**UNIT-III**

**Use case diagrams https://www.uml-diagrams.org/index-examples.html**

**Analysis: Meta Model, Workflows,**

**Finding Analysis Classes – using noun/verb analysis, CRC analysis, using RUP stereotypes - entity, boundary and control;**

**Modeling Classes – Association (role name, multiplicity, navigability, association classes, qualified association) dependencies (usage, abstraction, permission), class generalization, generalization sets, power types.**

**Use Case Realization – Class diagrams, interaction diagram, sequence diagrams, collaboration diagrams, Activity Diagrams.**

**Analysis: Finding the classes and objects**

Analysis is an attempt to build a model that describes the application domain -- developers do this. Analysis takes place after (or during) requirements specification.

The analysis model will typically consist of all three types of models:

* Functional model (denoted with use cases)
* Analysis object model (class and object diagrams)
* Dynamic model

At this level, application domain needs to be considered. This is not yet system design. However, many things discovered in analysis could translate closely into the system design. The main goal is to completely understand the application domain. (the problem at hand, any constraints that must be adhered to, etc.) New insights gained during analysis might cause requirements to be updated.

Analysis activities include:

* Identifying objects (often from use cases as a starting point).
* Identifying the associations between objects.
* Identifying general attributes and responsibilities of objects.
* Modeling interactions between objects.
* Modeling how individual objects change state -- helps identify operations.
* Checking the model against requirements, making adjustments, iterating through the process more than once.

**Finding the objects**

Some categories of objects to look for:

* **Entity objects** -- these represent persistent information tracked by a system. This is the closest parallel to "real world" objects.
* **Boundary objects** -- these represent interactions between user and system. (For instance, a button, a form, a display)
* **Control objects** -- usually set up to manage a given usage of the system. Often represent the control of some activity performed by a system

UML diagrams can include a label known as a *stereotype*, above the class name in a class diagram. This would be placed inside << >> marks, like this:

* + <<entity>>
  + <<boundary>>
  + <<control>>

There are some different popular techniques for identifying objects. Two traditional and popular ones that we will discuss are:

* + natural language analysis (i.e. parts of speech)
  + CRC cards

It also helps to interact with domain experts -- these are people who are already well-versed in the realm being studied. The goal in the analysis phase is NOT to find implementation specific objects, like HashTable or Stack. This stage still models the application domain.

**Using natural language analysis**

This is pioneered by Russell Abbott (1983) and popularized by Grady Booch this technique is not perfect, but coupled with other techniques, it's a good start. This can be done from a general problem description, or better, from a use case or scenario.

It map parts of speech to object model components.

* + nouns usually map to classes, objects, or attributes
  + verbs usually map to operations or associations

|  |  |  |
| --- | --- | --- |
| **Part of speech** | **model component** | **Examples** |
| Proper noun | Instance (object) | Alice, Ace of Hearts |
| Common noun | Class (or attribute) | Field Officer, PlayingCard, value |
| Doing verb | Operation | Creates, submits, shuffles |
| Being verb | Inheritance | Is a kind of, is one of either |
| Having verb | Aggregation/Composition | Has, consists of, includes |
| Modal verb | Constraint | Must be |
| Adjective | Helps identify an attribute | a *yellow* ball (i.e. color) |

**Identifying different object types**

**Finding Entity Objects**

These may be candidates for objects, or they may help identify objects: 1) Terms that are domain-specific in use cases 2) Recurring nouns and 3) Real-world entities and activities tracked by system.

Use good naming conventions. Good to use names from the application domain -- they understand their own terminology best. For example: In a ReportEmergency use case -- "A field officer submits information to the system by filling out a form and pressing the 'Send Report' button". FieldOfficer is a real world entity that interacts with the system. This is also likely an *actor* from the use case. As an actor, FieldOfficer is an external entity But we see that the field officer submits *information* -- here's data to be tracked. We'll create the entity object type EmergencyReport, as that's the more common name for the information the officer submits (according to client)

**Finding Boundary Objects**

* Identify general user interface controls that initiate a use case (Don't bother with the visual details here.)
* Identify forms or windows for entering data into a system
* Identify messages used by system to respond to a user

**Finding Control Objects**

Control objects can help manage communication and interaction of other objects

* If a use case is complex and involves many objects, create a control object to manage the use case
* Identify one control object per actor involved in a use case
* Life span of control object should last through the use case

**CRC Cards**

A simple object-oriented analysis technique includes the users and developers in the analysis process. A CRC card is an index card with three parts:

* + *Class* -- name goes at the top of the card
  + *Responsibilities* -- as a list on the left side of the card
  + *Collaborators* -- as a list on the right side of the card

Here's the layout:

|  |  |
| --- | --- |
| **ClassName** | |
| *Responsibility* | *Collaborator* |
|  |  |
|  |  |
|  |  |
|  |  |

**Class**

* + Represents a type of object being modeled
  + One card per class

**Responsibility**

* + Something that the class knows (keeps track of) or does
  + These should be the high-level responsibilities. Not trying to list out all member functions here
  + Example: class Mailbox in a voice mail system might have these responsibilities:
    - keep new and saved messages
    - manage the recorded greeting

**Collaborator**

* + Another class that the current class has to work with to complete its responsibilities
  + Could be a class that has information we need
  + Could be a class that helps perform a task
  + Typically, we list a class as a collaborator if we (the current class) need to call upon it to help complete our own responsibilities
  + Example: for successfully keep new and saved messages, the Mailbox class has to send them to a MessageQueue to be added and stored. So on the Mailbox card, we list MessageQueue as a collaborator

