**UNIT III**

Class & Object Diagrams: Terms, concepts, modeling techniques for Class & Object Diagrams. Advanced Structural Modeling: Advanced classes, advanced relationships, Interfaces, Types and Roles, Packages. Case Study: AI: Cryptanalysis.

**Class diagrams**

Class diagrams are most commonly found diagrams while modeling software systems. A class diagram consists of classes, collaborations and interfaces. A class diagram is used to model the static design view of the system.

Imagine you're building a house. You need to plan out all the rooms, their sizes, and how they connect. A class diagram is like a blueprint for a software system. It helps you visualize the different parts of your software (called "classes") and how they interact with each other.

**Basic Terms:**

* **Class**: A blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of that class will have.
* **Object**: An instance of a class. Think of it as a specific house built from the blueprint.
* **Attribute**: A property of a class. For example, a "Car" class might have attributes like "color," "model," and "speed."
* **Method**: An action that an object can perform. For example, a "Car" class might have methods like "start," "accelerate," and "brake."

**Common properties**

The class diagram shares the common properties with the rest of the diagrams, a name and graphical elements that are a projection into the model. What distinguish the class diagram from the others are the contents of the diagram

Class diagram contains the following:

* Classes
* Interfaces
* Collaborations
* Generalization, dependencies and associations

Like other diagrams, class diagrams may also contain notes and constraints. Class diagrams may also contain packages or subsystems which are used to organize the elements into groups. Generally, the class diagrams are used for:

* Modeling the vocabulary of the system
* Modeling the collaborations
* Modeling the logical database schema

**Common Modeling Techniques**

**Modelling simple collaborations**

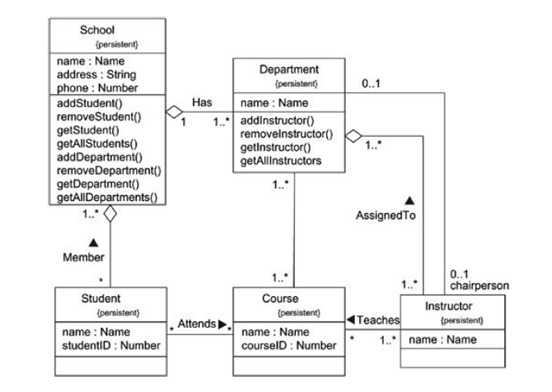
To model simple collaborations

1. Identify the function or behavior of the part of a system you would like to model.
2. For each function or mechanism identify the classes, interfaces, collaborations and relationships between them.
3. Use scenarios (sequence of activities) to walk through these things. You may find new things or find that existing things are semantically wrong.
4. Populate the things found in the above steps. For example, take a class and fill its responsibilities. Now, convert these responsibilities into attributes and operations.

**Modeling a Logical Database Schema**

**To model a schema**

1. Identify the classes whose state must be saved over the lifetime of the application.
2. Create a class diagram and mark these classes as persistent by using tagged values.
3. Provide the structural details for these classes like the attributes, associations with other classes and the multiplicity.
4. Minimize the common patterns which complicate the physical database design like cyclic associations, one-to-one associations and n-ary associations.
5. Provide the behavior for these classes by listing out the operations that are important for data access and integrity.
6. Wherever possible, use tools to convert the logical design to physical design**.**

****

**Class Diagram Database Schema**

**Forward and Reverse Engineering**

**To forward engineer a class diagram,**

1. Identify the rules for mapping the models to the implementation language of your choice.
2. Depending on the semantics of the language, you may want to restrict the information in your UML models. For example, UML supports multiple inheritance. But some programming languages might not allow this.
3. Use tagged values to specify the target language.
4. Use tools to convert your models to code**.**

**To reverse engineer the code**

1. Identify the rules for mapping the implementation language to a model.
2. Using a tool, navigate to the code that you want to reverse engineer. Use the tool to generate a new model.

In a class diagram, you use these terms and concepts to define the structure of your system. You draw classes with their attributes and methods, show relationships like inheritance and association between them, and use different notations to explain visibility, multiplicity, and dependency. This helps in understanding how different parts of the system interact and how data flows through it.

**Object Diagrams**

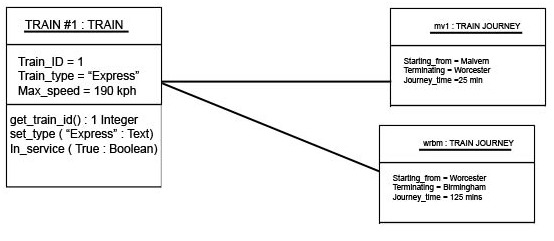
An Object Diagram is a type of diagram used in the Unified Modeling Language (UML) to show how objects (specific instances of classes) interact with each other in a system at a particular moment in time. It's like taking a snapshot of the objects and their relationships in a program**.**

Before learning about object diagrams in UML, have a look at class diagrams in UML.

An object diagram models the instances of classes in a class diagram. An object diagram contains objects and links connecting the related objects.

Object diagrams allow us to model the static design view or static process view. It involves modeling a snapshot of the system which contains objects (instances), their state and links between objects.

An object diagram represents the static part of an interaction containing the objects that communicate and the links between them without any messages.



**Terms and Concepts**

* + **Object Diagram:** A graphical representation of a set of objects and their relationships at a particular point in time**.**
  + **Object**: An instance of a class, representing a real-world entity or concept.
  + **Class:** A blueprint or template that defines the properties and behavior of an object**.**
  + **Attribute:** A property or characteristic of an object, such as its name, age, or address.
  + **Link:** A connection between two objects, representing a relationship between them**.**
  + **Role:** The name or label given to an object in a specific context or relationship

**Common Properties**

An object diagram shares the same common properties like the rest of the diagrams, a name which identifies the diagram and graphical contents which are a projection into the model. What distinguish an object diagram from the rest of the diagrams are its contents.

**Contents**

Object diagrams contain:

* Objects
* Links

Like all other diagrams, object diagrams may contain notes and constraints.

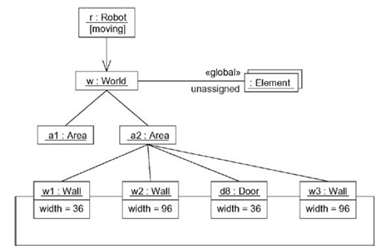
Object diagrams may also contain packages and subsystems which are used to group the related elements in the model together. Sometimes we can also include classes in the object diagrams. Generally, object diagrams are used to model the object structures.

