



**INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH  
TECHNOLOGY**

**A SURVEY ON THE CONCEPT OF OBJECT ORIENTED TECHNOLOGY**  
**Aishwarya Vyas\*, Bandana Mahapatra, Anshul chhabra, Astha Paliwal, Aarti Suryavanshi**

---

**ABSTRACT**

Classes are a boon to computer programming. When classes were not there the code was not secure since all the functions can access all the information. The debugging of code was difficult as the program followed top-down approach and the program was not divided into modules. So, error detection was a difficult task. Another drawback without classes was lack of reusability of code. If we had to use a code more than once, then we had to rewrite the whole code. This was a monotonous task. With the introduction of classes, data hiding was improved by using access specifiers, data reusability was introduced by using the concept of inheritance and debugging was easy as the program was divided into modules and followed a bottom-up approach.

**KEYWORDS:** Boject Oriented Technology.

---

**INTRODUCTION**

**Relationships:-**

Relationships represent logical links between two or more entities. For example: Residence is a relationship that can exist between the entities city and employee.

Instance of a relationship:-

It is an n-tuple made up of instances of entities, one for each of the entities involved. The pair of objects made up of the employee named John and The city London, or the pair of objects made from the employee Peter and the city New York, are examples of instances in the relationship Residence.

Types of relationships:-

Relationships can be classified as under:

**1. Association**

Association represents a relationship between two objects that is; association defines the multiplicity between objects. You may be aware of the terms one-to-one, one-to-many, many-to-one, many-to-many. All these words define an association between objects.

**2. Aggregation**

Aggregation is a special type of association. It is a directional association between objects. When an object 'has-a' another object, then you have got a case of aggregation between them. The direction between them specifies which object contains the other object. Aggregation is also called a "Has-a" relationship.

**3. Composition**

Composition is a special type of aggregation. More specifically, a restricted aggregation is

called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is known as composition.

**4. Dependency**

If the change in structure or behavior of a class affects the other related class, then there exists a dependency between those two classes. It need not be true for vice-versa. When one class contains the other class in it then this happens.

**5. Generalization**

Generalization works on a "is-a" relationship from a specialization to the generalization class. Common structure and behavior are used from the specialized to the generalized class. In a broader sense you can understand this as inheritance. Generalization is also known as "Is-a" relationship.

**6. Realization**

It is a relationship between the blueprint class and the object containing its respective implementation details. The object is said to realize the blueprint class. In a broader sense, you can understand this as the relationship between the interface and the implementing class. The above are the non-functional parameters over which different types of relationships are compared with one another. These parameters help us in choosing the best suited relationship for Finding out the solution to a given problem.

**Reusability:**

Reusability means being able to create a new class that uses the features of an existing class without recoding those features. Inheritance means reusing code in a hierarchical structure. For instance, a Basic Programmer is a Programmer is a Worker is a Person is an Animal. All Animals have heads, and therefore the Head property of a Basic Programmer should inherit all the general features of Animal heads plus all the features of Person heads plus all the features of Worker heads plus all the features of Programmer heads. When creating a Head property for a Basic Programmer object, you should need to write only the head code unique to Basic Programmers.

**Complexity:**

By complexity we mean the time and space required for solving a computational problem.

The time the computer requires for solving a given problem and the space required for the same.

**Efficiency:**

Code efficiency is a broad term used to depict the reliability, speed and programming methodology used in developing codes for an application. Code efficiency is linked with algorithmic efficiency and the speed of runtime execution for software. It is the key element in ensuring high performance. The objective of code efficiency is to reduce resource consumption and completion time as much as possible with minimum risk to the business or operating environment. The software product quality can be accessed and evaluated with the help of the efficiency of the code used.

**Maintainability:**

To a developer, maintainable code simply means “code that is easy to modify or extend”. At the heart of maintainability is carefully constructed code that is easy to read; code that is easy to dissect in order to locate the particular component relating to a given change request; code that is then easy to modify without the risk of starting a chain reaction of breakages in dependent modules.

**Understandability:**

A code that is a robust, quick and optimized code while it is structured enough to be readable by you and others later now understanding the relationships in brief:

**UML VIEW****ASSOCIATION:**

For example, a Customer class has a single association (1) to an Account class, indicating that each Account instance is owned by one Customer instance. If you have an account, you can locate the owning customer of that account, and for a given customer, you can find the account of that customer. The association between the Customer class and the Account class is important because it shows the structure between the two classifiers.

Multiplicity information can be linked to an association to show how many instances of class are linked with instances of class B. Multiplicity information can be linked to both ends of association relationships. In class diagrams, association relationships in a C/C++ application represent the following things:

- A semantic relationship between two or more classes that specifies connections among their instances,
- A structural relationship that specifies that objects of one class are connected to objects of a second, possibly the same, class.

In visualization mapping, instance variables in a C/C++ application become attributes in classifiers in class diagrams. By default, all C/C++ fields are shown as attributes.

An association relationship connector appears as a solid line between two classifiers.

