagrams such as interaction diagrams and activity diagrams.

How to Draw: State Diagrams

State diagrams have very few elements. The basic elements are rounded boxes representing the state of the object and arrows indicting the transition to the next state. The activity section of the state symbol depicts what activities the object will be doing while it is in that state.
All state diagrams begin with an initial state of the object. This is the state of the object when it is created. After the initial state, the object begins changing states. Conditions based on the activities can determine what the next state the object transitions to.

Below is an example of a state diagram that might look like for an Order object. When the object enters the Checking state, it performs the activity "check items." After the activity is completed, the object transitions to the next state based on the conditions [all items available] or [an item is not available]. If an item is not available, the order is canceled. If all items are available, then the order is dispatched. When the object transitions to the Dispatching state, the activity "initiate delivery" is performed. After this activity is complete, the object transitions again to the Delivered state.
State diagrams can also show a super-state for the object. A super-state is used when many transitions lead to the a certain state. Instead of showing all of the transitions from each state to the redundant state a super-state can be used to show that all of the states inside of the super-state can transition to the redundant state. This helps make the state diagram easier to read. The diagram below shows a super-state. Both the Checking and Dispatching states can transition into the Canceled state, so a transition is shown from a super-state named Active to the state Cancel. By contrast, the state Dispatching can only transition to the Delivered state, so we show an arrow only from the Dispatching state to the Delivered state.
Elements of state machine diagrams

Events

An event is a significant or noteworthy occurrence. An event causes a transition from one state to another state.

Example: A telephone receiver is taken off the hook

Transition

A transition is a relationship between two states that indicates that when an event occurs, the object moves from the prior state to the subsequent state.

Example: When the event "off hook" occurs, transition the telephone from the "idle" to "active" state.

State

A state is the condition of an object at a moment in time—the time between events.

Example: A telephone is in the state of being "idle" after the receiver is placed on the hook and until it is taken off the hook.
Example

State machine diagram for telephone
2. Explain Operation contracts with an example?

**Operation Contracts:**

Operation Contracts gives the more detailed or precise description of system behavior.

Operation Contracts use pre- and post- conditions to describe detailed changes to objects in the domain model, as a result of the system operation.

A UML Operation contract identifies system state changes when an operation happens. Effectively, it will define what each system operation does. An operation is taken from a system sequence diagram. It is a single event from that diagram. A domain model can be used to help generate an operation contract.

The domain model can be marked as follows to help with the operation contract:
- Green - Pre existing concepts and associations.
- Blue - Created associations and concepts.
- Red - Destroyed concepts and associations.

**Sections of a contract**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Name of operation, and parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross Ref</td>
<td>UCs that this Op occurs within</td>
</tr>
<tr>
<td>Preconditions</td>
<td>Noteworthy assumption about the state of the system or objects in the DM before executing the operation</td>
</tr>
</tbody>
</table>

Prepared By: Mrs.R.BARONA, AP/IT
Post conditions  The most important section. The state of objects in the DM after the completion of the operation.

**Preconditions**

- Describe changes in the state of objects in the Domain Model
- Include instances created, associations formed or broken, and attributes changed
- Post conditions are not actions to perform
- Rather, they are declarations about Domain Model objects that are true when the operation has finished
- Express post conditions in the past tense, to emphasize they are declarations about a state change in the past.

**Example**

Here's an operation contract for the `enterItem` system operation.

<table>
<thead>
<tr>
<th>Contract CO2</th>
<th>enterItem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td><code>enterItem(itemID: ItemID, quantity: integer)</code></td>
</tr>
<tr>
<td>Cross References</td>
<td>Use Cases: Process Sale</td>
</tr>
<tr>
<td>Preconditions</td>
<td>There is a sale underway.</td>
</tr>
<tr>
<td>Post conditions</td>
<td>A SalesLineItem instance sli was created</td>
</tr>
</tbody>
</table>

Prepared By: Mrs.R.BARONA, AP/IT
(instance creation).

- sli was associated with the current Sale (association formed).

- sli.quantity became quantity (attribute modification).

- sli was associated with a ProductDescription, based on itemID match (association formed).

The categorizations such as "(instance creation)" are a learning aid, not properly part of the contract.
3. Explain Mapping design to code with an Example?

- **Implementation Model:** The ultimate deliverables such as Source code, DB definitions, HTML pages etc.