**Common Modeling Techniques**

**Modeling Object Structures**

**To model an object structure,**

1. First, identify the function or behavior or part of a system you want to model as collection of classes, interfaces and other things.
2. For each function or mechanism identify the classes and interfaces that collaborate and also identify the relationships between them.
3. Consider a scenario (context) that walks through this mechanism and freeze at a moment in time and identify the participating objects.
4. Represent the state of objects by listing out the attributes and their values.
5. Represent the links between objects which are instances of associations.

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/modeling-object-structures.gif)

**Forward and Reverse Engineering**

Forward engineering a object diagram is theoretically possible but practically of limited value as the objects are created and destroyed dynamically at runtime, we cannot represent them statically.

**To reverse engineer a object diagram**,

1. Choose the target (context) you want to reverse engineer.
2. Use a tool to stop execution at a certain moment in time.
3. Identify the objects that collaborate with each other and represent them in an object diagram.
4. To understand their semantics, expose these object’s states.
5. Also identify the links between the objects to understand their semantics.

**Advanced Structural Modeling**

Advanced Structural Modeling, we use various diagrams and techniques to represent how different parts of a system are structured and related:

* + **Class Diagrams** show the structure of individual objects and their relationships.
  + **Component Diagrams** show how large parts of a system are made up of smaller, reusable components.
  + **Composite Structure Diagrams** represent internal structures and collaborations of components or classes.
  + **Dependency, Aggregation, Composition, and Interfaces** define how elements depend on each other, and how strong their relationships are.
  + **Deployment Diagrams** show how software components are deployed on hardware.

This type of modeling is essential when building complex systems, as it helps to visualize how everything fits together and works as a whole

**Advanced classes**

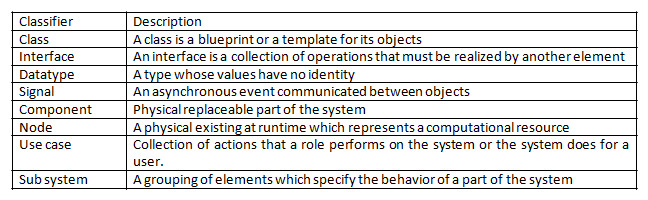
Advanced classes in Unified Modeling Language (UML) are classes with advanced features like classifiers, visibility, and scope. These features help to define the structure and behavior of classes and their relationships.

* **Classifiers**
* **Visibility**
* **Scope**

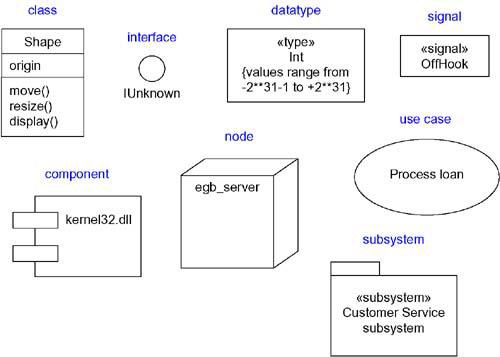
**classifiers**

* A classifier is a modeling element that defines the attributes and operations of a class.
* Classifiers include classes, interfaces, datatypes, signals, components, nodes, use cases, and subsystems.
* Every instance of a classifier shares the same features.

In general, all the modeling elements that can have instances are called as classifiers. For example, classes have instances known as objects. Some elements like packages and a generalization does not have instances.  Classifiers in UML are as follows:

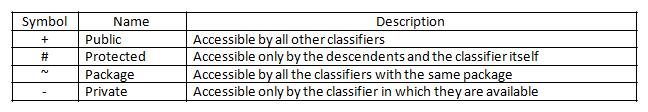


The graphical representation of the classifiers is as follows:



**Visibility**

One of the important features that can be applied to the classifier’s attributes and operations is the visibility. The visibility feature specifies whether a classifier can be used or accessed by the other classifiers. The four access specifiers in UML are as shown below:

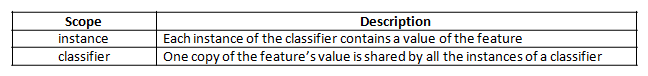
[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/3-visibility.gif)

**Example:**

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/4-visibility-example.gif)

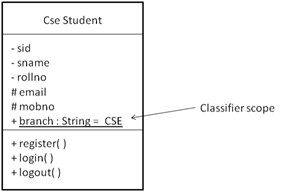
**Scope**

Another feature that can be applied to the classifier’s attributes and operations is the scope. The scope of an attribute or an operation denotes whether they have their existence in all the instances of the classifier or only one copy is available and is shared across all the instances of the classifier. The scope specifiers in UML are:

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/5-scope.gif)

A classifier scoped feature is graphically represented by underlining the feature within the classifier. All other normal features are treated as instanced scope.

**Example:**

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/6-scope-example1.gif)

**Abstract, Root, Leaf and Polymorphic elements**

While developing class diagrams, it is common to use generalization relationship to model generalization-specialization relationships between a group of classes.

The classes at the top of the hierarchy are more generalized or abstract and the classes lower in the hierarchy are more specialized. To represent abstract classes for which we cannot create instances, the class name is written in italics in UML. To represent abstract methods, we write the operation signature in italics.

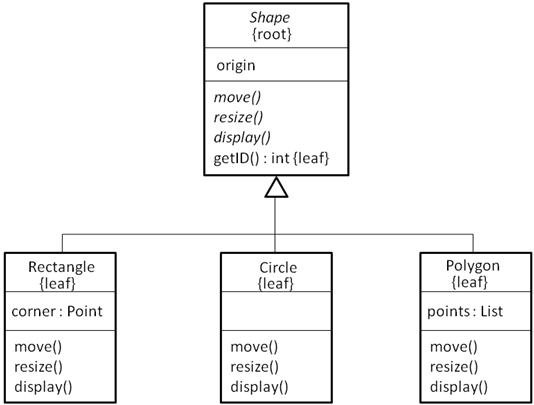
We can also represent leaf and root classes in UML. A class which has no child classes is known as a leaf class. Such a class can be represented by writing leaf as a property under the class name.

Similarly, a class with no parents is known as a root class and such a class can be represented by writing root as a property as a property under the class name.

In class hierarchies it is common for the classes to have methods with same signatures. This feature of declaring methods with same signatures is known as polymorphism. Such methods are represented in UML by writing their signatures in italics.

We can also specify that a method cannot be overrided by marking the method with the leaf property written beside the method signature.

**Example:**

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/7-abstract-root-leaf.gif)

Multiplicity

Multiplicity defines the number of objects that can participate in a relationship between classes. It specifies how many instances of one class are associated with a single instance of another class. It helps represent constraints on associations between classes.