**A CRC Card Session**

CRC cards can be used as a brainstorming technique, for the purpose of:

* Identifying objects/classes
* Identifying what each object's purpose is (responsibilities)
* Discovering the dependencies and relationships between objects (collaborators

A CRC card "session" involves users and developers:

* *Domain experts/users* -- intended users of the system, people who know the business being modeled. Good to have a few of these
* *Developer/Analyst* -- should have a couple members of the development team. People who understand OOP modeling and development processes
* *Facilitator* -- one person who keeps things on track and progressing forward

The process is based on going through use cases (or specific scenarios built from use cases), and using these to discover objects, responsibilities, and collaborators

The general process:

* Start with a scenario (usually representing a normal course through a use case)
* Identify initial classes/objects and make cards for them (this is can often be done by picking out the nouns)
* Going through a scenario helps identify responsibilities of a chosen object
* Identify collaborations between objects that have been created
* Sometimes, we'll identify a collaboration with a new object type that doesn't have a card yet -- this helps discover new classes
* When new classes are created, walk through scenarios again to discover any new responsibilities and collaborators (it's an iterative process)
* More use cases/scenarios will yield more classes, responsibilities, and collaborators

Finding responsibilities

* + Look for verbs in the scenario descriptions. These often tell us what an object *does*
  + Also ask what the class *knows*. This tells us what an object needs to store. Sometimes a primary responsibility of a class is management of certain unique information

Finding collaborators

* + If a class has a responsibility that required it to get, or modify, information it doesn't have on its own, it will need to collaborate with another class
  + Most often, one class specifically initiates the collaboration
  + Usually, the collaboration is a request for information or a request to do something
  + The *initiator's* card should list the helper class as a collaborator
  + In this case, the initiator class *depends on* the collaborator class to accomplish its tasks

### Analysis Class Stereotypes

Analysis classes may be stereotyped as one of the following:

* Boundary classes
* Control classes
* Entity classes

Apart from giving more specific process guidance when finding classes, this stereotyping result in a robust object model because changes to the model tend to affect only a specific area. Changes in the user interface, for example, will affect only boundary classes. Changes in the control flow will affect only control classes. Changes in long-term information will affect only entity classes. However, these stereotypes are specially useful in helping you to identify classes in analysis and early design. You should probably consider using a slightly different set of stereotypes in later phases of design to better correlate to the implementation environment, the application type, and so on.

#### Boundary Class

A **boundary class** is a class used to model interaction between the system's surroundings and its inner workings. Such interaction involves transforming and translating events and noting changes in the system presentation (such as the interface).

Boundary classes model the parts of the system that depend on its surroundings. Entity classes and control classes model the parts that are independent of the system's surroundings. Thus, changing the GUI or communication protocol should mean changing only the boundary classes, not the entity and control classes.

Boundary classes also make it easier to understand the system because they clarify the system's boundaries. They aid design by providing a good point of departure for identifying related services. For example, if you identify a printer interface early in the design, you will soon see that you must also model the formatting of printouts.

Common boundary classes include windows, communication protocols, printer interfaces, sensors, and terminals. You do not have to model routine interface parts, such as buttons, as separate boundary classes. Generally the entire window is the finest grained boundary object. Boundary classes are also useful for capturing interfaces to possibly nonobject oriented API's, such as legacy code.

You should model boundary classes according to what kind of boundary they represent. Communication with another system and communication with a human actor (through a user interface) have very different objectives. For communication with a human actor, the most important concern is how the interface will be presented to the user. For communication with another system, the most important concern is the communication protocol.

A boundary object (an instance of a boundary class) can outlive a use case instance if, for example, it must appear on a screen between the performance of two use cases. Normally, however, boundary objects live only as long as the use case instance.

##### Finding boundary classes

A boundary class intermediates the interface to something outside the system. Boundary objects insulate the system from changes in the surroundings (changes in interfaces to other systems, changes in user requirements, etc.), keeping these changes from affecting the rest of the system.

A system may have several types of boundary classes:

* **User interface classes** - classes which intermediate communication with human users of the system
* **System interface classes** - classes which intermediate communication with other system
* **Device interface classes** - classes which provide the interface to devices (such as sensors), which detect external events

##### ***Find user-interface classes***

Define one boundary class for each use-case actor-pair. This class can be viewed as having responsibility for coordinating the interaction with the actor. You may also define additional boundary classes to represent **subsidiary** classes to which the primary boundary class delegates some of its responsibilities. This is particularly true for window-based GUI applications, where you may model one boundary object for each window, or one for each form. Only model the key abstractions of the system; do not model every button, list and widget in the GUI. The goal of analysis is to form a good picture of how the system is composed, not to design every last detail. In other words, identify boundary classes only for phenomena in the system or for things mentioned in the **flow of events** of the analysis use-case realization.

##### ***Find system-interface classes***

A boundary class which communicates with an external system is responsible for managing the dialogue with the external system; it provides the interface to that system for the system being built.

Example

In an Automated Teller Machine, withdrawal of funds must be verified through the ATM Network, an actor (which in turn verifies the withdrawal with the bank accounting system). An object called ATM Network Interface can be identified to provide communication with the ATM Network.

The interface to an existing system may already be well-defined; if it is, the responsibilities should be derived directly from the interface definition. If a formal interface definition exists, it may be reverse engineered and we need not formally define it here; simply make note of the fact that the existing interface will be reused during design.

##### ***Find device interface classes***

The system may contain elements that act as if they were external (change value spontaneously without any object in the system affecting them), such as sensor equipment. Although it is possible to represent this type of external device using actors, users of the system may find doing so "confusing", as it tends to put devices and human actors on the same "level." Once we move away from gathering requirements, however, we need to consider the source for all external events and make sure we have a way for the system to detect these events.

