Topic 3 Notes

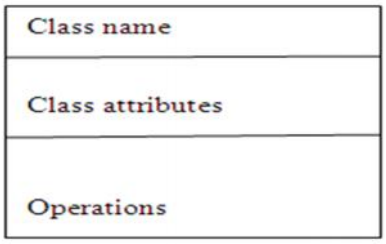
A class encapsulates characteristics common to a group of objects. We model the classes in our system based on the classes we have created. The UML enables us to model class diagrams and their interrelationships. Each class is modeled as rectangle with three components. A class diagram shows us what classes exist and how they're related. (Officially, class diagrams can also show attributes and operations, but that requires a lot more space.)

OVERVIEW OF THE CLASS MODELING PROCESS

1. Use the system requirements model to find candidate classes that describe the objects that might be relevant to the system and record them on a class diagram.
2. Find relationships (association, aggregation, composition and inheritance) between the classes.
3. Find attributes (simple, named properties of the objects) for the classes.
4. Walk through the system use cases, checking that they're supported by the objects that we have, fine-tuning the classes, attributes and relationships as we go – this use case realization will produce operations to complement the attributes.
5. Update the glossary and the nonfunctional requirements as necessary – the use cases themselves should not need updating, although perhaps they will need some correcting.

MODELING CLASSES

Each class is modeled as rectangle with the three compartments.



Finding (Identifying) Classes

The process identifies three types of classes

Entity classes

Models information that is long lived in the system e.g. account class is an entity class in bank software because information on accounts must stay in the software product. A class inside the system, representing a business concept such as a customer, a car or a car model and containing useful information. Typically, entities are manipulated by boundary and controller objects, rather than having much behavior of their own. Entity classes are the ones that appear on our analysis class diagram. Most entities survive through to design

Boundary classes

Models the interaction between the software product and its actors e.g print customer withdrawals report class. This is class at the edge of the system, between the system and the actors. For system actors, boundaries provide a communication path. For human actors, a boundary means a user interface, capturing commands and queries and displaying feedback and results. Each boundary object usually corresponds to a use case, or a group of related use cases. More specifically, such a boundary usually maps to a user interface sketch (in which case it may be an entire interface or just a sub-window). It is quite reasonable for boundary objects to survive through to design

Control class

Models complex computations and algorithms e.g calculate loan balance class. A class inside the system that encapsulates a complex or untidy process. A controller is a service object that provides the following kinds of service: control of all or part of a system process; creation of new entities; retrieval of existing entities. Without controllers, our entities would become polluted with messy details. Since controllers are just a convenience for the benefit of analysis, we do not expect many of them to survive through to design; an important exception to this is the idea of a home. A home is a controller that is used for the creation of new entities and the retrieval of existing ones.

Object identification

Noun extraction technique

There are formal methods for object identification in the requirements specification documents. The easiest is the noun extraction technique. Use these links for more information on object identification https://www.youtube.com/watch?v=V3n8qtOW7gM or https://www.youtube.com/watch?v=P2X9N4-xEvc or https://www.youtube.com/watch?v=5d2blHVH3h8 for more information. Going through the

processes of business requirements modeling and system requirements modeling, we have a good source of candidate classes in the form of system use cases. Candidate classes are often indicated by nouns in the use cases. Identify the nouns in the formal strategy; exclude those that lie outside the problem. These nouns are known as entity classes.

We review the requirements documents and verify key noun phrases to help in identifying classes that comprises the system. We may decide that some of these nouns and noun phrases are attributes of other classes in the system. Also identify abstract nouns. Abstract nouns rarely end up corresponding to classes. Identify also some of the nouns that do not correspond to of parts of the system and therefore should not be modeled. Additional classes may become apparent to us as we proceed through the design process. We create classes only for nouns and noun phrases that have significance in the system.

Identifying Class Relationships

According to the UML standard, all run-time relationships come under the umbrella term association. However, most people use the term 'association' to mean 'an association that isn't aggregation or composition'. Choosing between relationships can be tricky – you need to use intuition, experience and guesswork. During analysis, you should expect the frequency of these kinds of relationship to be: **association > aggregation > inheritance > composition**

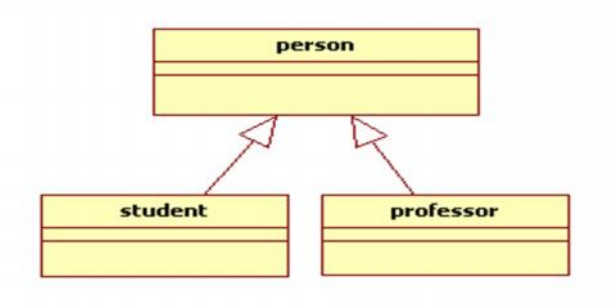
Once we have a list of candidate classes, we can try to draw relationships between them. Class diagrams show relationships between classes of the system. A solid line connecting classes represents a relationship.

THERE ARE FOUR POSSIBLE TYPES OF RELATIONSHIP:

1. **Inheritance:**

A subclass inherits all of the attributes and behavior of its super class(es). Inheritance models "is-a" and "is like" relationships enabling you to reuse existing data and code easily. Inheritance in UML is depicted by a peculiar triangle arrow head, the arrowhead points the base class. Inheritance is a different kind of relationship to the other three: inheritance describes a compile-time relationship between classes while the others describe a run-time connection between objects.