- **In Design:** code is more than mechanical code generation
  - Especially with agile modeling because many details skipped intentionally
  - Developers may have good ideas that improve design

- Implementation in OO language requires writing source code for:
  - Definitions of classes & interfaces: get these from design class diagram. There are many tools can generate them.
  - Definitions of methods

- Work from OOA/D artifacts
  - Create class definitions for Domain Class Diagrams (DCDs) DCD’s depict.
    - Class or interface name
    - Super classes
    - Operation Signature
    - Attributes of class

  These things are enough for creating a basic class definition in OO language.
  - Create methods from Interaction diagrams
    - The sequence of messages in an interaction diagram translates to series of statements in method definitions

Prepared By: Mrs.R.BARONA, AP/IT
Collection classes in code

- Collection classes are implemented with the collection object such as List or map or array.

Exception handling

- In UML, exceptions can be inserted as property strings of messages

Order of implementation

- Start implementing the classes from least-coupled to most-coupled

Design Codes

In this post we’ll see how deployment diagrams are used to model the physical architecture of a system; we’ll start from the most simple use of the deployment diagram in which we only present the nodes and their inter-relationships, and complete the picture by including the components and the applications that run in the nodes.

Connecting the Nodes

Very early in the system life time - deployment diagrams are used to show the nodes (computers, virtual machines) and the external devices (if there are any) which construct the system. A „node“ usually refers to a computer which can be stereotyped as server, client, workstation etc. A „device“ is a subclass of „node“ which refers to a resource with processing capability such as camera, printer, measurement instrument etc. The nodes and the devices are usually wired though the „Communication Path“ connector which illustrates the exchange of signals and messages between both ends.
Notice that the client node is stereotyped as „pc-client” (indicated by the icon) and the server node is stereotyped as „pc-server”. The following diagram shows the deployment architecture of a scalable, fault tolerant „Camera control and image processing” system. The system consist of N servers, load balancer with redundancy, and several clients.

The client machines present live state of all the cameras available in the system, and allow the user to control the cameras and initiate all kind of activities on the servers. The load balancer process the inputs that it receives from the clients and send the appropriate instructions to the appropriate server, it is designed to gracefully scale to increasing number of servers.

Since the load balancer is a single point of failure, a passive load balancer (that maintains copy of the active load balancer state) run in the background, ready to replace the active load balancer in case of a crush. All the servers run the same application, they support different kinds of cameras and can be configured to manage up to 200 cameras of different kinds.

Prepared By: Mrs.R.BARONA, AP/IT
4. Explain guidelines for how to create and write contracts?

Apply the following advice to create contracts:

1. Identify system operations from the SSDs.
   
   For system operations that are complex and perhaps subtle in their results, or which are not clear in the use case, construct a contract.

2. To describe the postconditions, use the following categories:
   
   - instance creation and deletion
   - attribute modification
   - associations formed and broken

Writing Contracts

1. Write the postconditions in a declarative, passive past tense form (was …) to emphasize the observation of a change rather than a design of how it is going to be achieved.

For example:

   (better) A SalesLineItem was created.
   (worse) Create a SalesLineItem.

2. Remember to establish an association between existing objects and with those newly created.
For example:

It is not enough that a new SalesLineItem instance is created when the enterItem operation occurs. After the operation is complete, it should also be true that the newly created instance was associated with Sale. Thus: the SalesLineItem was associated with the Sale (association formed).
5. Explain UML deployment diagrams

Deployment diagram

A deployment diagram shows the assignment of concrete software artifacts (such as executable files) to computational nodes (something with processing services). It shows the deployment of software elements to the physical architecture and the communication (usually on a network) between physical elements.

Deployment diagram shows execution architecture of systems that represent the assignment (deployment) of software artifacts to deployment targets (usually nodes). Nodes represent either hardware devices or software execution environments.

They could be connected through communication paths to create network systems of arbitrary complexity. Artifacts represent concrete elements in the physical world that are the result of a development process and are deployed on nodes.

Note, that components were directly deployed to nodes in UML 1.x deployment diagrams. In UML 2.x artifacts are deployed to nodes, and artifacts could manifest components. So components are now deployed to nodes indirectly through artifacts.

The following nodes and edges are typically drawn in a UML deployment diagram: artifact, association between artifacts, dependency between artifacts, component, manifestation, node, device, execution environment, composition of nodes, communication path, deployment specification, deployment specification dependency, deployment specification association, deployment.
You can find some deployment diagrams examples here:

- Web Application Deployment
- Clustered Deployment of J2EE Web Application
- Apple iTunes Deployment

Artifact

An artifact is a classifier that represents some physical entity, piece of information that is used or is produced by a software development process, or by deployment and operation of a system. Artifact is source of a deployment to a node. A particular instance (or "copy") of an artifact is deployed to a node instance.