If the device is represented as an actor in the use-case model, it is easy to justify using a boundary class to intermediate communication between the device and the system. If the use-case model does not include these "device-actors", now is the appropriate time to add them, updating the Supplementary Descriptions of the Use Cases where appropriate.

For each "device-actor", create a boundary class to capture the responsibilities of the device or sensor. If there is a well-defined interface already existing for the device, make note of it for later reference during design.

#### Control Class

A **control class** is a class used to model control behavior specific to one or a few use cases. Control objects (instances of control classes) often control other objects, so their behavior is of the coordinating type. Control classes encapsulate use-case specific behavior.

The behavior of a control object is closely related to the realization of a specific use case. In many scenarios, you might even say that the control objects "run" the analysis use-case realizations. However, some control objects can participate in more than one analysis use-case realization if the use-case tasks are strongly related. Furthermore, several control objects of different control classes can participate in one use case. Not all use cases require a control object. For example, if the flow of events in a use case is related to one entity object, a boundary object may realize the use case in cooperation with the entity object. You can start by identifying one control class per analysis use-case realization, and then refine this as more analysis use-case realizations are identified and commonality is discovered.

Control classes can contribute to understanding the system because they represent the dynamics of the system, handling the main tasks and control flows.

When the system performs the use case, a control object is created. Control objects usually die when their corresponding use case has been performed.

Note that a control class does not handle **everything** required in a use case. Instead, it coordinates the tasks of other objects that implement the functionality. The control class delegates work to the objects that have been assigned the responsibility for the functionality.

##### Finding control classes

Control classes provide coordinating behavior in the system. The system can perform some use cases without control objects (just using entity and boundary objects)-particularly use cases that involve only the simple manipulation of stored information.

More complex use cases generally require one or more control classes to coordinate the behavior of other objects in the system. Examples of control objects include programs such as transaction managers, resource coordinators, and error handlers.

Control classes effectively de-couple boundary and entity objects from one another, making the system more tolerant of changes in the system boundary. They also de-couple the use-case specific behavior from the entity objects, making them more reusable across use cases and systems.

Control classes provide behavior that:

* Is surroundings-independent (does not change when the surroundings change),
* Defines control logic (order between events) and transactions within a use case.
* Changes little if the internal structure or behavior of the entity classes changes,
* Uses or sets the contents of several entity classes, and therefore needs to coordinate the behavior of these entity classes.
* Is not performed in the same way every time it is activated (flow of events features several states).

##### Determine whether a control class is needed

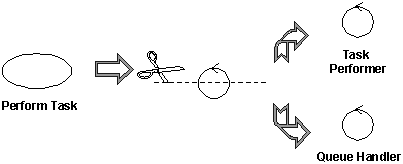
The flow of events of a use case defines the order in which different tasks are performed. Start by investigating if the flow can be handled by the already identified boundary and entity classes. For simple **flows of events** which primarily enter, retrieve and display, or modify information, a separate control class is not usually justified; the boundary classes will be responsible for coordinating the use case.

The **flows of events** should be encapsulated in a separate control class when it is complex and consists of dynamic behavior that may change independently from the interfaces (boundary classes) or information stores (entity classes) of the system. By encapsulating the **flows of events**, the same control class can potentially be re-used for a variety of systems which may have different interfaces and different information stores (or at least the underlying data structures).

**Example: managing a queue of tasks**

You can identify a control class from the use case Perform Task in the Depot-Handling System. This control class handles a queue of Tasks, ensuring that Tasks are performed in the right order. It performs the next Task in the queue as soon as suitable transportation equipment is allocated. The system can therefore perform several Tasks at the same time.

The behavior defined by the corresponding control object is easier to describe if you split it into two control classes, Task Performer and Queue Handler. A Queue Handler object will handle only the queue order and the allocation of transportation equipment. One Queue Handler object is needed for the whole queue. As soon as the system performs a Task, it will create a new Task Performer object, which will perform the Task. We thus need one Task Performer object for each Task the system performs.



Complex classes should be divided along lines of similar responsibilities

The principal benefit of this split is that we have separated queue handling responsibilities (something generic to many use cases) from the specific tasks of task management, which are specific to this use case. This makes the classes easier to understand and easier to adapt as the design matures. It also has benefits in balancing the load of the system, as many Task Performers can be created as necessary to handle the workload.

##### Encapsulating the main flow of events and alternate/exceptional flows of events in separate control classes

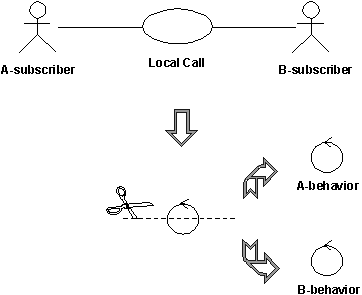
To simplify changes, encapsulate the main flow of events and alternate flows of events in different control classes. If alternate and exception flows are completely independent, separate them as well. This will make the system easier to extend and maintain over time.

##### Divide control classes where two actors share the same control class

Control classes may also need to be divided when several actors use the same control class. By doing this, we isolate changes in the requirements of one actor from the rest of the system. In cases where the cost of change is high or the consequences dire, you should identify all control classes which are related to more than one actor and divide them. In the ideal case, each control class should interact (via some boundary object) with one actor or none at all.

Example: call management

Consider the use case **Local Call**. Initially, we can identify a control class to manage the call itself.