**Key Notations:**

**1 → Exactly one object.**

**0..1 → Zero or one object.**

**0..\* or \* → Zero or more objects (many).**

**1..\* → One or more objects**.

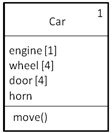
1..\* → One or more objects.

.

**Example:** Imagine a Library and a Book:

* A Library can have many Books (Multiplicity: 0..\*).
* A Book belongs to exactly one Library (Multiplicity: 1).

**Example :** Car Multiplicity



**UML Representation:**

**Library 1 --------- 0..\* Book**

**This means:**

* One Library can have multiple Books.
* Each Book belongs to exactly one Library.

**Attributes**

Attributes represent the data or properties of a class. They define what information each object of the class will hold. They help describe the characteristics of a class.

**Key Features:**

* Attributes have a name, a type, and optionally a default value.
* Visibility modifiers (+, -, #) specify access levels:

**+ (Public): Accessible by all.**

**- (Private): Accessible only within the class.**

**# (Protected): Accessible by the class and its subclasses**.

**Example:** For a Car class:

* Attributes could include:
  + +brand: String
  + +speed: int
  + -fuelType: String

**Class: Car**

**+ brand: String**

**+ speed: int**

**-fuelType:String**

**UML Representation:**

**This means:**

* brand and speed are public and can be accessed from anywhere.
* fuelType is private and can only be accessed within the Car class.

**Operations**

Operations define the behavior or functionality of a class. They represent what the class can do.They specify actions or methods associated with the class.

**Key Features:**

* Operations also use visibility modifiers (+, -, #).
* They may have parameters and return types.

**Example:** For the Car class, operations could include:

* +drive(distance: int): void → A method to drive the car.
* +refuel(amount: float): void → A method to refuel the car.
* -calculateEfficiency(): float → A private method to calculate fuel efficiency.

**UML Representation:**

**Class: Car**

**+ drive(distance: int): void**

**+ refuel(amount: float): void**

**- calculateEfficiency(): float**

**This means:**

* drive and refuel are public and can be called from outside the class.
* calculateEfficiency is private and used internally within the Car class.

1. **Multiplicity** describes how many objects of one class relate to another.
2. **Attributes**  define the data stored in a class.
3. **Operations** specify what actions a class can perform.

By using these elements in UML diagrams, you can design systems that are clear, scalable, and easy to understand

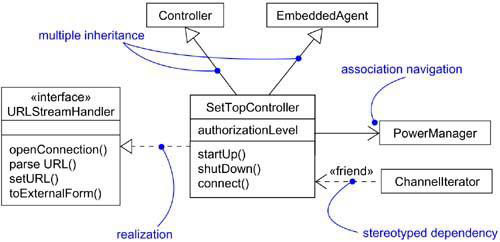
**Common Modeling Techniques**

**To model the semantics of a class:**

1. Specify the responsibilities of the class in a note and attach it to the class with a dependency relationship.
2. Specify the semantics of a class as a whole in a note stereotyped with “semantics”.
3. Specify the body of a method using structured text or a programming language in a note and attach it to the class or operation using a dependency relationship.
4. Specify the pre-conditions and post-conditions for operations using structured text in a note.
5. Specify a state machine for a class which represents different states the object undergoes.
6. Specify a collaboration that represents a class.

**Advanced relationships**

A relationship is a connection among things(things can be any of, structural, behavioural, annotational or  grouping. This module contains mainly structural things). In object-oriented modeling, the four most important relationships are dependencies, generalizations, associations, and realizations. Diagram indicating advanced relationship is shown in

**[](https://praveenthomasln.wordpress.com/wp-content/uploads/2012/03/advanced-relationships.png)**

The things in diagrams are connected with one another through relationships. So, relationships are the connections between things. In UML, the four important relationships are dependency, generalization, association and realization. Each type of relationship has its own graphical representation.

1. Dependency

2. Generalization

3. Association

4. Realization

**1. Dependency**

The dependency relationship is also known as using relationship i.e., if the specification of one thing (for example Graphics class code) changes then it will automatically affect the behavior of another thing (for example Hello class using the drawstring method of Graphics class) that uses it. A dependency relationship is graphically represented as a dashed arrow.

**Target**

**Target**

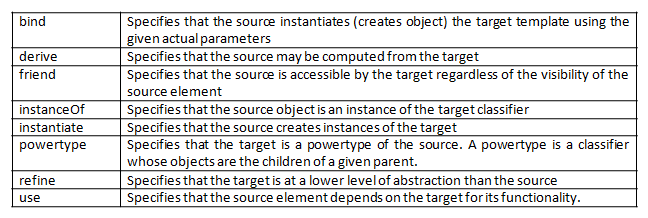
**Sorce**

--------------------->

In general, a plain dependency relationship is reasonable for representing the using relationship between two things. But, if the user needs to specify extra information like the nature of the dependency relationship, he/she can use the predefined stereotypes that can be applied to dependency relationship.

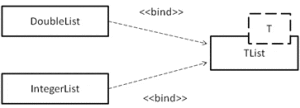
There are about 17 such stereotypes and are organized into 6 groups based on which things are participating in the dependency relationship. The stereotypes are as follows:

First group consists of eight stereotypes that apply to dependency relationship between classes. They are as follows:

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/2-dependency-stereotypes.gif)

**Examples:**

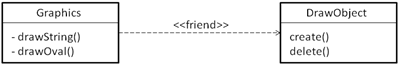
**<<bind>>**

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/3-bind-example.gif)

**<<derive>>**

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/4-derive-example.gif)

**<<friend>>**

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/5-friend-example.gif)

**<<instanceOf>>**

[instanceof stereotype](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/6-instance-of-example.gif)

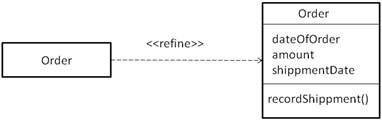
**<<instantiate>>**

[instantiate stereotype](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/7-instantiate-example.gif)

**<<powertype>>**

[powertype stereotype](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/8-powertype-example.gif)

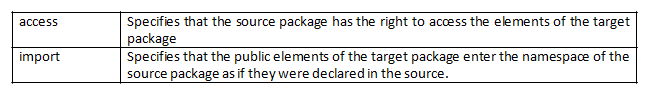
**<<refine>>**

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/9-refine-example.gif)

**<<use>>**

[use stereotype](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/10-use-example.gif)

There are two stereotypes that apply to dependencies between packages. They are:

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/11-dependency-package-stereotypes.gif)

**Examples:**

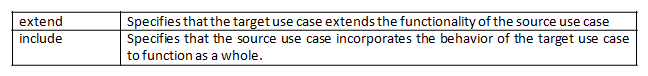
**<<access>>**

[access example](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/12-access-example.gif)