Artifacts may have properties that represent features of the artifact, and operations that can be performed on its instances. Artifacts have `fileName` attribute - a concrete name that is used to refer to the artifact in a physical context - e.g. `file name` or `URI`.

Some real life examples of artifacts are:

- model file
- source file
- script
- binary executable file
- text document
- mail message
- table in a database
**Associations Between Artifacts**

Artifacts can be involved in associations to other artifacts, e.g. composition associations. For instance, a deployment descriptor artifact for a component may be contained within the artifact that manifests that component. In that way, the component and its descriptor are deployed to a node instance as one artifact instance.

Application book-club.ear artifact contains EJB user-service.jar artifact and deployment descriptor.

**Dependency Between Artifacts**

Artifacts can be involved in dependency relationship with other artifacts.

Dependency between artifacts is notated in the same way as general dependency, i.e. as a general dashed line with an open arrow head directed from client artifact to supplier artifact.

Prepared By: Mrs.R.BARONA, AP/IT
Artifact Manifestation

Manifestation is an abstraction relationship which represents the concrete physical rendering of one or more model elements by an artifact or utilization of the model elements in the construction or generation of the artifact. An artifact manifests one or more model elements.

Note, that since UML 2.0 artifacts can manifest any packageable element, not just component as it was in previous versions of UML. The artifact owns the manifestations, each representing the utilization of a packageable element.

Specific profiles are expected to stereotype the manifestation relationship to indicate particular forms of manifestation. For example, «tool generated» and «custom code» might be two manifestations for different classes embodied in an artifact.

A manifestation is notated in the same way as abstraction dependency, i.e. as a dashed line with an open arrow head directed from artifact to packageable element, (e.g. to component or package) and is labeled with the keyword «manifest».

Elements in deployment diagram
The basic element in the deployment diagram is node.

Prepared By: Mrs.R.BARONA, AP/IT
There are two types of nodes

1. Device Node

Device node (or device) is a physical (e.g., digital electronic) computing resource with processing and memory services to execute software, such as a typical computer or a mobile phone.

2. Execution Environment node

Execution environment node (EEN) is a software computing resource that runs within an outer node (such as a computer) and which itself provides a service to host and execute other executable software elements.

Example:

- An operating system (OS) an operating system (OS) is software that hosts and executes programs.
- A virtual machine (VM, such as the Java or .NET VM) hosts and executes programs.
- A database engine (such as PostgreSQL) receives SQL program requests and executes them, and hosts/executes internal stored procedures (written in Java or a proprietary language)

Prepared By: Mrs.R.BARONA, AP/IT

Component Diagram

The Component Diagram helps to model the physical aspect of an Object-Oriented software system. It illustrates the architectures of the software components and the dependencies between them.

Component

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces.

Basic Component Diagram Symbols and Notations

Component

A component is a physical building block of the system. It is represented as a rectangle with tabs. Learn how to resize grouped objects like components.

Interface

An interface describes a group of operations used or created by components.
Dependencies

Draw dependencies among components using dashed arrows. Learn about line styles in SmartDraw.

Explain all elements of a Component diagram.

Elements of a component diagram:

Rectangle: A single component is described using a rectangle and having the component’s name inside it. <<Component>>Component Name

Additional compartments: Additional compartments are stacked below the component name.

Interfaces provided/required: Another compartment exists for displaying the interface provided and required by the component.
Relationships:

A lollipop and socket notation is used along with showing dependency arrows. Dependency arrow points towards the needed socket and arrowhead connects with provider’s lollipop.

Subsystem:

A subsystem is represented using a rectangle with stereotype <> subsystem name.
7. Explain UML Component Diagram

Overview:

Component diagrams are different in terms of nature and behaviour. Component diagrams are used to model physical aspects of a system. Now the question is what are these physical aspects? Physical aspects are the elements like executables, libraries, files, documents etc which resides in a node. So component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Purpose:

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities. So from that point component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment. A single component diagram cannot represent the entire system but a collection of diagrams are used to represent the whole.

So the purpose of the component diagram can be summarized as:

- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

Prepared By: Mrs.R.BARONA, AP/IT
How to draw Component Diagram?

