OOP and UML

OOP

OOP is the abbreviation for Object orientated programming. OOP is a paradigm which defines the way a developer structures software into objects. Because such objects often correlate to real existing articles or procedures, it is often easier to read and less complicated to manage

A type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another

A good example for such an object is a wheel. There are certain characteristics that every wheel has: it is round and it turns. A wheel also has attributes that are different from wheel to wheel; for example the material it is made of or the diameter of the wheel. In the world of programming, this concept of heredity saves much time and effort, and helps to significantly reduce the initial costs for your software.

The Five concepts of oop are:

Objects, Class, Encapsulation, Inheritance, Polymorphism.

1. Objects:- An Object is a computer representation of some real-world thing (i.e person, place) or event. Objects can have both attributes and behaviours

2. Class:- Objects with the same data structure (Attributes) and behavior (Methods or Operations) are grouped together and it is called a class .

3. Encapsulation is a principle, about hiding the details of the implementation of the interface. It is to reveal as little as possible about the inner workings of the Interface.

4. Inheritance :- As objects do not exist by themselves but are instances of a CLASS, a class can inherit the features of another class and add its own modifications. (This could mean restrictions or additions to its functionality). Inheritance aids in the reuse of code.

5. Polymorphism means the ability to request that the same Operations be performed by a wide range of different types of things.

Object Oriented Programming Design:-

Design is a very important aspect of programming, to get your programs right the first time. Designs help determine what functionality is needed and how they function. It also breaks down the building of program so that sections can be built at a time according to a pre determined schedule.

Design in Object Orientation is a very big area, and UML itself deserves a tutorial on its own right. In this tutorial we shall only cover class diagrams within UML. and show you how to represent class diagrams in Graphical notation form, which is extending the diagraming technique from UML notation.

UML

UML (Unified Modelling Language) is a graphic language to depict processes and objects. With this, one could illustrate the relationship between our general and special wheel. In software development, UML is utilized to create the logical general view of software.

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing object oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

The primary goals in the design of the UML were:

1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.

2. Provide extensibility and specialization mechanisms to extend the core concepts.

3. Be independent of particular programming languages and development processes.

4. Provide a formal basis for understanding the modeling language.

5. Encourage the growth of the OO tools market.

6. Support higher-level development concepts such as collaborations, frameworks, patterns and components.

7. Integrate best practices.

As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, they recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing and fault tolerance. Additionally, the development for the World Wide Web, while making some things simpler, has exacerbated these architectural problems. The Unified Modeling Language (UML) was designed to respond to these needs.

The development of UML began in late 1994 when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. In the Fall of 1995, Ivar Jacobson and his Objectory company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method.

As the primary authors of the Booch, OMT, and OOSE methods, Grady Booch, Jim Rumbaugh, and Ivar Jacobson were motivated to create a unified modeling language for three reasons.

1. These methods were already evolving toward each other independently. It made sense to continue that evolution together rather than apart, eliminating the potential for any unnecessary and gratuitous differences that would further confuse users.

2. By unifying the semantics and notation, they could bring some stability to the object-oriented marketplace, allowing projects to settle on one mature modeling language and letting tool builders focus on delivering more useful features.

3. They expected that their collaboration would yield improvements in all three earlier methods, helping them to capture lessons learned and to address problems that none of their methods previously handled well.

The efforts of Booch, Rumbaugh, and Jacobson resulted in the release of the UML 0.9 and 0.91 documents in June and October of 1996. During 1996, the UML authors invited and received feedback from the general community. They incorporated this feedback, but it was clear that additional focused attention was still required.

While Rational was bringing UML together, efforts were being made on achieving the broader goal of an industry standard modeling language. In early 1995, Ivar Jacobson (then Chief Technology Officer of Objectory) and Richard Soley (then Chief Technology Officer of OMG) decided to push harder to achieve standardization in the methods marketplace. In June 1995, an OMG-hosted meeting of all major methodologists (or their representatives) resulted in the first worldwide agreement to seek methodology standards, under the aegis of the OMG process

During 1996, it became clear that several organizations saw UML as strategic to their business. A Request for Proposal (RFP) issued by the Object Management Group (OMG) provided the catalyst for these organizations to join forces around producing a joint RFP response. Rational established the UML Partners consortium with several organizations willing to dedicate resources to work toward a strong UML 1.0 definition. Those contributing most to the UML 1.0 definition included: Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, and Unisys. This collaboration produced UML 1.0, a modeling language that was well defined, expressive, powerful, and generally applicable. This was submitted to the OMG in January 1997 as an initial RFP response.