According to the UML standard, all run-time relationships come under the umbrella term association. However, most people use the term 'association' to mean 'an association that isn't aggregation or composition'. How inheritance is depicted on a class diagram, a white filled arrowhead on a solid line is drawn from the subclass to the superclass. In order to emphasize hierarchies of



Specialization

Specialization is used to define an entity type (class) that represents a specific segment of a large entity type.

Generalization

Generalization entity type (parent type) represents the common structure and behavior of all subtypes and contains all of the common attributes for child entity types.
• Inheritance: Trains inherit the characteristics of land vehicles.
• Generalization/specialization: A train is more specialized than a land vehicle; a land vehicle is more generalized than a train.

2) Association

Objects of one class are associated with objects of another class i.e objects will relate to each other.
An association between classes is shown by a line joining the two classes
Occasionally, an association has some information or behavior related to it. An association class can be introduced alongside the association, as depicted in the figure below.

When you model associations in UML class diagrams you show a thin line connecting two classes.

The association indicates that a CarModel can be associated with any number of Customer objects and a

Customer can be associated with any number of CarModel objects. Each relationship link has two ends called roles and each role has a name, multiplicity, navigability and type. Association label, indicates the nature of the association. If it's not obvious which way the association name should be read, a black arrowhead can be used

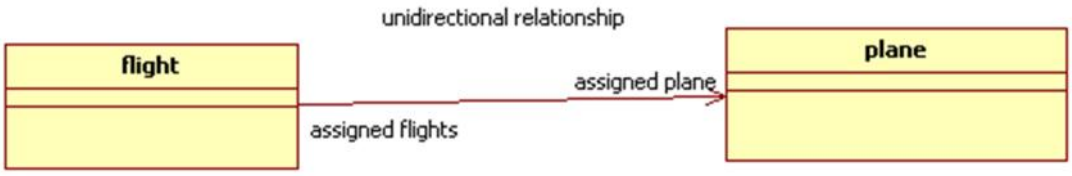
Name

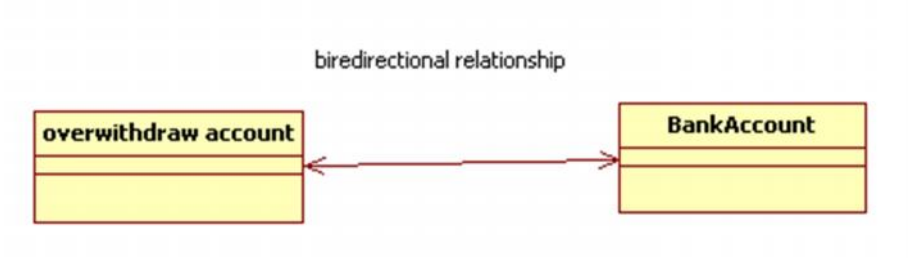
Name is represented by a label indicating what the relationship does facilitating understanding. Through association names, we can show roles. A role indicates the part played by an object in the association – the role is shown as a label near the object that plays the role – the role is shown as a label near the object that plays the role

Navigability

The navigability of a role shows which class has the responsibility of maintaining the relationship between the classes. It indicates which end of a relationship is aware of the end. A **navigability** arrow on an association shows which direction the association can be traversed or queried. The arrow also lets you know who "owns" the association's implementation.

Relationships can be **bidirectional navigability** (each end is aware of the other) or single **bidirectional navigability** (unidirectional) i.e. one end is aware of the other.





Association Labels, Roles and Comments

Association label, indicating the nature of the association. If it's not obvious which way the association name should be read, a black arrowhead can be used. For example, in Figure 7.8, we can see that there is at least one Wheel that turns the Car.

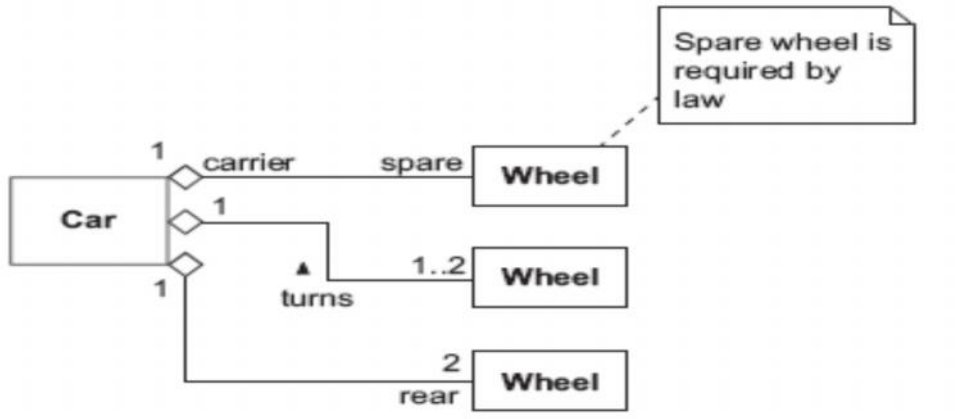


Figure 7.8: Association labels, roles and comments in UML

As well as association names, we can show roles. A role indicates the part played by an object in the association – the role is shown as a label near the object that plays the role. For example, Figure 7.8 indicates the following roles:

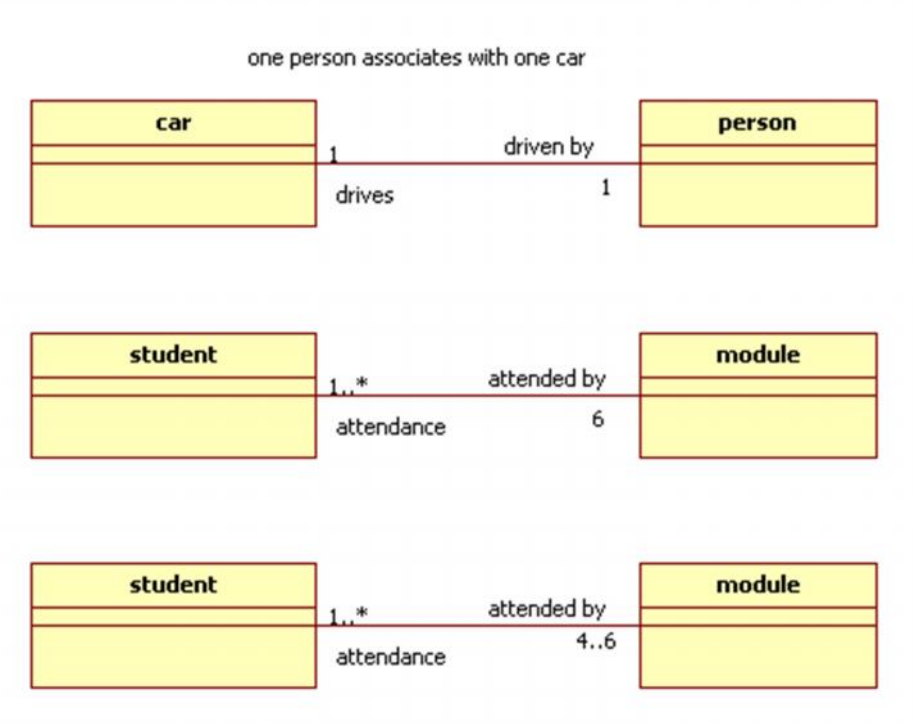
- A Car has one Wheel acting as a spare.
- The spare Wheel has one Car acting as its carrier.
- A Car has two rear wheels.

Multiplicity Relationship

The multiplicity of a role shows the number of objects that relate with a number of objects in a related class. It indicates how many objects can participate in a relationship .Multiplicity is a special type of association which shows the number of objects from one class that can relate with a number of objects in a associated class.

Notation for multiplicities

1 exactly one
n (where n is a single number)
* (the range 0 to infinity)
1..* (the range 1 to infinity)
n1..n2 (where n1 and n2 are single numbers forming a range)
0..1 (zero or one)
• * Shorthand for 0..*.
• 0..1 Optional



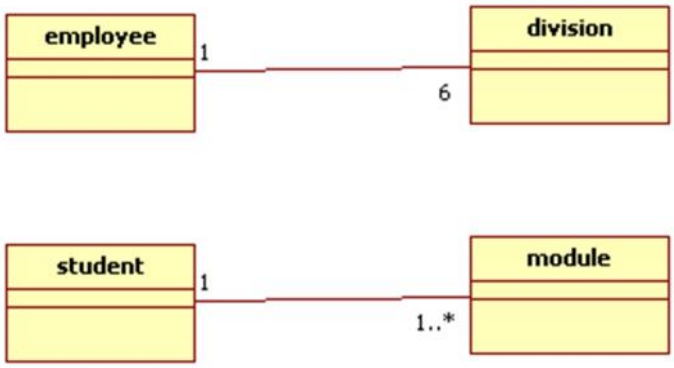
Types of Multiplicity Relationships

One-to-one relationship.

This is a relationship where the maximum of each multiplicity is one e.g one employee holds one position and one position may be held by one employee.

One-to-many relationship

This occurs where the maximum of one multiplicity is one and the other is greater than one. e.g. employee and division an employee works in one division and any given division has one or more employees working in it.



Many-to-many

This is a relationship where the maximum of both multiplicities is greater than one. E.g employee and task. An employee is assigned one

Others

One-to- one or more

One-to-exactly n.

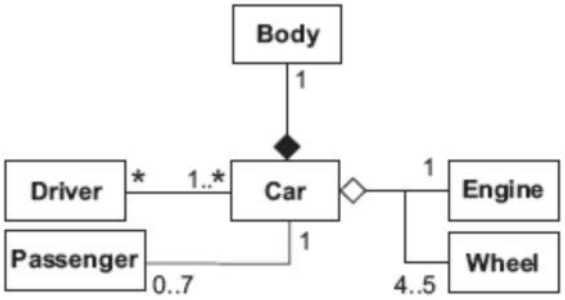


Figure 7.7: Depicting multiplicities in UML

3) Aggregation

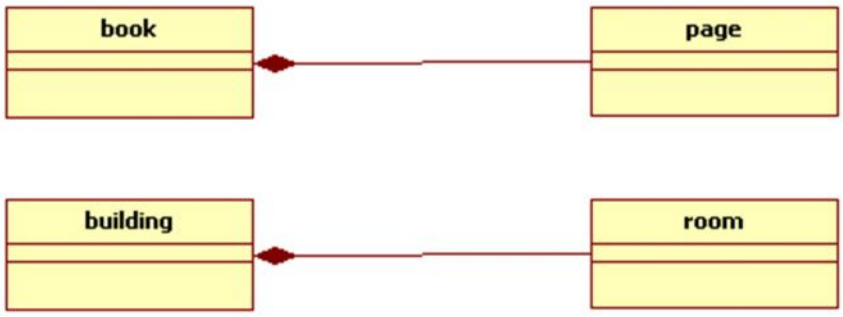
Aggregation: an instance of one class is made up of instances of another class.

