Object-Oriented Analysis and Design

Training Course Material

Source of information

1. Object-Oriented Analysis and Design by Grady Booch
2. The Unified Modeling Language User Guide by Grady Booch, James Rumbaugh, Ivar Jacobson
3. UML Distilled by Martin Fowler
4. Notes from previous OOAD training course

Pre-requisite for this course

Knowledge of an object-oriented programming language such as C++ or Java etc. is assumed. This course is not meant for newcomers to software industry.

TABLE OF CONTENTS

OBJECT-ORIENTED ANALYSIS AND DESIGN....1

1.1 Introduction....1
1.2 OOAD Process Outline....1
1.3 What are classes and objects....2
1.3.1 Objects....2
1.3.2 Classes....2
1.3.3 Class Vs Object....3
1.4 Abstraction and Encapsulation....4
1.5 Relationship among classes and objects....5
1.6 Visual Modeling....7
1.7 UML – Unified Modeling Language....8
1.8 Tools....13
1.9 Analysis and Design....14
1.9.1 Object-Oriented Analysis....14
1.9.2 Object-Oriented Design....16
1.10 Modeling in the real world: an example....19
1.11 Modeling in the real world: an example....19
1.12 Modeling in the real world: an example....19
1.13 Why OOAD?....19
1.14 Object-Oriented Metrics....20
1.15 Books and References....20
1.16 Assignments....21

Object-Oriented Analysis and Design

1.1 Introduction

Object-Oriented Analysis and Design (OOAD) has established itself as a proven method of designing software systems however complex. The ability to visualize a problem in terms of its objects and their interactions is key to understanding object-oriented analysis and design. It takes a while to think in terms of objects in the software world especially if the background is procedural and algorithmic.

Since the concept of collaborating objects is closer to the real world, therefore it is easier to follow.
Unlike Structured Systems Analysis and Design, which follows a waterfall model, OOAD is an incremental and iterative process.

1.2 OOAD Process Outline

In summary the whole process of OOAD is discovering abstractions and refining them. Looking ahead at the process, following are a few steps involved in object-oriented analysis and design.

- Use Case Analysis
- Scenario Analysis through Sequence diagrams
- Identifying Classes and Objects
- Identifying attributes
- Identifying behavior
- Class Diagrams
- Identifying relationship among classes
  - Inheritance
  - Association
  - Aggregation
- Refining and detailing the classes and relationships
- Splitting or merging classes
- Inventing additional (implementation level) classes
- Component Diagrams
- Deployment Diagrams

Broadly,
Analysis or Outside view

a) Identify classes and objects that form the vocabulary of the problem domain. (Entities)
b) Identify how these objects work together to meet the requirements of the problem. (Relationships)

Design or Inside view

c) Refine the relationship between classes and objects.
d) Discover new classes (e.g. helper, controller, and manager).
e) Detail the classes for implementation.

Since OOAD is incremental and iterative in nature, the steps above are not a one pass to completion procedure.

1.3 What are classes and objects

Conceptually classes and objects are closely tied together. A class defines the structural understanding of an object, whereas, an object is an entity which actually exists in time and space.

1.3.1 Objects

What is an object?

It is an instance of a class. E.g. If "Microprocessor" is a class, then the Pentium chip in a certain PC is an object. If "Company" is a class, then "PCS" is an object.

Object is an entity, which has Identity, State and Behavior. It can either be a physical/tangible object or a conceptual/intellectual object. Every object has a role to play while handling some responsibilities. According to Booch, objects exist in time, have state, can be created and destroyed.

Identity of an object is a characteristic, which establishes its uniqueness with respect to all other objects.
If an object has state, but no behavior, it is a dead object and it could as well be a structure.

Behavior of an object is implemented in terms of operations. Operations can be of the following types:

- Construct, Destroy
- Get State
- Modify State
- Compute data

An object provides methods, which can be called by other objects. Invoking a method is called sending a message to the object. In this way an object behaves like a server to its calling object.

Is a river an object?

Is it the same river? The water is not the same?
Yes, it is an object! It has a unique *identity*. It has *state*, although varying at different points along the river. Its attributes are name, width, length, depth, volume of water, direction of flow, geographical location, source (from where it starts), destination (where it merges into the sea) etc. It has *behavior*. Such as it flows, it drops, it meanders, it floods, it dries etc.