**<<import>>**

[import example](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/13-import-example.gif)

Two stereotypes apply to dependency relationship among use case. They are:

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/14-dependency-usecase-stereotypes.gif)

Examples:

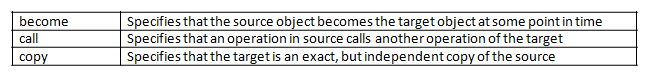
**<<extend>>**

[extend use case example](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/15-extend-example.gif)

**<<include>>**

[include use case example](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/16-include-example.gif)

Three stereotypes apply to interactions among objects. They are:

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/17-dependency-object-stereotypes.gif)

Examples:

**<<become>>**

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/18-become-example.gif)

**<<call>>**

[call dependency example](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/19-call-example.gif)

**<<copy>>**

[copy dependency example](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/20-copy-example.gif)

One stereotype applies to dependencies in the context of state machines:

[state dependency stereotype](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/21-dependency-state-stereotypes.gif)

One stereotype applies to dependencies in the context of subsystem:

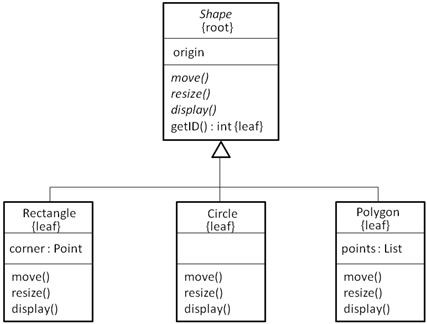
[subsystem dependency stereotype](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/22-dependency-subsystem-stereotype.gif)

**Example:**

[trace dependency example](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/23-trace-example.gif)

1. **Generalization**

A generalization relationship represents generalization-specialization relationship between classes. The class with the general structure and behavior is known as the parent or superclass and the class with specific structure and behavior is known as the child or subclass. Consider the below class hierarchy:

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/1-generalization.gif)

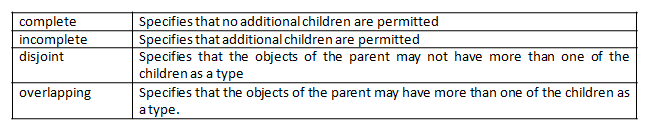
Shape class is the parent or super class and the remaining three classes namely Rectangle, Circle and Polygon are the child or subclasses of the Shape class.

A subclass in the generalization relationship automatically inherits the state and behavior of the superclass. The generalization relationship is also known as the “is-a” relationship. If a class has only one parent, such inheritance is known as single inheritance and if a class has one or more parents, such inheritance is known as multiple inheritance.

To represent extra semantics in a generalization relationship, UML provides one stereotype and four constraints. The stereotype on generalization relationship is:

[generalization stereotype](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/2-generalization-stereotype.gif)

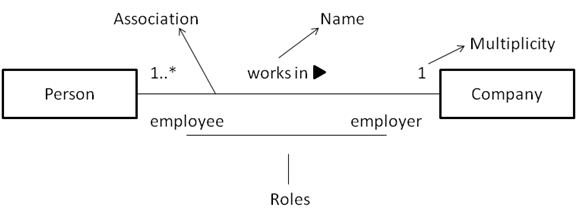
There are four standard constraints that apply to the generalization relationship:

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/3-generalization-constraints.gif)

**Association**

Association is a structural relationship which denotes a connection between two or more things. The association relationship can represent either physical or logical connections between things. The graphical representation of the association relationship is a solid line.

The four basic adornments for an association relationship are: name, role at each end of the association, multiplicity at each end of the association and aggregation. Over these basic features, there are other advanced features like: navigation, visibility, qualification, composition and association classes.

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/4-association.gif)

**Navigation**

Given an association between two things, we can navigate from one thing to another and vice versa. By default the navigation of an association is bidirectional. In some cases we may need to navigate in only direction. For example, given an order, we can get the details of the products in the order. But, if we consider a product, there might be no need to know about the order to which it belongs.

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/5-association-navigation.gif)

**Visibility**

Given an association, we can navigate from one object to another. However, in some situations we may want to limit the visibility (access) of an object to the objects outside the association relationship.

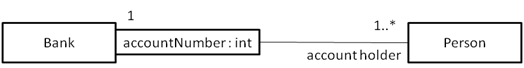
To limit the visibility of an object, we can use the visibility specifiers: public (+), private (-) and protected (#). For example, a usergroup object can access the user object and an user object can access the password. If, private visibility is specified for the password object, the usergroup object cannot access the password of the user object.

[association visibility example](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/6-association-visibility.gif)

**Qualification**

In binary associations, where two classes are connected together, if one object of a class has to identify a set of instances of another class, we can use qualifiers. A qualifier is a set of attributes of an association which is used to identify a set of instances of a class.

Each qualifier is represented with a name and type in a rectangle box at the qualified end of the association relationship. For example, a bank object is able to recognize the person (account holder) based on the account number. So, account number is the qualifier which is used to identify an account holder.

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/7-association-qualification.gif)

**Composition**

The composition is a flavor of association relationship. Composition as well as aggregation relationships represent whole-part relationships, in which one thing is a part of the other thing.

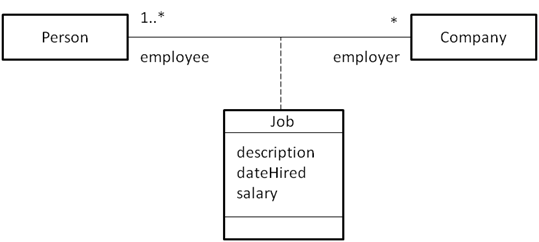
There is a simple difference between the association and composition relationships. In composition, the lifetime of the part is dependent of the lifetime of the whole thing. Where as in aggregation, the lifetime of the part is independent of the whole thing.

Composition is graphically represented by adorning the association relationship with a filled diamond head near the whole end.

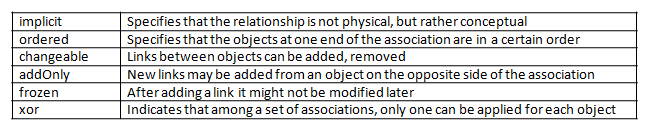
[association composition example](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/8-association-composition.gif)

**Association Class**

Sometimes in an association relationship, the association might have attributes or properties like a class does. In such cases, it is modeled as an association class. An association class in UML is represented with a class icon attached to the association with a dashed line.