The control class handling local phone calls in a telephone system can quickly be divided into two control classes, **A-behavior** and **B-behavior**, one for each actor involved.

In a local phone call, there are two actors: **A-subscriber** who initiates the call, and **B-subscriber** who receives the call. The **A-subscriber** lifts the receiver, hears the dial tone, and then dials a number of digits, which the system stores and analyzes. When the system has received all the digits, it sends a ringing tone to **A-subscriber**, and a ringing signal to **B-subscriber**. When **B-subscriber** answers, the tone and the signal stop, and the conversation between the subscribers can begin. The call is finished when both subscribers hang up.

Two behaviors must be controlled: What happens at A-subscriber's place and what happens at B-subscriber's place. For this reason, the original control object was split into two control objects, **A-behavior** and **B-behavior**.

You do not have to divide a control class if:

* You can be reasonably sure that the behavior of the actors related to the objects of the control class will never change, or change very little.
* The behavior of an object of the control class toward one actor is very insignificant compared with its behavior toward another actor, a single object can hold all the behavior. Combining behavior in this way will have a negligible effect on changeability.

#### Entity Class

An **entity class** is a class used to model information and associated behavior that must be stored. Entity objects (instances of entity classes) are used to hold and update information about some phenomenon, such as an event, a person, or some real-life object. They are usually persistent, having attributes and relationships needed for a long period, sometimes for the life of the system.

An entity object is usually not specific to one analysis use-case realization; sometimes, an entity object is not even specific to the system itself. The values of its attributes and relationships are often given by an actor. An entity object may also be needed to help perform internal system tasks. Entity objects can have behavior as complicated as that of other object stereotypes. However, unlike other objects, this behavior is strongly related to the phenomenon the entity object represents. Entity objects are independent of the environment (the actors).

Entity objects represent the key concepts of the system being developed. Typical examples of entity classes in a banking system are **Account** and **Customer.** In a network-handling system, examples are **Node** and **Link**.

If the phenomenon you wish to model is not used by any other class, you can model it as an attribute of an entity class, or even as a relationship between entity classes. On the other hand, if the phenomenon is used by any other class in the design model, you must model it as a class.

Entity classes provide another point of view from which to understand the system because they show the logical data structure, which can help you understand what the system is supposed to offer its users.

##### Finding entity classes

Entity classes represent stores of information in the system; they are typically used to represent the key concepts the system manages. Entity objects are frequently passive and persistent. Their main responsibilities are to store and manage information in the system.

A frequent source of inspiration for entity classes are the Glossary (developed during requirements) and a business-domain model (developed during business modeling, if business modeling has been performed).

### Association Stereotype Usage Restrictions

#### Restrictions for Boundary Classes

The following are allowable:

* Communicate associations between two Boundary classes, for instance, to describe how a specific window is related to other boundary objects.
* Communicate or subscribe associations from a Boundary class to an Entity class, because boundary objects might need to keep track of certain entity objects between actions in the boundary object, or be informed of state changes in the entity object.
* Communicate associations from a Boundary class to a Control class, so that a boundary object may trigger particular behavior.

#### Restrictions for Control Classes

The following are allowable:

* Communicate or subscribe associations between Control classes and Entity classes, because control objects might need to keep track of certain entity objects between actions in the control object, or be informed of state changes in the entity object.
* Communicate associations between Control and Boundary classes, allowing the results of invoked behavior to be communicated to the environment.
* Communicate associations between Control classes, allowing the construction of more complex behavioral patterns.

#### Restrictions for Entity Classes

Entity classes should only be the source of associations (communicate or subscribe) to other entity classes. Entity class objects tend to be long-lived; control and boundary class objects tend to be short-lived. It is sensible from an architectural viewpoint to limit the visibility that an entity object has of its surroundings, that way, the system is more amenable to change.

#### Summary of Restrictions

|  |  |  |  |
| --- | --- | --- | --- |
| **FromTo (navigability)** | **Boundary** | **Entity** | **Control** |
| **Boundary** | communicate | communicate  subscribe | communicate |
| **Entity** |  | communicate  subscribe |  |
| **Control** | communicate | communicate  subscribe | communicate |

Valid Association Stereotype Combinations

### Enforcing Consistency

* When a new behavior is identified, check to see if there is an existing class that has similar responsibilities, reusing classes where possible. Only when sure that there is not an existing object that can perform the behavior should you create new classes.
* As classes are identified, examine them to ensure they have consistent responsibilities. When classes responsibilities are disjoint, split the object into two or more classes. Update the interaction diagrams accordingly.
* If the a class is split because disjoint responsibilities are discovered, examine the collaborations in which the class plays a role to see if the collaboration needs to be updated. Update the collaboration if needed.
* A class with only one responsibility is not a problem, per se, but it should raise questions on why it is needed. Be prepared to challenge and justify the existence of all classes.

In UML models, an association is a relationship between two classifiers, such as classes or use cases, that describes the reasons for the relationship and the rules that govern the relationship.

An association represents a structural relationship that connects two classifiers. Like attributes, associations record the properties of classifiers. For example, in relationships between classes, you can use associations to show the design decisions that you made about classes in your application that contain data, and to show which of those classes need to share data. You can use an association's navigability feature to show how an object of one class gains access to an object of another class or, in a reflexive association, to an object of the same class.

The name of an association describes the nature of the relationship between two classifiers and should be a verb or phrase.

In the diagram editor, an association appears as a solid line between two classifiers.

## Association ends

An association end specifies the role that the object at one end of a relationship performs. Each end of a relationship has properties that specify the role of the association end, its multiplicity, visibility, navigability, and constraints.