Interaction among objects for a purpose:

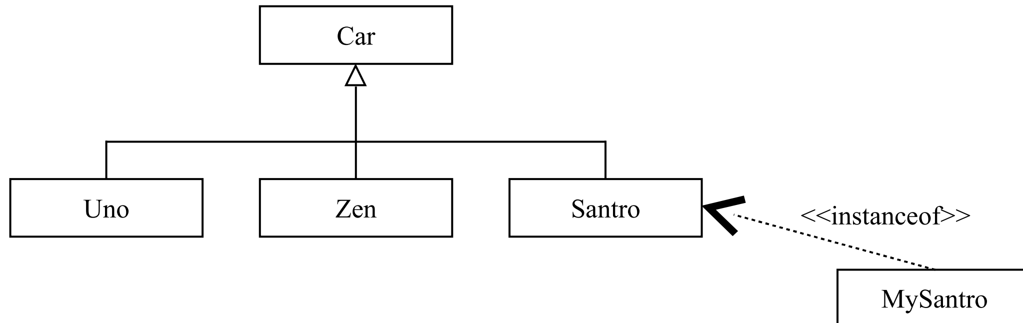
When objects interact, they do so with a purpose. Consider acceleration and deceleration in a car. When driver presses the brake, brake object sends a message to the wheel object to stop.
When user presses accelerator, carburetor object is asked to release more fuel into the combustion chamber.

1.3.2 Classes

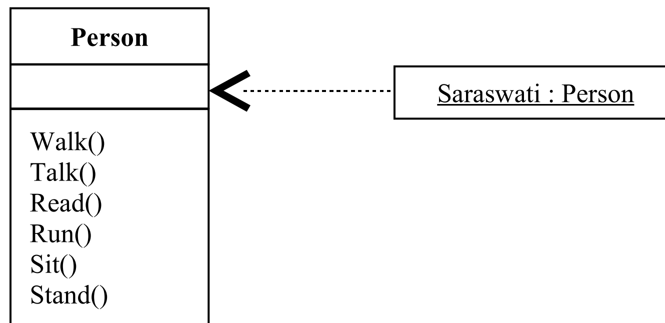
According to Booch "A class is a set of objects that share a common structure and a common behavior." A Class is like a template; its definition provides an interface, which is a binding contract between its objects (instances), and other client objects. A class can be refined further into a subclass.

1.3.3 Class Vs Object

E.g. Car is a class and subclasses can be Uno, Zen, Santro etc.
Objects can be instances of these subclasses. Instance of Santro can be MySantro, which has registration number MH-1216. The object MySantro has state, behavior and a unique identity.



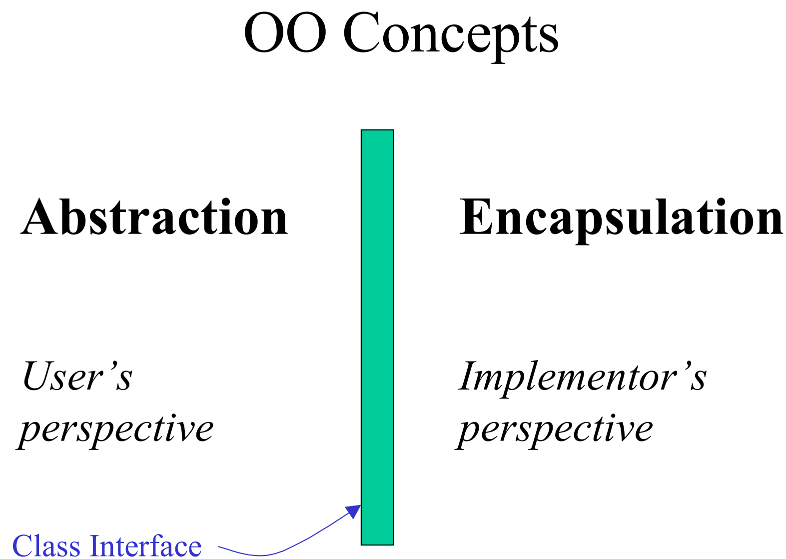
E.g. Person is a class and Saraswati is an object. Walk, talk, read, run, sit, stand are her methods.



1.4 Abstraction and Encapsulation

Abstraction deals with simplifying the complexity of an object, by focusing on its essential characteristics. Encapsulation is related to hiding details of an object, which are not relevant to other objects. Abstraction and encapsulation are like two sides of the same coin.
Abstraction focuses on the outside view of an object. What interface an object provides is the outside view.
Encapsulation focuses on the inside view of an object. What is the actual implementation is the inside view of an object.

Diagram shows the user view and implementer's view.



E.g. Consider a class 'Person' which has attribute 'DateofBirth'. It provides methods to set the date of birth and to get age of a person.

```
class Person {
        Date DateofBirth;
public:
        void  SetDateofBirth(Date d);
        int GetAge();
};
```

The method GetAge() can be changed from one implementation to another without the clients caring about it.

a)