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/9-association-class.gif)

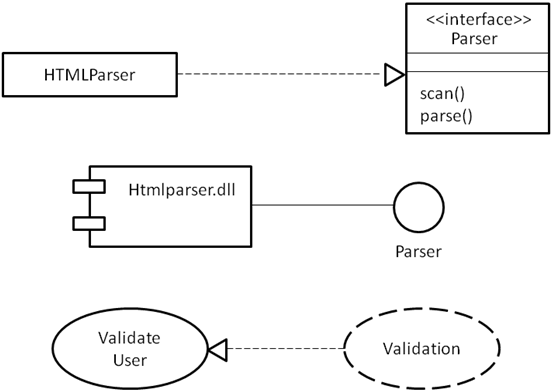
Apart from the above advanced features, UML provides six constraints that can be applied to an association. They are:

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/10-association-constraints.gif)

**Realization**

Realization is a semantic relationship between classifiers, where one classifier provides the specification which is implemented by the other classifier. Realization can exist between an interface and class, interface and a component and between a use case and collaboration.

Realization is graphically represented as dashed line with a hollow arrow head pointing towards the classifier which provides the specification.

[](http://www.startertutorials.com/uml/wp-content/uploads/2013/08/11-relaization-examples.gif)

**Common Modeling Techniques**

To model webs of relationships,

1. Apply use cases and scenarios to find the relationships between the abstractions in the system.
2. Start by modeling the structural relationships (associations) between the things. These specify the structure of the system.
3. Then, model the generalization-specialization relationships.
4. Finally, after modeling the remaining relationships go for dependency relationships.
5. After representing all the relationships, transform their basic representation by applying the advanced features to your intent.

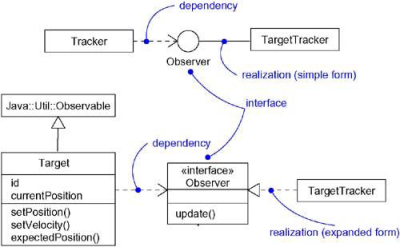
**Interfaces, Types, and Roles**

In UML (Unified Modeling Language), **Interfaces, Types, and Roles** are key concepts used to model different aspects of a system’s structure and behaviour.

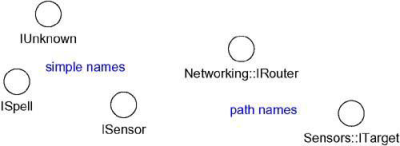
| **Concept** | **Definition** | **Example** |
| --- | --- | --- |
| **Interface** | A contract defining methods for a class | Payment interface implemented by Credit Card Payment |
| **Type** | A classification for objects based on behavior or attributes | Vehicle type for Car and Bike |
| **Role** | A specific behavior or purpose an object plays in context | Person as a Teacher or a Parent |

using **Interfaces, Types, and Roles**, UML helps create flexible and reusable designs, allowing you to model both the structure and dynamic behavior of complex systems effectively.

**Interface**  
 An interface is a collection of operations that are used to specify a service of a class or a component. **Graphically**, an interface is rendered(represented) as a circle; in its ex panded form, an interface may be rendered as a stereotyped class(a class with sterotype interface) as shown in Figure:1. An interface name must be unique within its enclosing package. **Two naming mechanism**; a simple name(only name of the interface), a path name is the interface name prefixed by the name of the package in which that interface lives represented in Figure: 2. To distinguish an interface from a class, prepend an ‘I’ to every interface name. Operations in an interface may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.interface dont have attributes. interfaces span model boundaries and it does’nt have direct instances.

[](https://praveenthomasln.wordpress.com/wp-content/uploads/2012/03/interface-representation1.png)

**Figure:1**

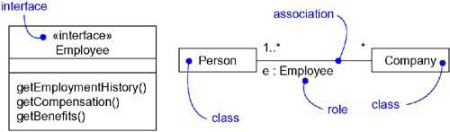
[](https://praveenthomasln.wordpress.com/wp-content/uploads/2012/03/naming-interfaces.png)

**Figure:2**

**Interface relationships**  
 An interface may participate in generalization, association, dependency and realization relationships.  
**Note:** Interfaces may also be used to specify a contract for a use case or subsystem.

**Type**  
 A type is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object of that type. To distinguish a type from an interface or a class, prepend a ‘T’ to every type. Stereotype type is used to formally model the semantics of an abstraction and its conformance to a specific interface.

**Role**  
 A role names(indicates) a behavior of an entity participating in a particular context. Or, a role is the face that an abstraction presents to the world. For example, consider an instance of the class Person. Depending on the context, that Person instance may play the role of Mother, Comforter, PayerOfBills, Employee, Customer, Manager, Pilot, Singer, and so on. When an object plays a particular role, it presents a face to the world, and clients that interact with it expect a certain behavior depending on the role that it plays at the time. For example, an instance of Person in the role of Manager would present a different set of properties than if the instance were playing the role of Mother. Figure:3 indicates a role employee played by person and is represented statically there.

[](https://praveenthomasln.wordpress.com/wp-content/uploads/2012/03/roles.png)

**Figure:3 Roles**

In Figure:3 the Person presents the role of Employee to the Company, and in that context, only the properties specified by Employee are visible and relevant to the Company.

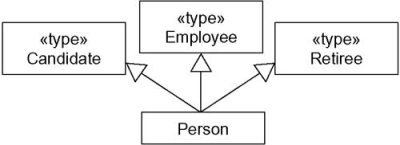
**Static and Dynamic modeling in UML**  
 A class diagram that indicates a particular role is useful for modeling the static binding of an abstraction to its interface. To model the dynamic binding of an abstraction to its interface by using the become stereotype in an interaction diagram, showing an object changing from one role to another.

**To Model A Dynamic Type**

* Specify the different possible types of that object by rendering each type as a class stereotyped as type (if the abstraction requires structure and behavior) or as interface (if the abstraction requires only behavior).
* Model all the roles the class of the object may take on at any point in time. It can be done in two ways:

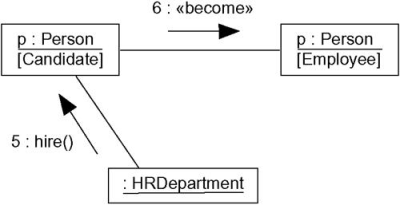
1. First, in a class diagram, explicitly type each role that the class plays in its association with other classes. Doing this specifies the face instances of that class put on in the context of the associated object.
2. Second, also in a class diagram, specify the class-to-type relationships using generalization

**They are represented in the following figures:**

[](https://praveenthomasln.wordpress.com/wp-content/uploads/2012/03/static-modelling.png)

**Figure:4 Static modeling**

Figure:4 shows statically that instances of the Person class may be any of the three types  namely, Candidate, Employee, or Retiree.