Its weak form of composition, the two parts can exit independently.(“has- a”/ “part of”). Aggregations usually form a part–whole hierarchy. Aggregation implies close dependency, at

least of. Open diamond denotes aggregation and the diamond goes at the end of the whole not the part (aggregate class). Aggregation is drawn as a line between two classes with a white diamond on the aggregator's end.

4) Composition

Composition: Strong aggregation – the composed object can't be shared by other objects and dies with its composer. The part exits only as long as the whole exits, the whole is responsible for creation and destruction of the parts. Composition is shown by a black diamond on the end of association next to the composite class. Composition is drawn in a similar way to aggregation, but with a black diamond on the composer's end



Indicate whether this is association, aggression and composition

i. Houses on a street
ii. Pages in a book
iii. Notes in symphony
iv. Components in a home entertainment system (television, VCR, tape decks, amplifier, games console)
v. Railway carriage and train
vi. Student and class
vii. Book and pages
viii. library and information/enquiry desk

5) Dependency

Sometimes the relationship between a two classes is very weak. They are not implemented with member variables at all. Rather they might be implemented as member function arguments. This is shown by a dashed arrow between the two classes.

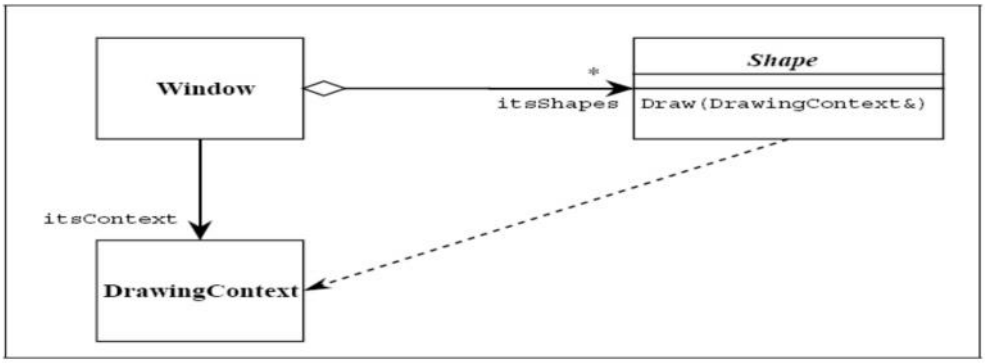


Figure above shows a dashed arrow between the Shape class and the DrawingContext class. This is the dependency relationship. In Booch94 this was called a 'using' relationship. This relationship simply means that Shape somehow depends upon DrawingContext. In C++ this almost always results in a #include.

Drawing Relationships

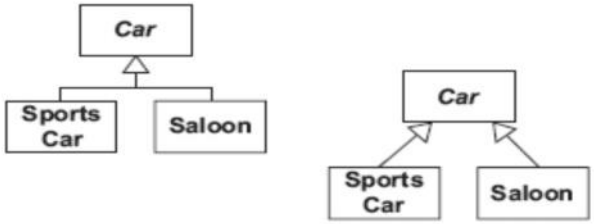


Figure 7.3: Depicting inheritance in UML

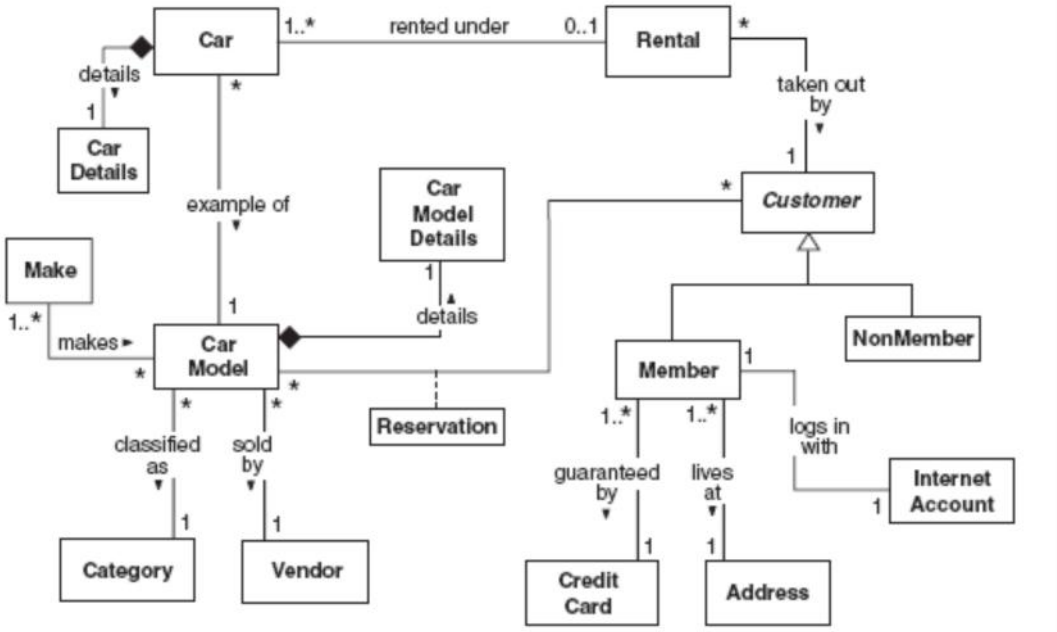
Figure 7.4: Depicting aggregation in UML