## Example

In an e-commerce application, a customer class has a single association with an account class. The association shows that a customer instance owns one or more instances of the account class. If you have an account, you can locate the customer that owns the account. Given a particular customer, you can navigate to each of the customer’s accounts. The association between the customer class and the account class is important because it shows the structure between the two classifiers.

### Navigability

Associations between UML classes or data types may have a navigable direction. This can be from one class to the other (Source -> Destination or Destination -> Source), or both directions (Bi-Directional). Associations may also have unspecified direction, which in most cases will be the same as bi-directional, but this option should not be used. The navigable direction describe how implementations of the model can be navigated, and is **symbolized with arrows** in the diagram. In the figure above, all associations are navigable in both directions.

The navigable direction will have influence on implementation.

**For GML implementation, it is stated in ISO19136 Annex E that all navigable ends shall be marked with an arrow. In other words, the navigable direction shall always be set.**

Also, consider that bi-directional associations means that implementations of both classes know about each other. This may also make implementations more complicated - if both sides are [mandatory](https://github.com/ISO-TC211/UML-Best-Practices/wiki/Association-roles-and-navigability/_edit#multiplicity), both classes must have reference to the other in an GML implementation.

**Prefer one-way associations if it is not important that implementations of both classes know about each other**

### Role name

In implementations, associations will be similar to attributes. Therefore, it is important that

**each navigable end shall have a role name. This is also stated in ISO19136 Annex E - and it shall be an "NCName", in the same way as attribute names.**

In the figure above, an implementation of a Person class may have a reference to an implementation of a Car class, and this reference shall be called property. And vice versa - an implementation of a Car class shall have a reference to an implementation of a Person class, and the reference shall be called owner. But if we look at the association between Car and Wheel, we can see that the Car end have no name. So the Wheel does not know what to call its reference to a Car.

### Multiplicity

The multiplicity on an association end describes how many instances of one class that may or shall be referenced by the other. A Person may have none or many Cars (property), while a Car must have a Person (owner). A Commitee must have two or more Persons (members), and a Person can be part of none or many Commitees. And a Car must have three or more Wheels, while a Wheel can belong to none or one car.

Setting the multiplicity is also important for implementation.

**ISO 19136 Annex E states that multiplicity for all navigable end shall be set explicitly**

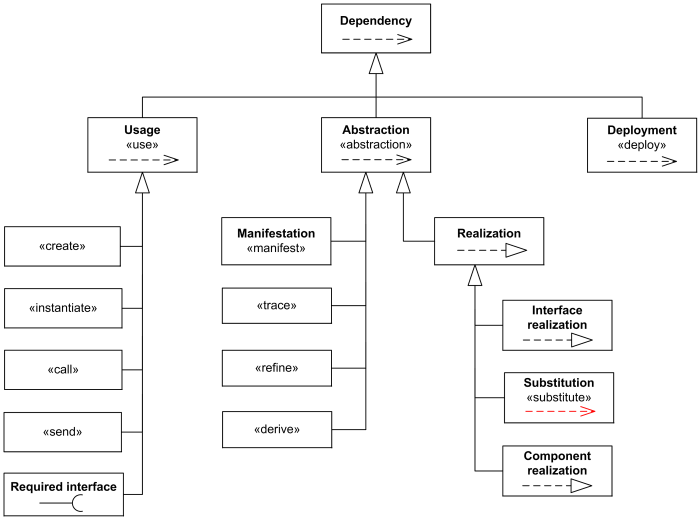
Multiplicity 1 and 1.. should be used with care.\*

The consequence of setting the multiplicity to 1 or 1..\* is that implementations of a class must always have this reference (similar to a mandatory attribute). If the multiplicity on the Car side of the Car-Wheel association had been 1, implementations of the Wheel class would have to have a reference to an implementation of a Car class. And the Wheel side is already mandatory (3..\*). If the model had been like this, both classes would have to have reference to the other in an implementation, and would likely have ended up with duplicate information.

# Dependency in UML

**Dependency** is a [**directed relationship**](https://www.uml-diagrams.org/uml-core.html#directed-relationship) which is used to show that some UML element or a set of elements requires, needs or depends on other model elements for **specification** or **implementation**. Because of this, dependency is called a **supplier** - **client** relationship, where supplier provides something to the client, and thus the client is in some sense incomplete while semantically or structurally dependent on the supplier element(s). Modification of the supplier may impact the client elements.

Dependency is a relationship between [**named elements**](https://www.uml-diagrams.org/uml-core.html#named-element), which in UML includes a lot of different elements, e.g. [**classes**](https://www.uml-diagrams.org/class.html), [**interfaces**](https://www.uml-diagrams.org/interface.html), [**components**](https://www.uml-diagrams.org/component.html), [**artifacts**](https://www.uml-diagrams.org/artifact.html), [**packages**](https://www.uml-diagrams.org/package-diagrams.html#package), etc. There are several kinds of dependencies shown on the diagram below.



***Dependency****relationship overview diagram - usage, abstraction, deployment.*

[**Usage**](https://www.uml-diagrams.org/dependency.html#usage) is a dependency in which one [**named element**](https://www.uml-diagrams.org/uml-core.html#named-element) (client) requires another named element (supplier) for its full **definition** or **implementation**.

The [**abstraction**](https://www.uml-diagrams.org/abstraction.html) relates two elements representing the same concept but at different levels of abstraction.

The [**deployment**](https://www.uml-diagrams.org/deployment-diagrams.html#deployment) is a dependency which shows allocation (deployment) of an artifact to a deployment target. (It is not very clear why UML 2.4.1 defines deployment as a dependency but not as an [**association**](https://www.uml-diagrams.org/association.html) or just a [**directed relationship**](https://www.uml-diagrams.org/uml-core.html#directed-relationship).)