[](https://praveenthomasln.wordpress.com/wp-content/uploads/2012/03/dynamic-modeling.png)

**Figure:5 Dynamic-modeling**

Figure:5 shows the dynamic nature of a person’s type. In this fragment of an interaction diagram, p (the Person object) changes its role from Candidate to Employee.

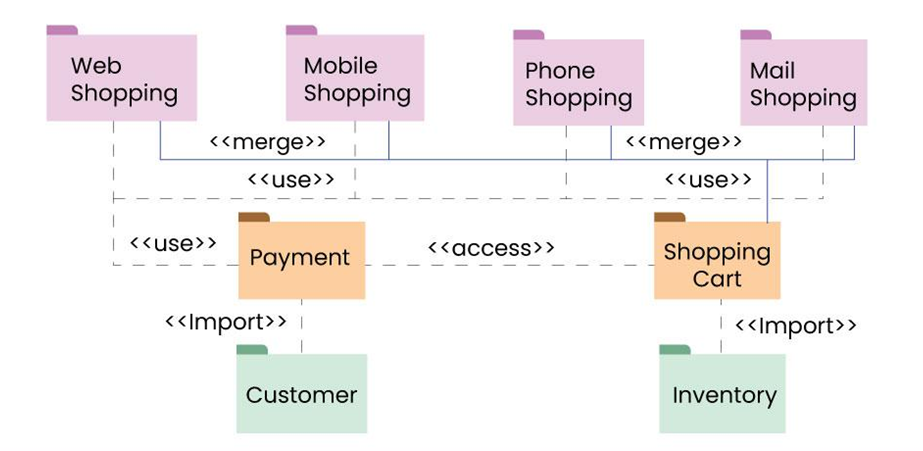
**Packages in UML**

* In UML (Unified Modeling Language), a **package** is like a folder or a container.
* It is used to **group related elements** together, such as classes, interfaces, or other packages.
* Packages help in organizing a large and complex system into smaller, manageable parts.

A package is used as a container to organize the elements present in the system into a more manageable unit. It is very useful to represent the system's architecture and design as a cohesive unit and a concise manner

**Basic Elements of Package Diagrams**

The following are the basic elements of a Package Diagrams:



In a package diagram, several key elements help organize and clarify the relationships within a system:

* **Package**: This is the fundamental unit of a package diagram, serving as a container for various elements like classes and interfaces. It’s depicted as a folder-like icon with a name label, making it easy to identify.
* **NameSpace**: This denotes the name of the package and usually appears at the top of the package symbol. It helps uniquely identify the package within the diagram.
* **Package Merge**: This relationship illustrates how one package can be merged with another. It’s represented by a direct arrow between the two packages, indicating that their contents can combine.
* **Package Import**: This relationship shows that one package can access the contents of another package, depicted with a dashed arrow.
* **Dependency**: Dependencies indicate that changes in one package may affect another. This relationship signifies that one element or package relies on another, highlighting how interconnected they are.
* **Element**: An element is a single unit within a package, which can be a class, interface, or subsystem. Elements reside inside packages and are connected to the main package. For instance, a class may contain various functions and variables, all of which are considered elements tied to that class.
* **Constraint**: This represents a condition or requirement associated with a package, typically shown in curly braces. Constraints help define the rules or limitations for how the package operates.

**Package Structure and Notation**

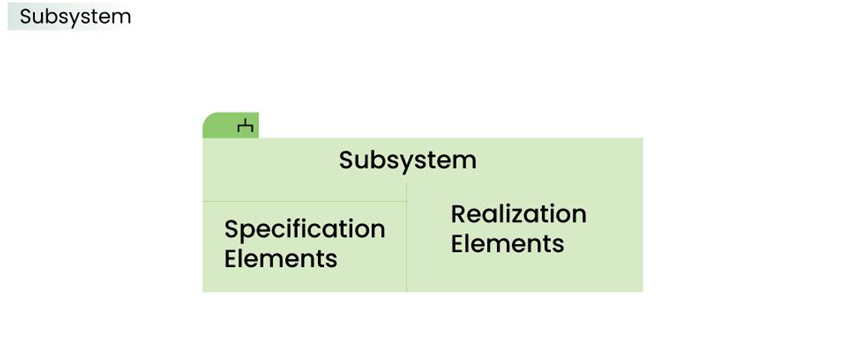
Below are the package structures and their notations:

**1. Package**



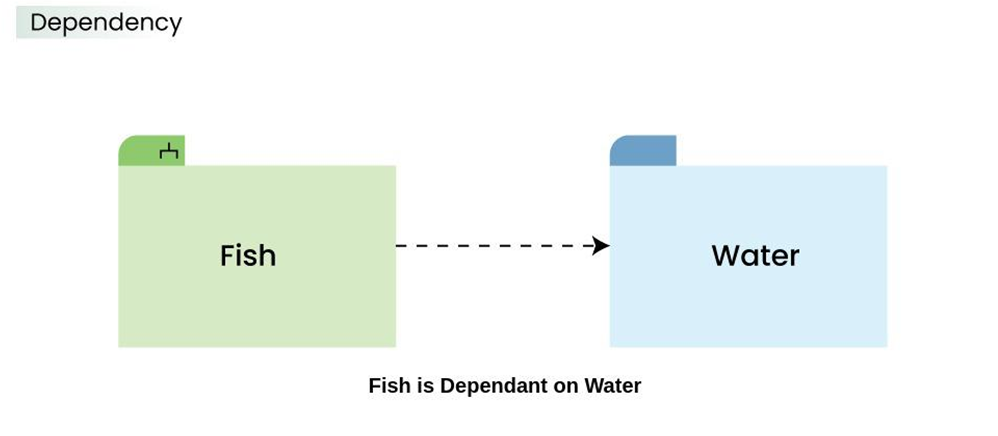
The above is the notation of a simple Package.

**2. Subsystem**



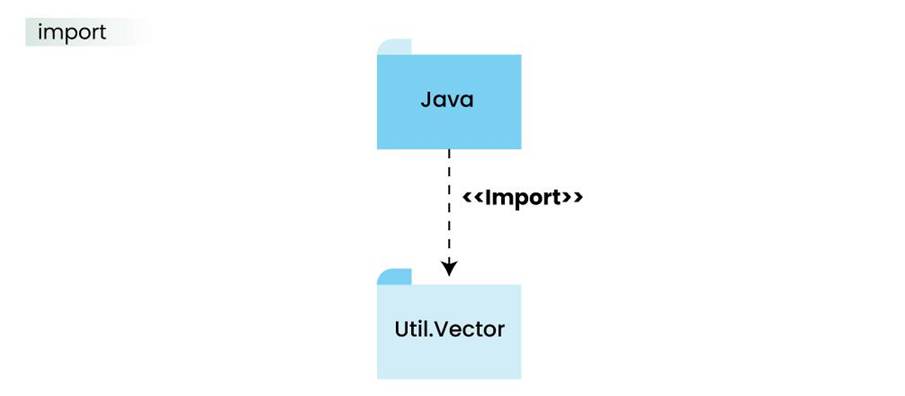
The above notation is used to represent subsystem.