Figure 7.5: Depicting composition in UML

Figure 7.6: Depicting association in UML

Drawing Class and Object Diagrams

The figure below shows a UML class diagram for iCoot. Every class is represented as a box with the class name inside (in bold, if not drawing by hand). If the class is abstract, the class name is italicized. If you're labeling an abstract class by hand, you can add the keyword {abstract} above or to the left of the class name instead of using italics.



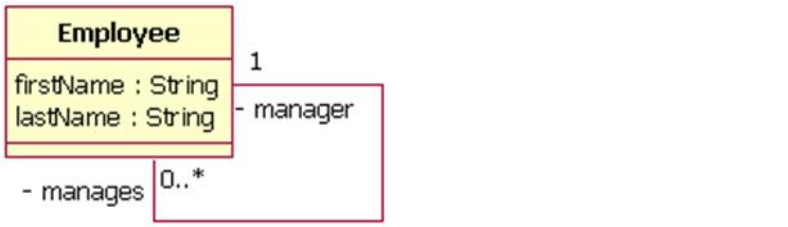
In object diagrams, objects are shown as boxes connected by links – the links are 'realized' associations.

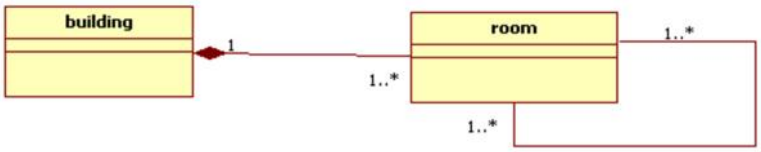
5) Reflexive Associations

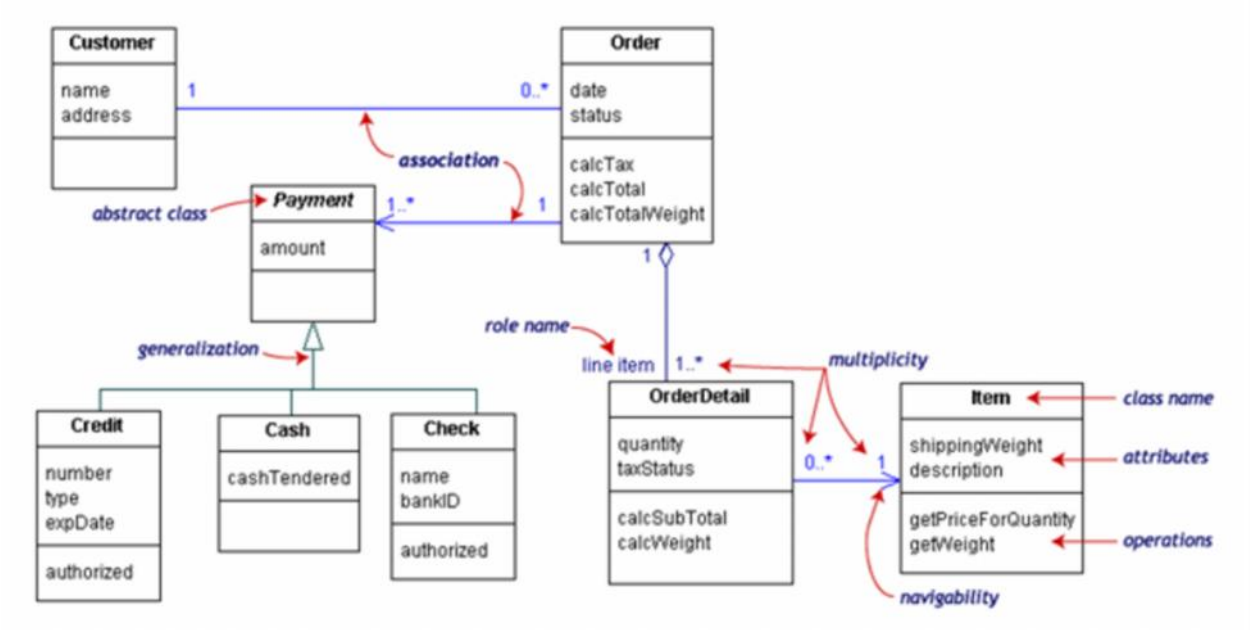
There exists a concept called reflexive or recursive relationship where a class is in relationship with itself.

However, a class can also be associated with itself, using a reflexive association. The figure shows how an Employee class could be related to itself through the manager/manages role. When a class is associated to itself, this does not mean that a class's instance is related to itself, but that an instance of the class is related to another instance of the class.

Figure 14: Example of a reflexive association relationship







Attributes

An attribute is a property of an object, such as its size, position, name, price, font, interest rate, or whatever. In UML, each attribute can be given a type, which is either a class or a primitive. If we choose to specify a type, it should be shown to the right of the attribute name, after a colon. (We might choose not to specify attribute types during analysis, either because the types are obvious or because we don't want to commit ourselves yet.). If you can't provide a short description for an attribute at this stage, perhaps it should be several attributes, or even a class in its own right. Consider the attributes of some real world objects. A person attributes include name, height, weight, complexion, eye color

A radio attributes include its station, volume and frequency.