Note, that **UML 2.4.1** specification has some confusing clarification that *"the presence of dependency relationships in a model does not have any runtime semantics implications, it is all given in terms of the model-elements that participate in the relationship, not in terms of their instances."*

This clarification contradicts to the definition of [**usage**](https://www.uml-diagrams.org/dependency.html#usage) dependency which is implementation dependency. An experienced software developer knows what happens at runtime when some dependency is missing, with application killed by LinkageError or ClassNotFoundException from the class loader. So dependency could in fact have some grave runtime semantics implications.

A dependency is generally shown as a dashed arrow pointing from the **client** (dependent) at the tail to the **supplier** (provider) at the arrowhead. The arrow may be labeled with an optional stereotype and an optional name. Because the direction of the arrow goes opposite to what we would normally expect, I usually stereotype it as client «depends on» supplier.

SearchController depends on (requires) SiteSearch interface.

*Class SearchController depends on (requires) SiteSearch interface.*

For many years UML specifications provide contradictory example of the dependency shown below. The explanation for the Figure 7.38 of UML 2.4.1 specification states: *"In the example below, the Car class has a dependency on the CarFactory class. In this case, the dependency is an instantiate dependency, where the Car class is an instance of the CarFactory class."*

UML spec inferior example: Car class has an instantiate dependency on the CarFactory class.

*Wrong: Car class has a dependency on the CarFactory class.  
Right: CarFactory class depends on the Car class.*

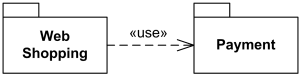
This example in fact shows opposite to what UML specification states. CarFactory depends on the Car class. Car class could be defined without the knowledge of CarFactory class, but CarFactory requires Car for its definition because it produces Cars. It is also wrong to say that *"... the Car class is an instance of the CarFactory class."* The Car class is **instantiated** by the CarFactory class.

It is possible to have a set of elements for the client or supplier. In this case, one or more arrows with their tails on the clients are connected to the tails of one or more arrows with their heads on the suppliers. A small dot can be placed on the junction if desired. A note on the dependency should be attached at the junction point.

Dependency could be used on several kinds of UML diagrams:

* [**class diagram**](https://www.uml-diagrams.org/class-diagrams-overview.html)
* [**composite structure diagram**](https://www.uml-diagrams.org/composite-structure-diagrams.html)
* [**package diagram**](https://www.uml-diagrams.org/package-diagrams-overview.html)
* [**component diagram**](https://www.uml-diagrams.org/component-diagrams.html)
* [**deployment diagram**](https://www.uml-diagrams.org/deployment-diagrams-overview.html)

Here's some examples of dependencies:

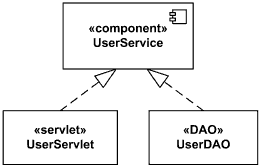


*Web Shopping package uses (depends on) Payment package.*

Interface realization dependency from a classifier to an interface.

***Interface****SiteSearch is****realized****(implemented) by SearchService.*

Note, that [**abstraction**](https://www.uml-diagrams.org/abstraction.html) dependency has a convention to have more abstract element as supplier and more specific or implementation element as client but UML specification also allows to draw it the opposite way.



*Component UserService realized by UserServlet and UserDAO.*

## *Usage*

A **usage** is a [**dependency**](https://www.uml-diagrams.org/dependency.html) relationship in which one element (**client**) requires another element (or set of elements) (**supplier**) for its full **implementation** or **operation**.

The usage dependency does not specify how the client uses the supplier other than the fact that the supplier is used by the definition or implementation of the client. For example, it could mean that some method(s) within a (**client**) class uses objects (e.g. parameters) of the another (**supplier**) class.

A usage dependency is shown as a dependency with a **«use»** keyword attached to it.

Usage dependency shown as a dependency with a use keyword.

*Class SearchController****uses****SearchEngine class.*

### Create

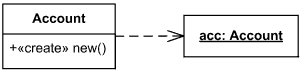
**Create** is a [**usage dependency**](https://www.uml-diagrams.org/dependency.html#usage) denoting that the client classifier creates instances of the supplier classifier. It is denoted with the standard stereotype **«create»**.

Client classifier creates instances of the supplier classifier.

*Class DataSource creates Connection.*

Create could also specify that the designated [**feature**](https://www.uml-diagrams.org/uml-core.html#feature) creates an instance of the classifier to which the feature is attached. This dependency may be promoted to the classifier containing the feature.

**Create** may relate an instance value to a constructor for a class, describing the single value returned by the constructor operation. The operation is the client, the created instance the supplier. The instance value may reference parameters declared by the operation.



*Account constructor creates new instances of Account*

**Instantiate** is another [**usage dependency**](https://www.uml-diagrams.org/dependency.html#usage) among classifiers indicating that operations on the client create instances of the supplier. It is denoted with the standard stereotype **«instantiate»**.

It is not very clear why **UML 2.4** standard has both **«create»** and **«instantiate»**.

### Call

**Call** is a [**usage dependency**](https://www.uml-diagrams.org/dependency.html#usage) that specifies that the source operation invokes the target operation. This dependency may connect a source operation to any target operation that is within the scope including, but not limited to, operations of the enclosing classifier and operations of other visible classifiers.

**Call** is denoted with the standard stereotype **«call»** whose source is an [**operation**](https://www.uml-diagrams.org/operation.html) and whose target is also an [**operation**](https://www.uml-diagrams.org/operation.html).