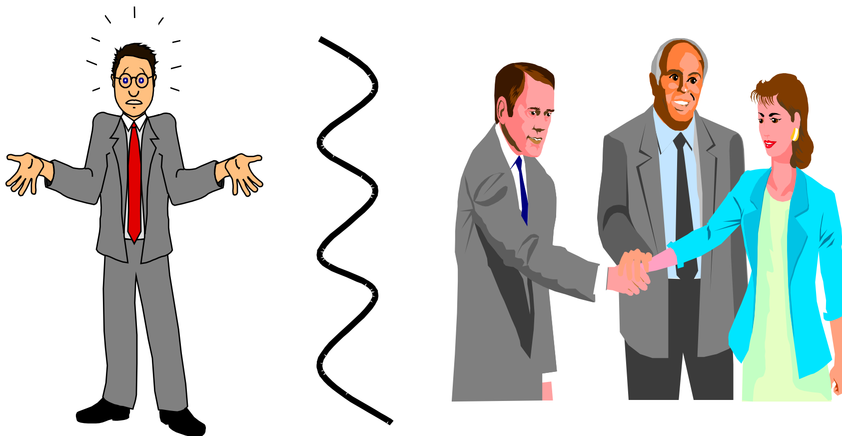
```
int Person::GetAge()
{
        return (TodayDate – DateofBirth);
}
```

OR:

b)
```
int Person::GetAge()
{
        return Age;     // In this case the implementation will require that 'Age' be stored as
an attribute.
}
```

The client does not know how GetAge() has been implemented. Even if implementation is changed, the client (i.e. another object which calls GetAge() method) is least affected. The client has an abstraction of the Person class. It only knows what facility, is provided by the Person class. The Person class encapsulates details about implementation, which are not exposed to the client.

1.5 Relationship among classes and objects



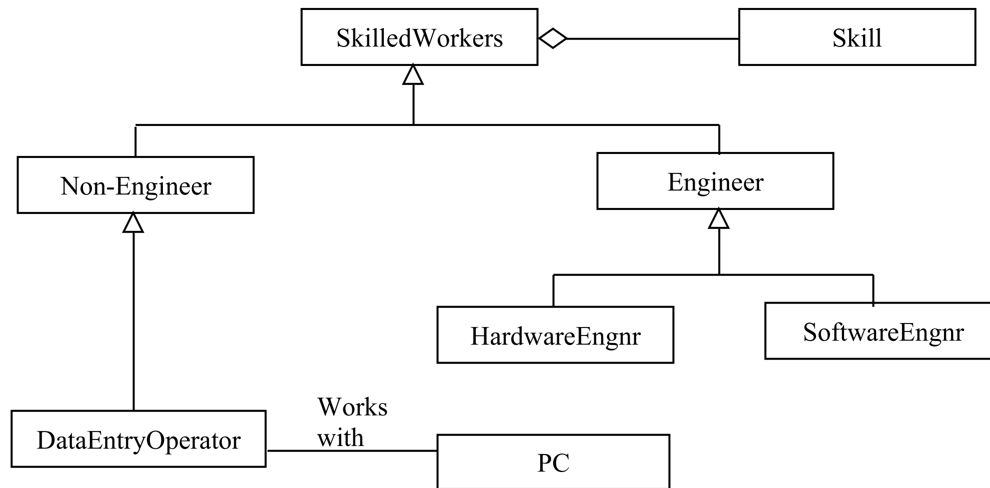
By themselves objects walk a lonely path. By working together in collaboration they contribute to the behavior of a system.

There are three types of relationships among classes:

- Inheritance - Kind of, Like a, Is like, Similar to
- Aggregation - Part of, Has a, Contains, Contained in (whole-part), Belongs to, Holds
- Association - Collaborates with, Is related to, Uses, Requires help from

e.g. A Data entry operator (DO), A hardware engineer(HE), A software engineer(SE), a Skill, a PC, are class of objects which have some similarities and differences.

Inheritance: DO, HE, SE are part of SkilledWorkers class, further subdivided into Non-Engineer and Engineer. Since a DO may not be an Engineer. A DO is not a kind of HE or SE. However they are all kind of skilled workers.



Advantage of inheritance implementation is reuse of code.
Discovering a good inheritance structure requires good classification.

What is classification?
Ordering items by recognizing sameness or likeness into a category. Identifying the sameness requires careful thought.

Aggregation:
Skill is part of each worker as shown above.

Containment by value:
In an aggregation relationship with physical containment, the contained object doesn't exist without its container object. When the container is created, it creates the contained object. Similarly when the container is destroyed, it destroys the contained object.

e.g. A PC contains a motherboard.

Containment by reference:
Implemented as a pointer. Here the creation and destruction of the container and containee are independent of each other.

e.g. A person owns real estates. The property he/she owns is not contained inside the person, however it belongs to him/her.

By aggregation with 'containment by value', a one way relationship is implied. The container contains the containee and not vice versa. In aggregation with 'containment by reference', the participating objects may hold references to each other.

The attribute (contained class) is a part of the state of the container, in an aggregation relationship between objects.

Inheritance-Aggregation difference:
Engine is not a kind of car, Uno is a kind of Car. Engine is part of a Car. Therefore an aggregation relationship exists between Engine and Car. And an inheritance relationship exists between Uno and Car.