Component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executables, libraries etc. So the purpose of this diagram is different, Component diagrams are used during the implementation phase of an application. But it is prepared well in advance to visualize the implementation details.

Initially the system is designed using different UML diagrams and then when the artifacts are ready component diagrams are used to get an idea of the implementation. This diagram is very important because without it the application cannot be implemented efficiently. A well prepared component diagram is also important for other aspects like application performance, maintenance etc.

So before drawing a component diagram the following artifacts are to be identified clearly:

- Files used in the system.
- Libraries and other artifacts relevant to the application.
- Relationships among the artifacts.

Now after identifying the artifacts the following points needs to be followed:

- Use a meaningful name to identify the component for which the diagram is to be drawn.
- Prepare a mental layout before producing using tools.
- Use notes for clarifying important points.
The following is a component diagram for order management system. Here the artifacts are files. So the diagram shows the files in the application and their relationships. In actual the component diagram also contains dlls, libraries, folders etc.

In the following diagram four files are identified and their relationships are produced. Component diagram cannot be matched directly with other UML diagrams discussed so far. Because it is drawn for completely different purpose.

So the following component diagram has been drawn considering all the points mentioned above:
Where to use Component Diagrams?

We have already described that component diagrams are used to visualize the static implementation view of a system. Component diagrams are special type of UML diagrams used for different purposes. These diagrams show the physical components of a system. To clarify it, we can say that component diagrams describe the organization of the components in a system. Organization can be further described as the location of the components in a system.

These components are organized in a special way to meet the system requirements. As we have already discussed those components are libraries, files, executables etc. Now before implementing the application these components are to be organized. This component organization is also designed separately as a part of project execution. Component diagrams are very important from implementation perspective. So the implementation team of an application should have a proper knowledge of the component details.

Now the usage of component diagrams can be described as:

- Model the components of a system.
- Model database schema.
- Model executables of an application.
- Model system's source code.
8. Explain Deployment Specification

A deployment specification is an artifact that specifies a set of deployment properties that determine execution parameters of a component artifact that is deployed on a node. A deployment specification can be aimed at a specific type of container for a components. A deployment specification is a general mechanism to parameterize a deployment relationship, as is common in various hardware and software technologies.

The deployment specification element is expected to be extended in specific component profiles. Non-normative examples of the standard stereotypes that a profile might add to deployment specification are, for example, «concurrencyMode» with tagged values {thread, process, none}, or «transactionMode» with tagged values {transaction, nestedTransaction, none}. A deployment specification at specification level is graphically displayed as a classifier rectangle with optional deployment properties in a compartment.

The ejb-jar.xml deployment specification An artifact that reifies or implements deployment specification properties is a deployment descriptor. A deployment specification at instance level is graphically displayed as a classifier rectangle with the name underlined and with deployment properties having specific values in a compartment.

Prepared By: Mrs.R.BARONA, AP/IT
Deployment Specification Dependency

A deployment specification could be displayed as a classifier rectangle attached to a component artifact using a regular dependency arrow pointing to deployed artifact.

Deployment Specification Association

Deployment Specification could be associated with the deployment of a component artifact on a node. In this case deployment specification could be displayed as a classifier rectangle attached to the deployment. Note, UML 2.2 specification shows this association as a dashed line (while association is normally displayed as solid line.)
Deployment

A deployment is dependency relationship which describes allocation of an artifact or artifact instance to a deployment target. Accordingly, deployed artifact is an artifact or artifact instance that has been deployed to a deployment target. A component deployment is the deployment of one or more artifacts or artifact instances to a deployment target, optionally parameterized by a deployment specification.

Examples are executables and configuration files. The deployment relationship between a deployed artifact and a deployment target can be defined at the “type” level and at the “instance level.” For example, a “type level” deployment relationship can be defined between an “application server” Node and an “order entry request handler” executable Artifact.

At the “instance level” 3 specific instances “appserver1” ... “app-server3” may be the deployment target for six “request handler*” instances. For modeling complex deployment target models consisting of nodes with a composite structure defined through „parts,” a Property (that functions as a part) may also be the target of a deployment. Deployment diagram shows deployed artifacts contained within a deployment target symbol.
The portfolio.ear artifact deployed on application server. Deployment could also be shown using textual list of deployed artifacts within a deployment target symbol.

**Deployment Diagram Example**

- Captures the distinct number of computers involved
- Shows the communication modes employed
- Component diagrams can be embedded into deployment diagrams effectively

Prepared By: Mrs. R. BARONA, AP/IT