A car's attributes includes its speedometer, odometer reading, amount of gas in the tank, what gear its in etc

Attributes are parts of the essential description of a class. Attributes are the common structures of what a member of a class can know. Each class has its own unique value of each attribute.

The current state of an object is defined by its attributes. Some attributes values change during the lifetime of an object e.g. library member may change his name, address.

When an attribute value changes, the object itself may change.

We can identify many attributes of the classes in a system by looking for descriptive words and phrases in the requirements documents. For each such words and phrases we find a significant role in the system. We also create attributes to represent any additional data that a class may need becomes clear throughout the design process. Descriptive words and phrases in the requirements also suggest some differences in the attributes required by each transaction.

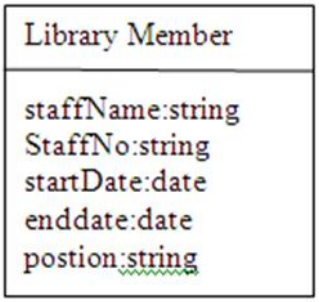
Modeling Attributes

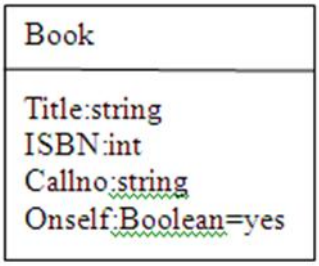
Attributes can be shown on a class diagram by adding a compartment under the class name. To save space, we can document them separately instead as an attribute list, complete with descriptions. If we were using a software development tool, we would expect to be able to zoom in to see attributes (and their descriptions) or zoom out to see class names only. Attributes are placed in the middle compartment of the class rectangle.

We list attributes name and type separated by colon (:), followed in some cases by equal sign(=) and an entail value. e.g userAuthenticated:Boolean=false

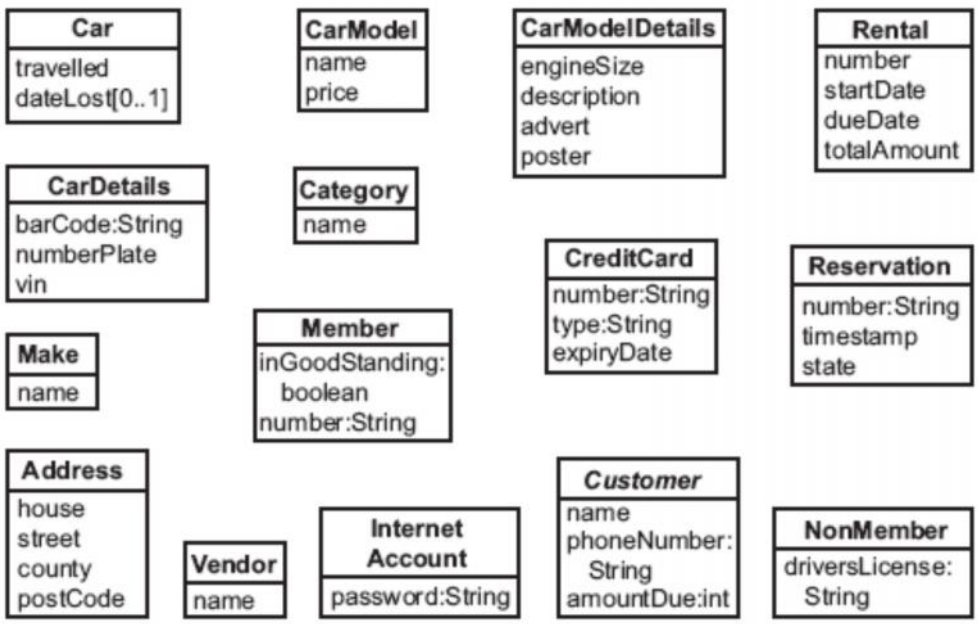
The attribute declarations contains three pieces of information

1. attribute name-is userAuthenticated
2. attribute type is Boolean(int, double-primitive data types)
3. attribute value-indicate an initial value for an attribute.





Although UML does allow us to define our own primitives in language-independent notation – Integer, Real and Boolean, for example – you might like to avoid using this facility because, when you come to design, you will have to be language-specific. (Another reason to avoid this issue is that, in Java, types such as Integer are classes, not primitives.)



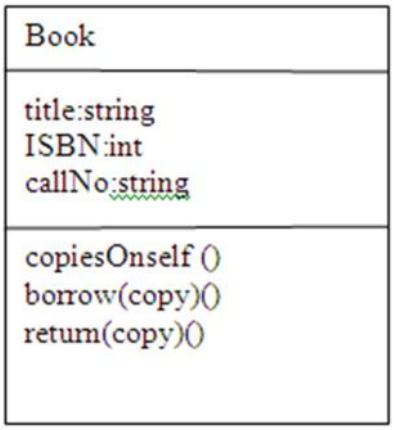
Identify Object Behaviors, Operations States and Activities

Attributes represents an objects state. We identify some key states that our objects may occupy and discuss how objects change state in response to various to various events occurring in the system. An object in system goes through a series of states. An object's state is indicated by the values of its attributes at a given time. Operations define the ways in which objects may interact. Operations are the elements of common behavior shared by all instances of a class. They are actions that be can be carried out by or an object. Operations represents aspects of behavior of objects or classes Operations are aspects of behavior required to simulate the way that application domain works. Operations are implemented by methods. When an object sends a message to another, it is asking the receiver to perform an operation. The receiver will invoke a method to perform the operation. Operations are implemented in the third compartment of the rectangle model.