Association: A PC and a data entry operator are not part of each other and they are not kind of each other. They collaborate to accomplish something. A data entry operator works with a PC. Association is a bi-directional relationship.

Association-Aggregation (by reference) difference:
A Person owns a Scooter, therefore the Scooter belongs to the person. There is a strong 'ownership' relationship here. In this case aggregation through 'containment by reference' is most appropriate.

If it was required to model the problem statement, which said, "A Person drives a Scooter to get somewhere." In this case, Person and Scooter are working together for a purpose. The purpose is to get somewhere. Therefore a strong association relationship is implied.

Using: Client-supplier relationships e.g. The speed control unit of a scooter, uses the accelerator handle unit for controlling speed of the scooter.

Typically this is not modeled as an association. It does not have a whole/part relationship, therefore it is not modeled as an aggregation. It has a strong "uses" relationship, which is implemented as follows:

> The client (speed control unit) object's methods receive the supplier (accelerator) object as a parameter or declare the supplier as a local variable.

Summary:
Confusion in the choice of applying "inheritance or aggregation" and "association or aggregation" can be avoided by asking specific questions and answering honestly.

Is every instance of a class like an instance of its parent and more? Then apply inheritance.
Is the object more than a sum of other parts? Then apply aggregation.
Is there a whole-part relationship? Then apply aggregation over association.
Is there a strong collaboration to achieve a purpose? Then apply association.

1.6 Visual Modeling

[A multimedia demonstration CD from Rational Corporation, containing short video clippings of the experts talk, explains the basics of Visual Modeling.]

Why Model?

- A model simplifies the complexity and helps in better understanding the system, because it is difficult to understand the entire system, all at once.

- A model helps convey and document thoughts and ideas, which can be understood by different people helping to build or maintain the system.

- A small visual model is much easier to understand compared to textual explanation.

About Modeling

- Use of appropriate model, modeling language is important. E.g. Architectural and structural diagrams for construction of buildings, physical models for study of molecular structures, mathematical models for wind tunnel tests or testing strength of materials.

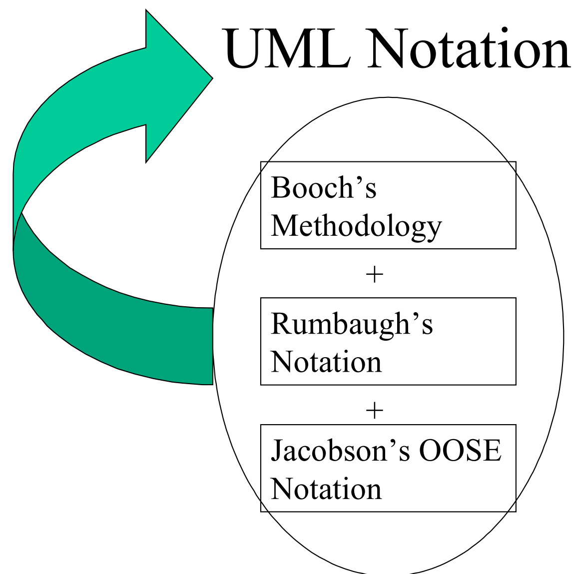
- Level of detail or granularity of the model should be at a sufficiently comprehensible level. Different viewers may require different degree of detail.

- Multiple models, which are nearly independent, may be required to understand the details of a system. E.g. A model representing the circulatory system of human body, another representing the respiratory system, another one for nervous system, yet another for understanding the musculo-skeletal system. Although each of these represents a different functionality, yet they can be superimposed or combined in parts to understand their interaction with other sub-systems.

OO Modeling
In the object-oriented perspective, basic building blocks are objects, derived from the vocabulary of the problem domain. Models require the ability to represent the static and dynamic behavior of relationships and interactions between such objects.

1.7 UML – Unified Modeling Language

The Unified Modeling Language (UML) is used for helping software analysts and designers to be able to visualize, document and construct object-oriented systems. Three leading advocates of object-oriented methodology, Grady Booch, James Rambaugh and Ivar Jacobson developed UML.



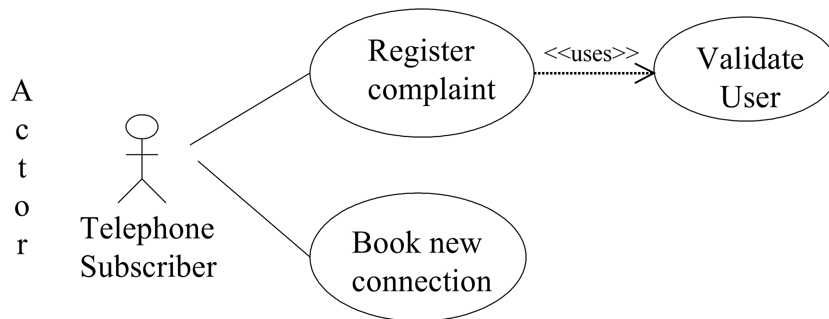
Following are the various modeling diagrams UML notation provides:
- Use Case diagram
- Class diagram
- Object diagram
- Interaction diagrams – Sequence diagram, Collaboration diagram
- Activity diagram
- Statechart diagram

- Component diagram
- Deployment diagram

Out of these, Use case diagram, Sequence diagram and Class diagram, are most often used. Sequence diagram and Statechart diagrams are dynamic diagrams, while Class diagram, Object diagram are static diagrams.