This relationship may also be applied to the [**class**](https://www.uml-diagrams.org/class.html) containing an operation, with the meaning that there exists an operation in the class to which the dependency applies.

### Send

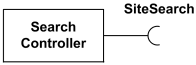
**Send** is a [**usage dependency**](https://www.uml-diagrams.org/dependency.html#usage) whose source is an [**operation**](https://www.uml-diagrams.org/operation.html) and whose target is a [**signal**](https://www.uml-diagrams.org/common-behaviors.html#signal), specifying that the source sends the target signal.

**Send** is denoted with the standard stereotype **«send»**.

### Required Interface

**Required interface** specifies services that a classifier needs in order to perform its function and fulfill its own obligations to its clients. It is specified by a [**usage dependency**](https://www.uml-diagrams.org/dependency.html#usage) between the [**classifier**](https://www.uml-diagrams.org/classifier.html) and the corresponding [**interface**](https://www.uml-diagrams.org/interface.html).

The **usage dependency** from a classifier to an interface is shown by representing the interface by a half-circle or socket, labeled with the name of the interface, attached by a solid line to the classifier that requires this interface.



***Interface****SiteSearch is****used****(required) by SearchController.*

If interface is represented using the rectangle notation, **interface usage** dependency is denoted with dependency arrow. The classifier at the tail of the arrow **uses** (requires) the interface at the head of the arrow.

Usage dependency from a classifier to an interface.

*Interface SiteSearch is****used****(required) by SearchController.*

Difference between Generalization and Specialization in DBMS

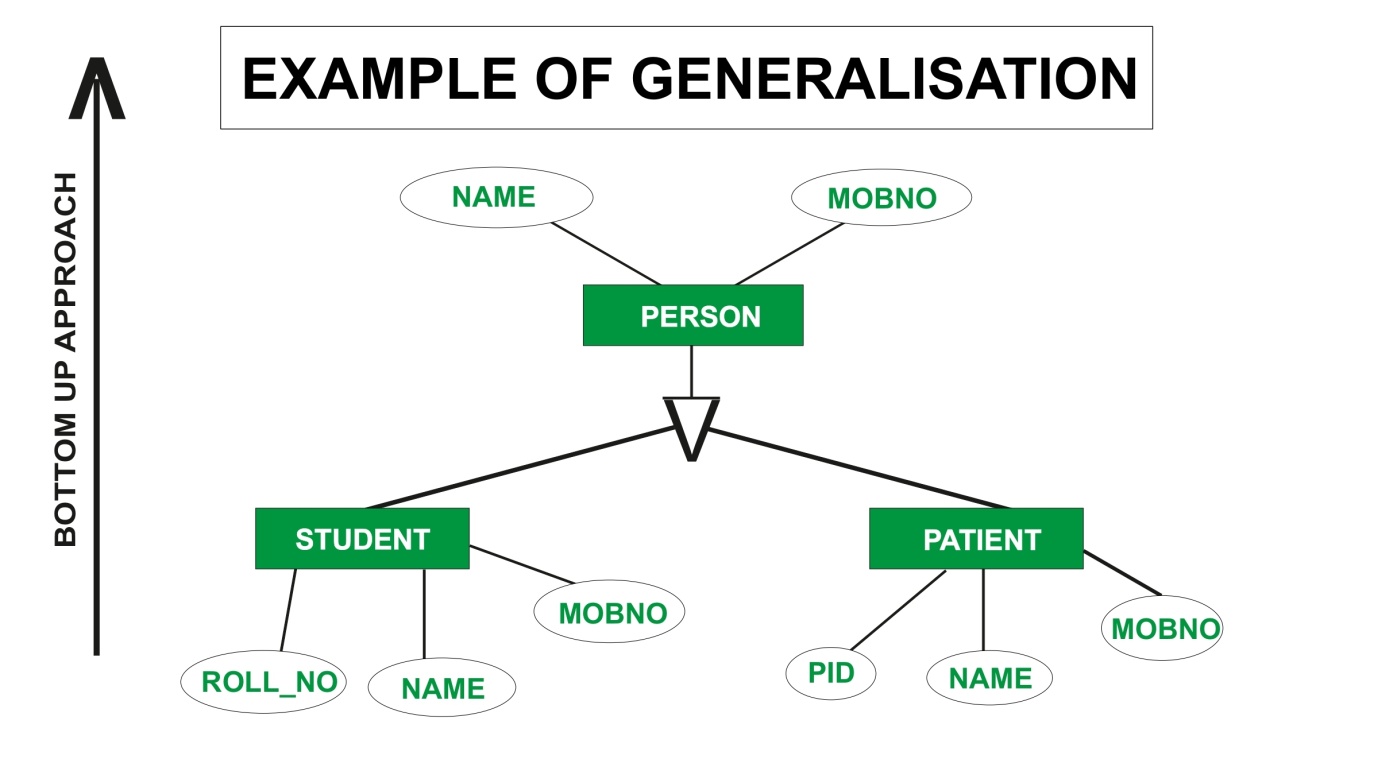
Last Updated: 20-05-2020

Generalization and specialization are the Enhanced Entity Relationship diagram (EER-diagram)

**1. Generalization :**  
It works on the principle of bottom up approach. In Generalization lower level functions are combined to form higher level function which is called as entities. This process is repeated further to make advanced level entities.