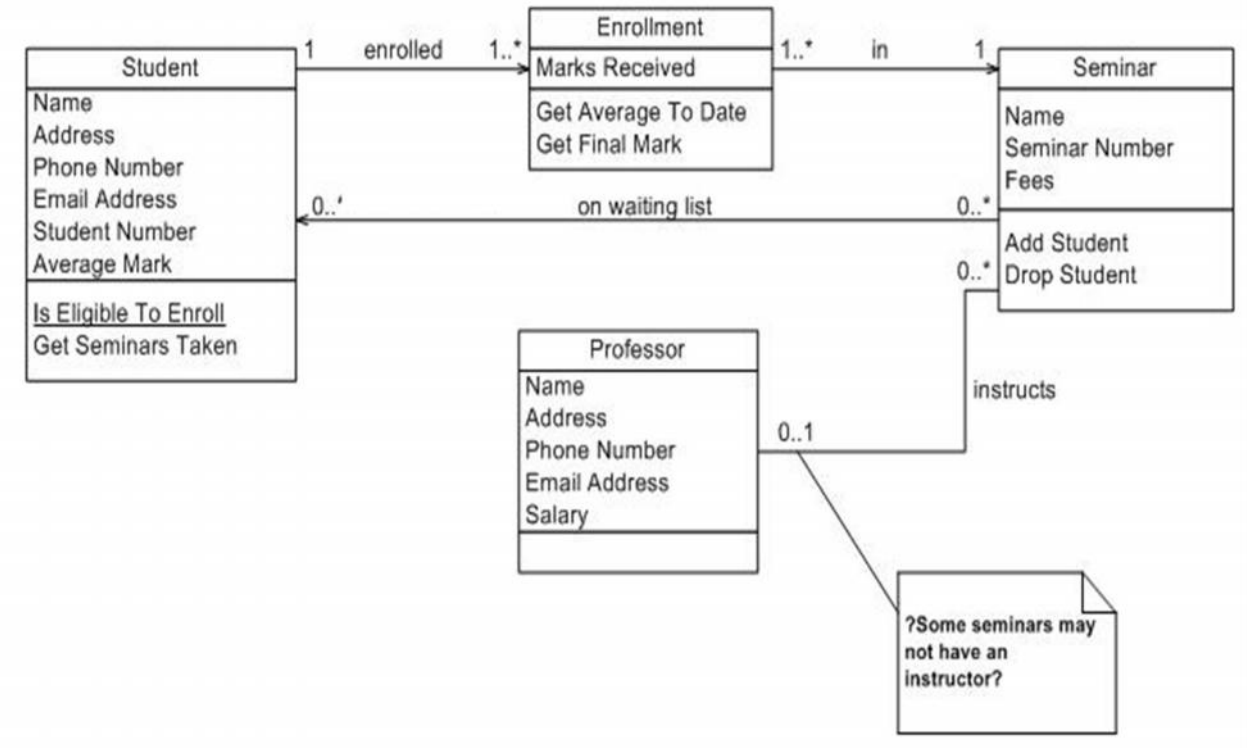
Modeling Operations

Operations are (modeled) implemented in the third compartment of the rectangle model.


Library Member
staffName:string
StaffNo:string
startDate:date
enddate:date
postion:string
registerNewMember()
assignMemberToPosition


Book
title:string
ISBN:int
callNo:string
copiesOnself ()
borrow(copy)()
return(copy)()


Course
Name
Course Number
Fees
getFullName()
getCourseNumber()
setCourseNumber(number)
getFees()
setFees(amount)
getName()
setName(name)



Abstract Classes

An **abstract class** is a class with at least one abstract method – the abstract method may be introduced on the class itself, or it may be inherited from a superclass
Abstract classes have the following advantages:
• They permit richer and more flexible modeling
• They lead to more code sharing, because we can write concrete methods that use abstract methods.

Scenarios

Scenario 1

Books and journals

The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow upto 12 items at any one time. Only members of staff may borrow journals

Candidate classes –objects identified using the noun and noun phrases technique described above

2. Library
3. Short term loan
4. Member of staff

5. Library member
6. Week
7. Item
8. Time
9. System
10. Book
11. Journal
12. Copy of book

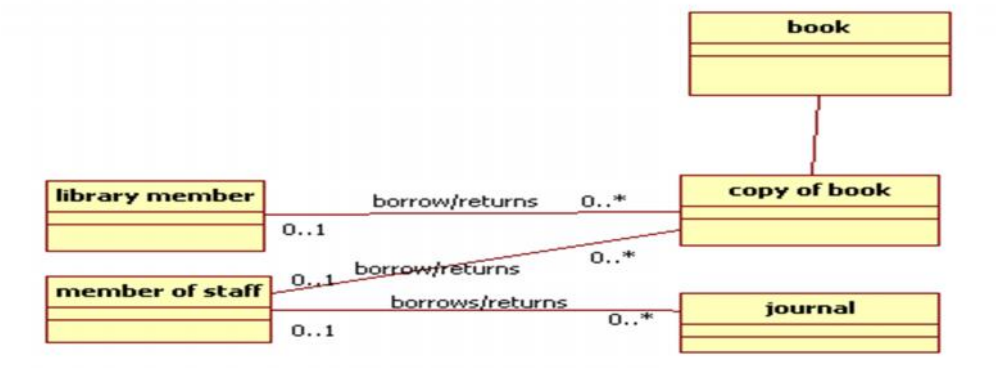
Refine the list of candidate classes (discard/eliminate from the list classes that you feel should not be part of the system)