**3. Dependency**



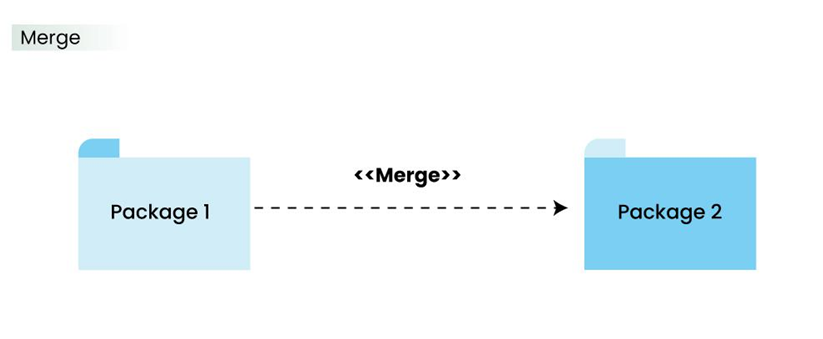
The above dashed arrow sign is used to show the dependency among two elements or two packages.

**4. Import**



The above notation is of the **Import**, here it also uses a dashed arrow line but the difference is, the word **<<import>>** is being written to represent the below package or function or element has been imported from the above package.

**5. Merge**



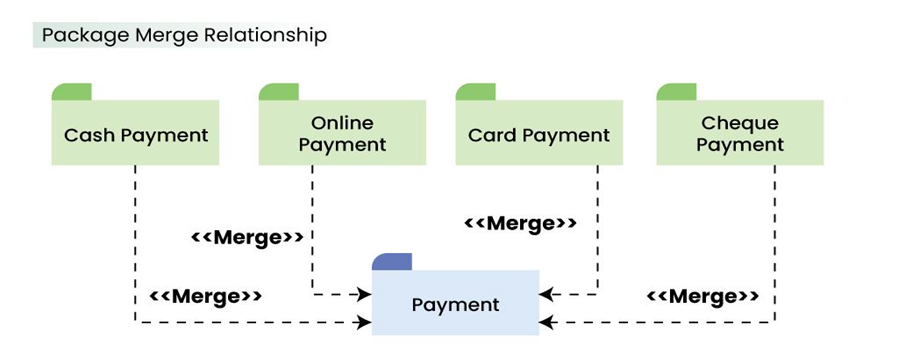
This notation above denotes that the Package 1 can be merged with Package 2

**Package Relationships**

**1. Package Merge Relationship**

This relationship is used to represent that the contents of a package can be merged with the contents of another package. This implies that the source and the target package has some elements common in them, so that they can be merged together.

**Example:**

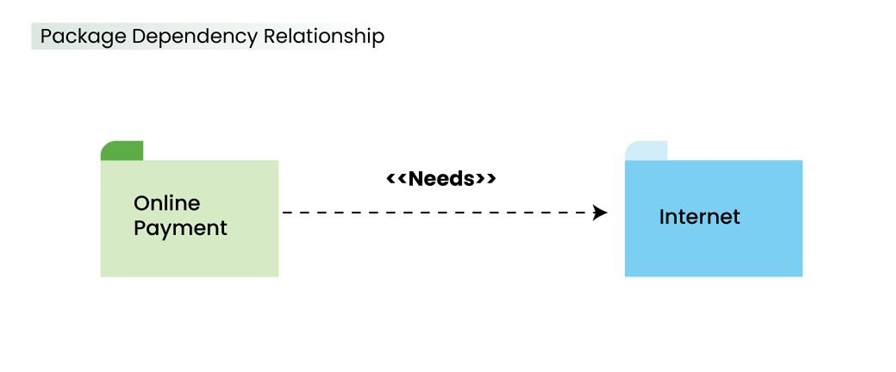


The above diagram depicts that the packages are of different type of payments, but all of them are a some kind of payment mechanism, so they can all be merged to be called as payment.

**2. Package Dependency Relationship**

A package can be dependant on other different packages, signifying that the source package is somehow dependent on the target package.

**Example:**

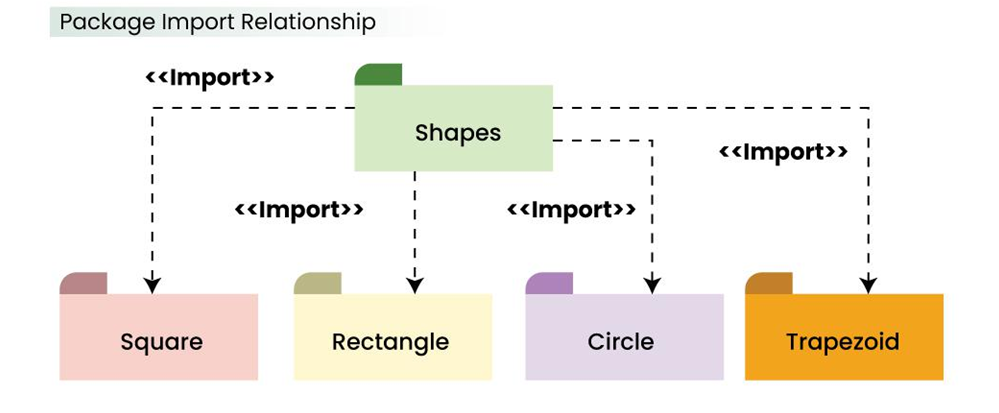


The above diagram depicts that the online payment package is dependent on the Internet package and uses "need" dependency.

1. **Package Import Relationship**

This relationship is used to represent that a package is importing another package to use. It signifies that the importing package can access the public contents of the imported package.