Use Case diagram
Is increasingly being used as a requirement gathering mechanism and for analysis of user requirements. A use case captures, some user-visible function, which may be at any chosen level of granularity. It achieves a distinct goal for the user. A use case diagram depicts functionality, which the user requires from the solution. Therefore whatever distinct functionality a user requires, is considered a use case.

UML - Use Case Diagram



Telephone Service Order System

Actors – Are shown by a human icon. They depict an entity external to the system, which will use the system, by providing inputs or requesting for output etc. An actor uses some functionality of the system.

Use case – Is shown by an oval. Represents functionality, which the actor invokes.

Flow of events – Textual description of flow of events associated with a use case (not shown in the diagram). Sample format is given below:

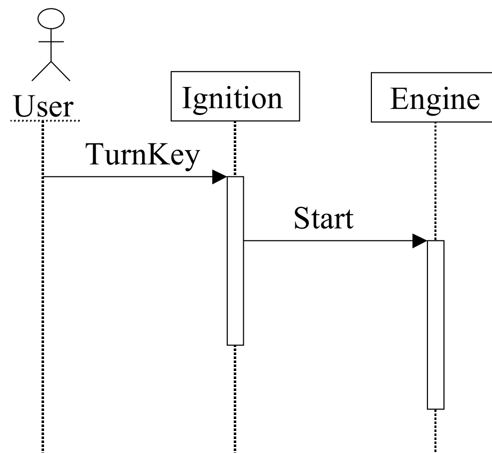
Flow of events in a use case
Use case: <Name of use case>
Actor: <External entity that invokes this use case>
Pre-conditions:
Main flow of events:
Sub flow of events:
Exceptional flow of events:
Post-conditions:

A use case can consist of more than one *scenario*.

What is a scenario?
Set of sequential steps, which form one logical combination or path through a use case. There could be several such combinations associated with one use case.
E.g. consider a use case "*dial telephone number*". One scenario could be where all goes well and the call is connected, one where there is no dial tone, one in which a wrong number gets connected, one in which the line is dead, one where the called number is engaged, etc.

Why use "use cases"?
Use cases help in capturing the requirements and documenting them. Any user can understand the simple visual diagrams. And the formatted documentation (flow of events), of each use case functionality, provides a good starting point for analysis and helps in identifying unclear requirements.

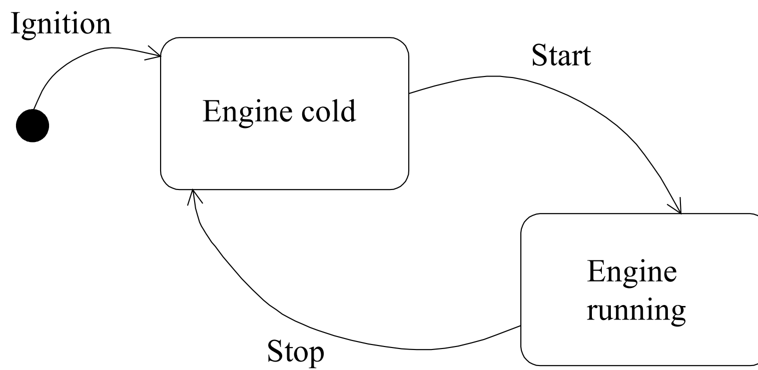
Interaction diagram: Sequence diagram
Is a dynamic diagram, used to depict sequence of events between classes to achieve a purpose.
e.g. A user starts the car by turning the ignition key.



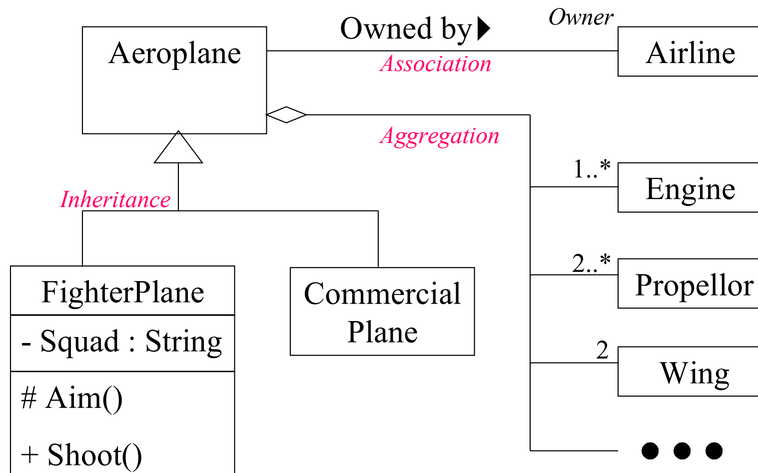
Statechart diagram
Is a dynamic diagram, to represent change of state of class due to events.

e.g. consider an Engine object. It can be in a stopped state or a running state.