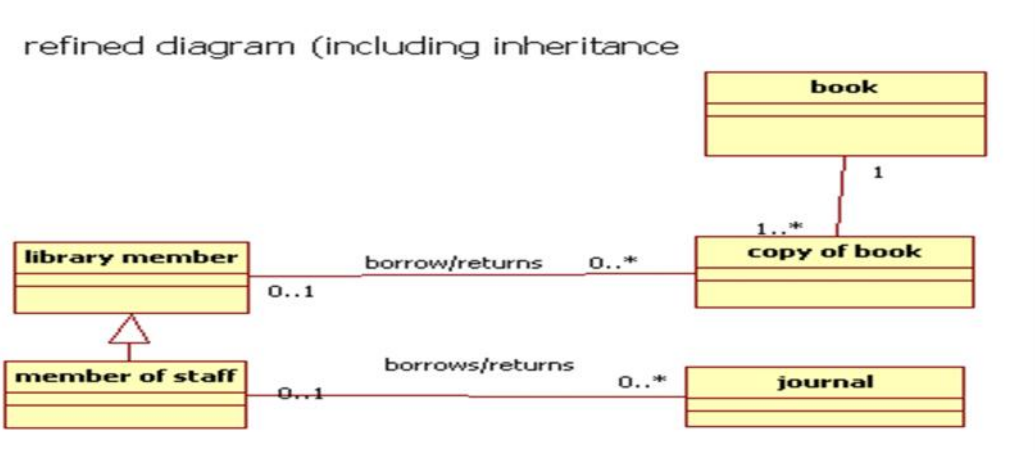
1. Library- out of scope(meta data)
2. Short term loan-its an event
3. **Member of staff**
4. **Library member**
5. Week-measure of time
6. Item-vague to mean book or journal
7. Time-outside the scope
8. System-meta language for requirements description.
9. **Book**
10. **Journal**
11. **Copy of book**

List of classes after refining

1. **Book**
2. **Journal**
3. **Copy of book**
4. **Member of staff**
5. **Library member**

Class diagram





Scenario 2

Buttons in elevators and on the floors control the movement of 8 elevators in a building in building with 20 floors. Buttons illuminate when pressed to request an elevator to stop at a specific floor. The illumination is cancelled when the request has been satisfied. When an elevator has no requests it remains at its current floor with its doors closed.

Candidate classes identified

1. Floor
2. Building
3. Door
4. Movement
5. Illumination
6. Request
7. Elevator
8. Button

Refine the list of classes

1. Floor-outside the problem
2. Building- outside the problem
3. Door- outside the problem
4. Movement-abstract classes (no physical existence)
5. Illumination-abstract classes (no physical existence)
6. Request-abstract classes (no physical existence)

List of classes after refining

1. Elevator
2. Button

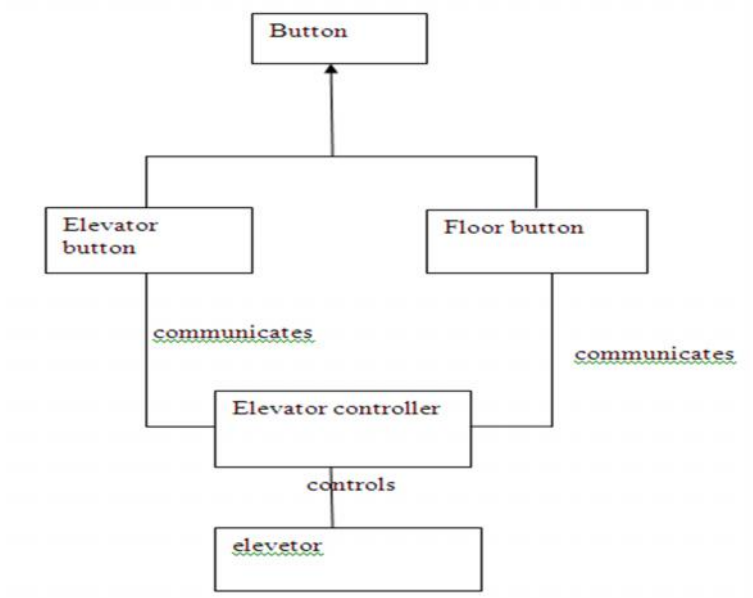
Create objects from the classes (you can create additional controller and boundary classes)

Elevator

1. Elevator door controller
2. Elevator controller

Elevator Button

1. Open door button
2. Close door button
3. Request floor button
4. Floor button



Follow up questions

A golf club wants to develop software to support a number of its activities. The club secretary will use the system to manage membership details, which includes adding and removing members. To become a member an application has to be recommended by two current members. The secretary can add tournaments as well as printing tournament results. Members of the club can enter tournaments if they wish and can view results