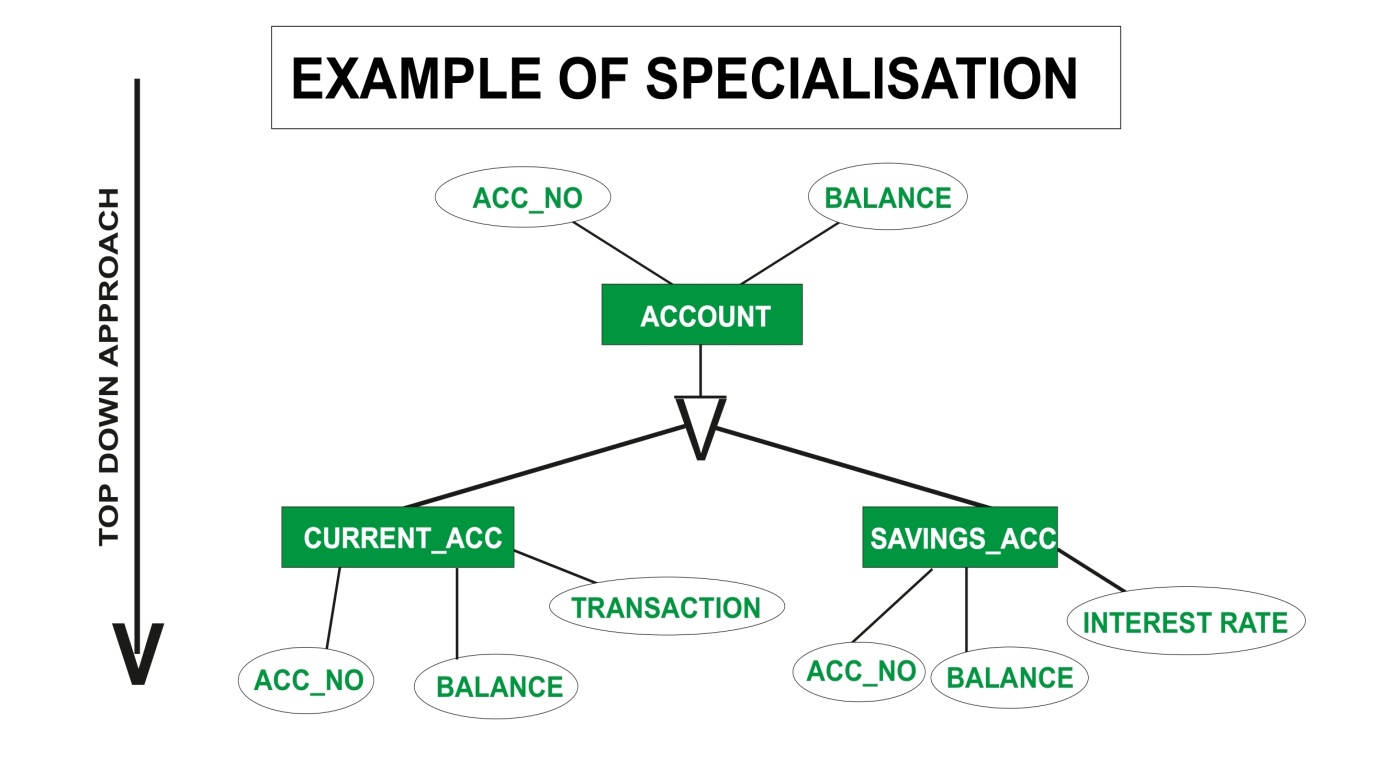
In the Generalization process properties are drawn from particular entities and thus we can create generalized entity. We can summarize Generalization process as it combines subclasses to form superclass.

**Example of Generalization –**  
Consider two entities Student and Patient. These two entities will have some characteristics of their own. For example Student entity will have Roll\_No, Name and Mob\_No while patient will have PId, Name and Mob\_No characteristics. Now in this example Name and Mob\_No of both Student and Patient can be combined as a Person to form one higher level entity and this process is called as Generalization Process.



**2. Specialization :**  
We can say that Specialization is opposite of Generalization. In Specialization things are broken down into smaller things to simplify it further. We can also say that in Specialization a particular entity gets divided into sub entities and it’s done on the basis of it’s characteristics. Also in Specialization Inheritance takes place.

**Example of Specialization –**  
Consider an entity Account. This will have some attributes consider them Acc\_No and Balance. Account entity may have some other attributes like Current\_Acc and Savings\_Acc. Now Current\_Acc may have Acc\_No, Balance and Transactions while Savings\_Acc may have Acc\_No, Balance and Interest\_Rate henceforth we can say that specialized entities inherits characteristics of higher level entity.



After applying generalization and specialization, the structure of resultant figures are same.

**Difference between Generalization and Specialization :**

| **GENERALIZATION** | **SPECIALIZATION** |
| --- | --- |
| Generalization works in Bottom-Up approach. | Specialization works in top-down approach. |
| In Generalization, size of schema gets reduced. | In Specialization, size of schema gets increased. |
| Generalization is normally applied to group of entities. | We can apply Specialization to a single entity. |
| Generalization can be defined as a process of creating groupings from various entity sets | Specialization can be defined as process of creating subgrouping within an entity set |
| In Generalization process, what actually happens is that it takes the union of two or more lower-level entity sets to produce a higher-level entity sets. | Specialization is reverse of Generalization. Specialization is a process of taking a subset of a higher level entity set to form a lower-level entity set. |
| Generalization process starts with the number of entity sets and it creates high-level entity with the help of some common features. | Specialization process starts from a single entity set and it creates a different entity set by using some different features. |
| In Generalization, the difference and similarities between lower entities are ignored to form a higher entity. | In Specialization, a higher entity is split to form lower entities. |
| There is no inheritance in Generalization. | There is inheritance in Specialization. |