UML - Statechart Diagram



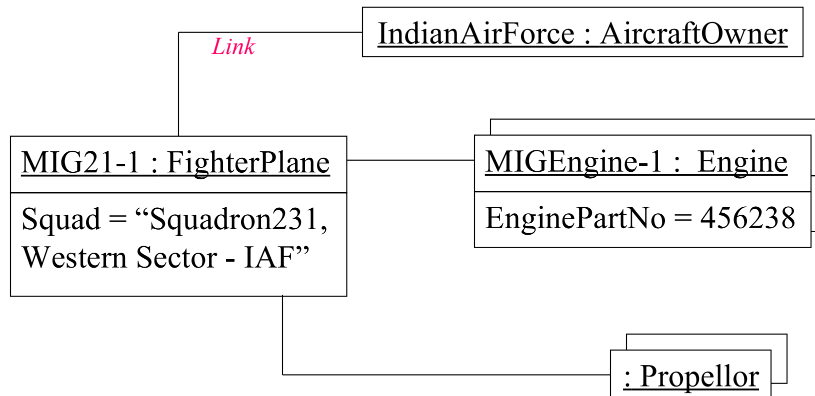
Class diagram
Is a static diagram, used to show the class structure of the application.



Association/Aggregation name: "Owned by" in the name given to the association between Aeroplane class and Airline class.
Rolename: "Owner" is a rolename i.e. the class Airline plays the role of owner in the association relationship with class Aeroplane.

Multiplicity: The one-to-one, one-to-many, many-to-many relationships are shown by numbers such as 1..*, 2..*, 2 etc.

Object diagram
Static diagram, used to show the object structure in individual scenarios of the application.



All the above diagrams evolve as we progress through Analysis and Design phases. They are not frozen until the end of design phase, i.e. after analysis phase, the diagrams do not become untouchable, they are reused during design phase, modified and enhanced.

Traceability is established due to connectivity among these diagrams. The classes that participate in a sequence diagram are detailed in a class diagram. These classes are drawn from the vocabulary of the use case statements, which is finally related to the requirements.

1.8 Tools

A tool, which provides, the required notation support is definitely best suited for analysis and design as compared to manual drawings. Most popular OOAD tool in the market today is "Rose" from Rational Corporation. It is commonly called Rational Rose.

1.9 Analysis and Design

Although there is no clear separation of analysis and design phases, there is normally an overlap. Analysis is study of the problem domain to dwell on *what* has to be done whereas design moves into the solution domain emphasizing *how* it has to be done.

Note: Designing and drawing a diagram are two different activities. The diagram is simply an aid to designing.

1.9.1 Object-Oriented Analysis

Identification of problem domain classes and objects and their interaction is the essence of analysis phase. The input to this phase is the requirements document or discussions with the clients, requestors, users etc. These inputs are expected in the form of details of functional/business operations, user interface, interface with other systems, performance, security, network distribution, data retention, portability, reporting and periodic processing etc.

During OO-Analysis, asking proper questions leads to better understanding of roles and responsibilities of objects.

- Who does it?
- What does it do?
- Whose responsibility is it really?

Identification of classes and objects:

The key here is proper classification. Classification deals with identification of common patterns among objects. There are different approaches to identify classes and objects from the vocabulary of the problem statement or requirements, e.g.

- Classical approaches - Tangible things, roles, events, interactions, people, places, concepts etc.
- Nouns and verbs approach
- …
- Use-case analysis

Use-Case Analysis:

The process of identifying actors, capturing the different ways (cases) an actor (external entity) will be using the system and determining the flow of events, both normal and abnormal, requires many items to be considered. It requires focused thought and specific questions. E.g. who is interacting with whom? Why (purpose)? How (operations)? Is it a distinct functionality, which the user needs?

Subsequent to use case analysis the following techniques flow smoothly in helping to identify and refine classes/objects:

- Scenario analysis
- CRC cards

Scenario Analysis:

The functionality of each use case is detailed in the flow of events. Each normal flow, sub flow and exceptional flow of events in the use case description forms a scenario. Using this description sequence diagrams are drawn to discover participating classes and interactions between them.

CRC cards:
This is an interesting technique, which requires participation of developers. CRC cards are useful when doing walkthrough of a use case.

An example of a CRC card is shown below.