**Example:**

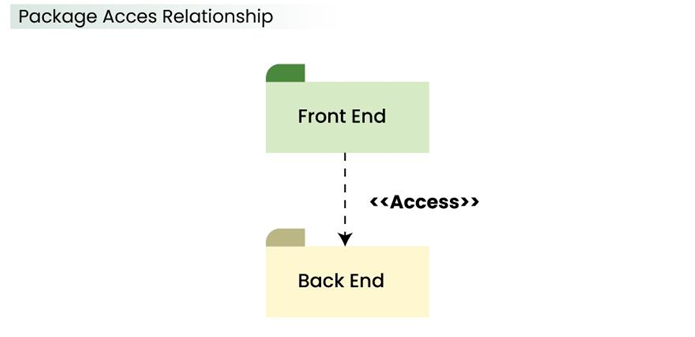


The above package diagram shows the import relationship between the main package Shapes and it's various other sub packages Square, Rectangle etc. They all are importing the real Shapes package so that the public contents of the Shapes package can be used by them.

**4. Package Access Relationship**

This type of relationship signifies that there is a access relationship between two or more packages, meaning that one package can access the contents of another package without importing it.

**Example:**



The above diagram depicts the Access relationship between the Front End and Back End services. It is much needed that a front end service can easily access the important Back End services to carry out any operation.

**Use Of Packages**

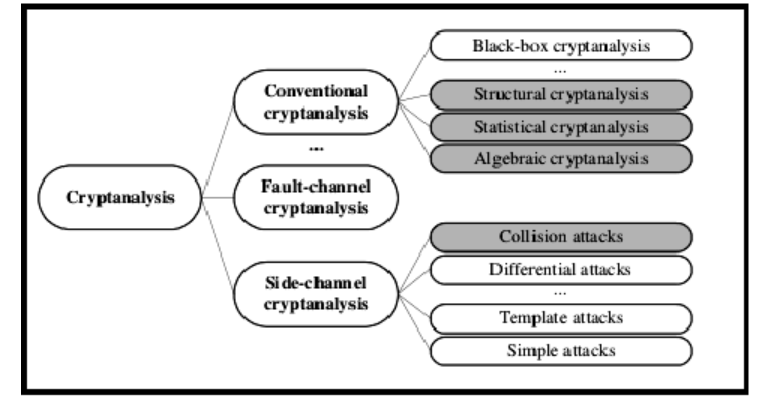
1. **Organization**: Helps keep the system neat and easy to understand.
2. **Reusability**: Packages can be reused in other systems or parts of the project.
3. **Modularity**: Allows dividing the system into independent modules.
4. **Encapsulation**: Hides the internal details of a package while exposing only necessary parts to the outside.

* Think of packages as folders in your computer.
* They group related classes and make large systems easier to manage.
* Use relationships to show how different packages work together.
* Always aim for simplicity and clarity when creating packages in UML.

**Case Study: AI: Cryptanalysis**

This case study explores the use of AI In Cryptanalysis.Cryptography has always been an important part of communication security,and with the advent of AI it has become more efficient than ever before

Cryptanalysis is the study of analyzing information systems in order to study the hidden aspects of the systems. Cryptanalysis is used to breach cryptographic security systems and gain access to the contents of encrypted messages,even if the cryptographic key is unknown



**What is AI**

**A**rtificial intelligence(AI) is the simulation of human intelligence process by computer systems.These processes include learning,reasoning,and self-correction.AI has become an essential tool in many fields,including Cryptanalysis

AI can be used in Cryptanalysis to quickely analyze massive amounts of data and identify patterns that would be impossible for humans to detect.AI can also be used to improve the efficiency of encryption systems and make them more secure.

**Role of AI in Cryptanalysis**

Traditionally, breaking a code required a lot of time, effort, and mathematical expertise. Now, **Artificial Intelligence (AI)** can assist in cryptanalysis by automating and speeding up the process.

**How AI Helps in Cryptanalysis:**

1. **Pattern Recognition**:
   * AI can identify patterns in encrypted messages that humans may miss.
   * Example: Spotting repeated sequences or statistical anomalies.
2. **Machine Learning (ML)**:

* AI models can be trained on large datasets of known ciphertexts and plaintexts.
* Over time, these models learn to predict or guess the decryption key or plaintext.

1. **Brute Force Optimization**:
   * Traditional brute force methods try every possible key. AI optimizes this by predicting the most likely keys to test first.
2. **Cryptographic Weakness Analysis**:
   * AI tools can find and exploit vulnerabilities in encryption algorithms faster than manual methods.

**Steps in AI-Driven Cryptanalysis**

1. **Data Collection**:
   * Gather samples of encrypted text (ciphertext) and, if possible, corresponding plain text.
2. **Preprocessing**:

* Clean and structure the data to feed into an AI model.

1. **Model Training**:
   * Train an AI model (like a neural network) using known examples to learn how to break similar ciphers.
2. **Prediction/Breaking**:

* Use the trained AI to predict decryption keys or decipher unknown encrypted messages.

**Examples of AI in Cryptanalysis**

1. **Breaking Historical Ciphers**:
   * AI has been used to decode ancient scripts like the Enigma Machine's messages from World War II.
2. **Testing Modern Encryption**:
   * Researchers use AI to test the security of modern cryptographic algorithms (e.g., AES, RSA) to ensure they’re resistant to attacks.
3. **Password Cracking**:
   * AI-driven tools can crack weak passwords by learning common password patterns and testing combinations efficiently.

**Challenges in AI Cryptanalysis**

1. **Complex Algorithms**:
   * Modern encryption methods like AES and RSA are designed to resist attacks. Breaking them requires vast computational resources.
2. **Data Requirements**:
   * AI needs large amounts of training data, which is not always available for every type of cipher.
3. **Ethical Concerns**:
   * Cryptanalysis using AI can be misused for illegal activities, such as hacking and spying.

**Advantages of AI in Cryptanalysis**

1. **Speed**:
   * AI can process and analyze data much faster than humans.
2. **Adaptability**:
   * Machine learning models can adapt to different types of encryption schemes.
3. **Efficiency**:
   * AI reduces the effort needed for manual cryptanalysis.

**Applications of AI in Cryptanalysis**

1. **Cybersecurity Testing**:
   * Companies use AI to test their encryption systems and find weaknesses before attackers do.
2. **National Security**:
   * Governments use AI tools to decrypt sensitive information for intelligence purposes.
3. **Historical Research**:
   * AI can help historians unlock the secrets of old, encoded documents.

**Ethical and Legal Considerations**

* Cryptanalysis using AI must be conducted responsibly:
  + **Legal Use**: Breaking codes should only be done with proper authorization.
  + **Ethical Boundaries**: Using AI for illegal hacking or to invade privacy is unethical.
* Cryptanalysis is about breaking codes, and AI makes this faster and more efficient.
* AI helps by recognizing patterns, optimizing brute force, and finding weaknesses in encryption.
* While powerful, it should be used responsibly to protect, not harm, society.