UML Diagrams

Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction. UML diagrams commonly created in visual modeling tools include:

Use Case Diagram displays the relationship among actors and use cases.

Class Diagram models class structure and contents using design elements such as classes, packages and objects. It also displays relationships such as containment, inheritance, associations and others.

Interaction Diagrams
• Sequence Diagram displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension and horizontal dimension .
• Collaboration Diagram displays an interaction organized around the objects and their links to one another. Numbers are used to show the sequence of messages.

State Diagram displays the sequences of states that an object of an interaction goes through during its life in response to received stimuli, together with its responses and actions.

Activity Diagram displays a special state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. This diagram focuses on flows driven by internal processing.

Physical Diagrams
• Component Diagram displays the high level packaged structure of the code itself. Dependencies among components are shown, including source code components, binary code components, and executable components. Some components exist at compile time, at link time, at run times well as at more than one time
• Deployment Diagram displays the configuration of run-time processing elements and the software components, processes, and objects that live on them.

Software component instances represent run-time manifestations of code units.

Lets start of by looking at a class diagram, The first diagram is off PERSON Class.

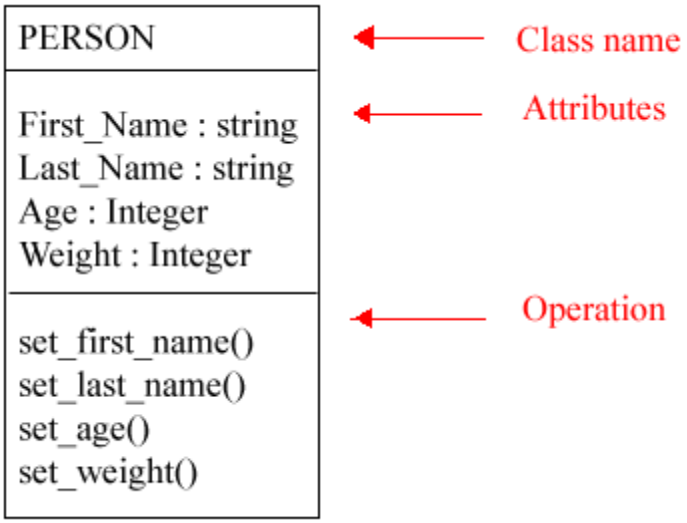
The data structure called Person has the following attributes.

1. First name
2. Last name
3. Age
4. Weight

with the following methods or operations for The Person object:-

1. set_first_name
2. get_first_name
3. set_last_name
4. get_middle_name etc..

To draw this in UML Notation, look at the diagram shown below.



A rectangle box is drawn, which is partitioned into three sections.
The top section is were you put the name of the class.
The middle section is for the attributes.
The bottom section you write the operations (which are going to be used to change the state of the attributes)

Aggregation

Aggregation differs from ordinary composition in that it does not imply ownership. In composition, when the owning object is destroyed, so are the contained objects. In aggregation, this is not necessarily true. For example, a university owns various departments (e.g., chemistry), and each department has a number of professors. If the university closes, the departments will no longer exist, but the professors in those departments will continue to exist. Therefore, a University can be seen as a composition of departments, whereas departments have an aggregation of professors. In addition, a Professor could work in more than one department, but a department could not be part of more than one university.

Composition is usually implemented such that an object contains another object.

For example, in C++:

Code: CPP

```
class Department;

class University
{
  ...
```

```
  private:
  Department faculty[20];
  ...
};


In aggregation, the object may only contain a reference or pointer to the
object:
class Professor;

class Department
{
  ...
  private:
  Professor* members[5];
  ...
};
```

Sometimes aggregation is referred to as composition when the distinction between ordinary composition and aggregation is unimportant.

Containment

Composition that is used to store several instances of the composited data type is referred to as containment. Examples of such containers are arrays, linked lists, binary trees and ociative arrays

In UML, containment is depicted with a multiplicity of 1 or 0..n (depending on the issue of ownership), indicating that the data type is composed of an unknown amount of instances of the composited data type.

In UML, composition is depicted as a filled diamond and a solid line. Aggregation is depicted as an open diamond and a solid line. The below image shows compositions, and then aggregation. The code below shows what the source code is likely to look like.

Code:

```
class Car
{
  public:
    virtual ~Car() {delete itsCarb;}
  private:
    Carburetor* itsCarb
};
class Pond
{
  private:
    vector<Duck*> itsDucks;
```

```
};
```

In this way OOP and UML together can make easier to understand Object Oriented Programming .

Source: http://www.go4expert.com/articles/oop-and-uml-t3007/