ClassName: HRDDept	
Responsibilities	**Collaborations**
Compensation and Payroll	Accounts Department
Development of Employees	Training Department & Management Consultants
Recruitment	Interview Panels, Recruitment Agencies, Training Institutes, Advertising Agencies
Pursue Legal Matters	Accounts Department & Legal Consultants

Under *responsibilities*, column the major responsibilities of this class are written, which should be 2-3 in number. In case there are more, higher level responsibilities listed, the class may have to be split.

The results of CRC analysis can be documented using UML diagrams such as class, object and sequence diagrams.

Justification of classes:

Once the initial classes are identified, they need to be questioned to establish the basis of their existence. Such as:

- Why are objects of this class required?
- How are objects of this class created and destroyed?
- What operations can be done on objects of this class?
- Which functionality can objects of this class provide by interacting with other objects?

If answers to these questions are not appropriate, additional thought may be required to arrive at a better classification of the problem.

Identification of attributes and behavior:

Attributes:
Attributes hold the state of an object. They can be identified from the description of an object and what the object needs to know about itself, in order to handle its responsibilities. The actual implementation details such as, type of attribute, i.e. int, double, void pointer etc. should be postponed to the design phase.

Behavior:
The usual types of behavior in a class are:
- Construction and Destruction
- Get state and Set state
- Calculation

Along with other responsibilities identified for objects of the class.

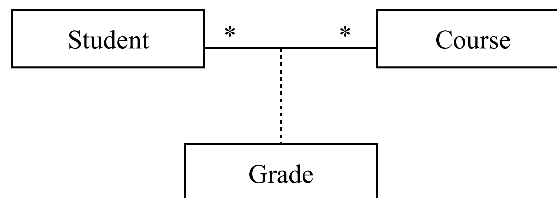
Identification of relationships and collaborations:

During analysis the dependencies such as inheritance, association, aggregation, between classes are identified and captured. However whether an association will be implemented as pointer, or double pointer, is an OO-Design decision.

Note: Any conditional behavior in terms of if-else, switch-case relationships or need to know the type of an object can be modeled into an inheritance structure.

Discovering Association Classes:

Consider two classes Student and Course. A student can take more than one course and a course can be attended by more than one student. However the marks obtained in a course by a student is a unique combination of an individual student and a particular course. The attribute marks does not belong to one class or other, but it belongs to a unique combination of objects of these classes. This is modeled using an association class. This class is a result of association between one or more classes. Thus marks will become an attribute of a new class e.g. Grade or Result.



1.9.2 Object-Oriented Design

Refining the analysis model, with the focus on providing details, for implementation of classes, is the core activity during design. Thought has to be clear and concentrated on the software implementation for given problem.

Fundamental of good object-oriented design is a good architecture in terms of its class and object structure through proper classification.

For a typical application the design activity can be categorized into three major areas:

- Designing the business component
- Designing the GUI component
- Designing the data management component

For each of the above areas or application layers the steps of design are similar.

Output from analysis phase such as class diagrams, sequence diagrams, statechart diagrams form the inputs to design phase. These are not thrown away, but the same diagrams can be refined and modified into the design diagrams.

- Scenario walkthrough – The first step in design is conducting a walkthrough of identified scenarios by using sequence diagrams already created during analysis.
- Inventing new classes – through refinement and detailed walkthrough of analysis level scenarios (sequence diagrams), i.e. adding implementation level classes.
- Class responsibility analysis – Based on responsibility, split large classes into smaller ones or combine smaller classes into one large class.
- Distribute or re-distribute correct responsibilities to each class - Only after the outside view of a class is finalized, does focusing on its internal view make sense.
- Correct Naming – class names, attribute names, method names, argument names for readability and maintainability.
- Finalize data type – attributes, method return type, argument data type.

- Interface detailing – public, protected, private.
- Establish Exception handling policy.
- Partition classes into modules.
- Deployment planning – which classes and modules to be packaged into which component, which component to reside on which machine etc.

Implementation of Aggregation and Association:
Aggregation Vs Association:
- Aggregation is better because it encapsulates the 'part' object as secret of the 'whole'.
- Association is better because it permits looser coupling between objects.

Association is usually implemented as a pointer from one to the other and vice-versa.

Aggregation by containment is usually implemented as class containment or a pointer, which is made to point to a new object, of the other class, created by the container class.

Aggregation by reference is implemented as a pointer where the other class is not necessarily created and destroyed by the container, however there is a whole-part relationship, which exists between them as long as they are related.

e.g.
```
class xyz
{
};

class abc
{
   xyz x1;
};        // Aggregation (containment by value)

class abc
{
   xyz *x1;
   abc() {    x1 = new xyz;   }
};        // Aggregation (containment by value)

class abc
{
   xyz *x1;
     Additem(xyz *ptr) {  x1 = ptr; }       // Object may be created independently and added
later.
};        // Aggregation (containment by reference)
```

Object Visibility
There are four different ways one object may be visible to another for the purpose of collaboration:

- The supplier object is global to the client object.
- The supplier object is a parameter to some operation of the client.
- The supplier object is a part of the client object.
- The supplier object is a locally declared object in some operation of the client object.

Decision about which of the above to use in a specific case is taken during designing.

Design patterns:

A set of well-documented patterns, which can be readily applied to similar problems are available for re-use. It is a good idea to look up the available design patterns before inventing new designs to save design time and be assured of a tested solution. A good source, for solution to commonly occurring design problems, is the book “Design Patterns” by Erich Gamma [et.al.]. There are many sites on the Web which have a treasure of information on other design patterns.

1.10 Modeling in the real world: an example
Imagine that a customer provides the following requirements.

Initial Requirements
He needs a vehicle (a transport mechanism),
Which can run automatically (without manual effort),
Which can seat 4 people at least,
It should have air conditioning,
And a stereo,
It should be possible to start it with the turn of a key.

Modeling
- Modeling the ignition system
- Modeling the electrical system
- Modeling the exhaust system
- Modeling the engine
- Modeling the braking system

1.11 Modeling in the real world: an example
Air-conditioning system in a company.

Modeling
Air-mixing unit (air mixed with cold water etc.)
Pumping mechanism,
Temperature controller mechanism

1.12 Modeling in the real world: an example
Help desk for a telephone complaint-registering system.

1.13 Why OOAD?
Finally, after having seen the actual process of OOAD here are a few benefits of using this method.

It has been found that focusing on algorithms and functions, during design of a software product, yields brittle systems, although nothing taboo about this traditional methodology. However as system requirements change or increase (through change requests or enhancement requests) such systems become hard to maintain.

In traditional design methodologies there is a basic disconnect between analysis and design. Design begins after analysis ends, especially where waterfall model of software development is used. In such case, concept and implementation may differ over time, making it difficult to maintain such systems.

The architecture of traditional non-object oriented systems results in tighter coupling between modules resulting in greater effort required when some change has to be made.

Object-Oriented system implementations are closer to their real world problem, making them easier to understand and modify. Changes have localized and smaller impact.

Risks in OOAD:
One of the major risks in object-oriented systems is performance. However the benefits are of adopting object-oriented technology are more paying than the concern caused by its risks.

1.14 Object-Oriented Metrics

Since the complexity in an object-oriented system is distributed in classes, the units of measurement should be related to a class rather than functions or lines of code. Following metrics are suggested by Chidamber and Kemerer to measure progress etc.:

- Weighted methods per class- gives relative measure of complexity
- Depth of inheritance tree - number of levels
- Number of children - number of sub-classes of a class
- Coupling between objects - measured by number of associations and method invocations from other classes
- Cohesion in methods - number of self method invocations

1.15 Books and References

Following is a list of good books related to Object-Oriented Analysis and Design:

Object-Oriented Analysis and Design
Grady Booch

Object-Oriented Analysis
Coad & Yourdon

Object-Oriented Design
Coad & Yourdon

Object-Oriented Software Construction
Bertrand Meyer

Object-Oriented Modeling and Design
James Rumbaugh, et.al.

UML User Guide
Grady Booch, Rumbaugh, Jacobson

Object-Oriented Software Engineering
Ivar Jacobson

Visual Modeling with Rational Rose and UML
Terry Quatrani

UML and C++
Richard Lee, William Tepfenhart

Analysis Patterns: Reusable Object Models
Martin Fowler

Design Patterns
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Objects, Components, and Frameworks with UML
Desmond F D'Souza, Analn C. Wells

Designing Object Applications using C++
Robert Martin

The Objectory Software Development Process

Grady Booch, Rumbaugh, Jacobson

Object Solutions: Managing the Object-Oriented Project
Grady Booch

1.16 Assignments

Following are a list of suggested assignments for this course. It is required that the participants analyze and design using object-oriented method by creating use-case diagrams, documenting flow of events, performing scenario analysis, creating sequence diagrams and class diagrams and submit their completed design for review (assessment).

- The HRD dept. requires a system to keep track of leaves taken by employees with various month-end and annual reports. Types of leaves taken by people, Seasonal leaves and generate trend reports (charts). Types of leaves by designation etc.

- The HRD dept. requires a system to analyze reasons for "floats" in a company. Seasonal floating tendency of the company.

- The training dept. of a company requires a system to plan, track and control training courses and provide detailed reports about courses conducted during specified period, how many and which courses were conducted by a faculty member, which courses were attended by an employee (for appraisal purposes), manage CSS responses etc.

- The Hardware dept. requires a system to keep track of physical location of PCs, IP addresses, Tel. Extensions, desk numbers and person/project to which PC is allocated, software installed on the PC, configuration of each PC.