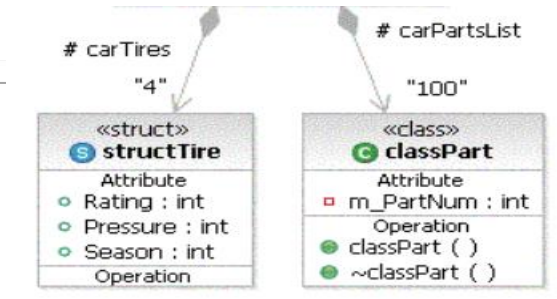
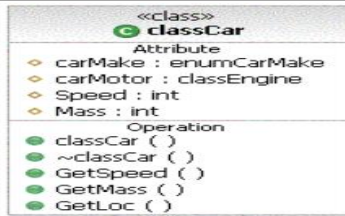
The following illustration displays a source code example and a graphical representation of an Association Relationship.

**C/C++ source code**

```

class classCar {
    protected:
        enumCarMake carMake;
        structTire carTires[4];
        classEngine carMotor;
        classPart carPartsList[100];
        int Speed;
        int Mass;
    public:
        classCar ();
        virtual ~classCar ();
        int GetSpeed ();
        int GetMass ();
};
    
```

**UML visualization**



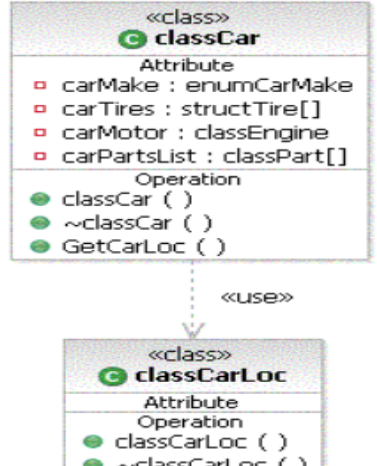
**DEPENDENCY:**

For example, a Cart class depends on a Product class because the Product class is used as a parameter for an add operation in the Cart class. In a class diagram, a dependency relationship points from the Cart class to the Product class. In other words, the Cart class is the consumer element, and the Product class is the supplier element. A change to the Product class may cause a change to the Cart class.

In class diagrams, dependency relationships in a C/C++ application connect two classes to indicate that there is a connection between the two classes, and that the connection is more temporary than an association relationship. A dependency relationship indicates that the consumer class does one of the following things:

- Temporarily uses a supplier class that has global scope,
- Temporarily uses a supplier class as a parameter for one of its operations,
- Temporarily uses a supplier class as a local variable for one of its operations,
- Sends a message to a supplier class.

As the figures in the following table illustrate, a dependency relationship connector appears as a dashed line with an open arrow that points from the consumer class to the supplier class. A dependency relationship means an "import" statement.

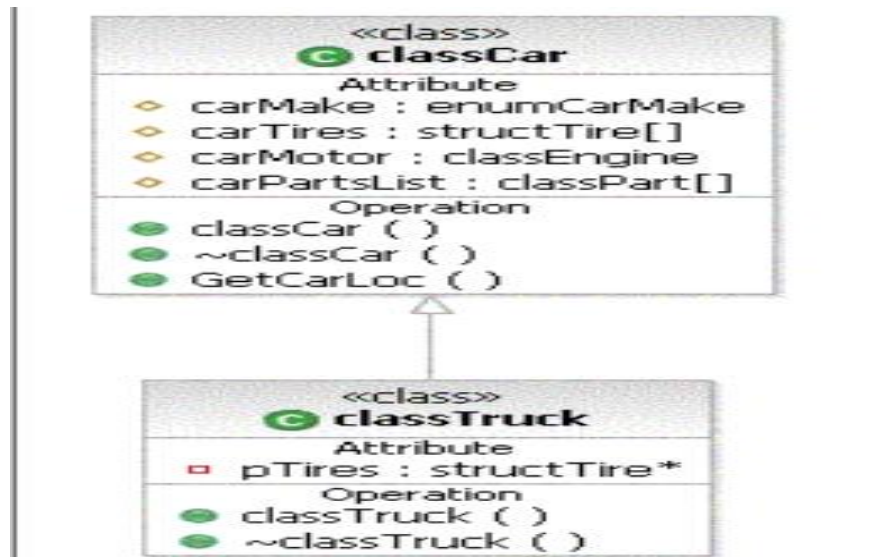
C/C++ source code	UML visualization
<pre> class classCar{     enumCarMake carMake;     structTire carTires[4];     classEngine carMotor;     classPart carPartsList[100]; public:     classCar();     virtual ~classCar();     void GetCarLoc(classCarLoc&amp; carLoc); };                     </pre>	

**GENERALISATION:**

In C/C++ domain modeling class diagrams, a generalization relationship, which is also called an inheritance or "an A is a B" (a human is a mammal, a mammal is an animal) relationship, implies that a specialized, child class is based on a general, parent class.

As the figure in the following table illustrates, a generalization relationship connector appears as a solid line with an unfilled arrowhead pointing from the specialized, child C/C++ class to the general, parent class. You can also visualize and design inheritance relationships between C/C++ classes.

C/C++ source code
<pre> class classCar{     protected:         enumCarMake carMake;         structTire carTires[4];         classEngine carMotor;         classPart carPartsList[100];     public:         classCar();         virtual ~classCar();         void GetCarLoc(classCarLoc&amp; carLoc); };  class classTruck : public classCar{     structTire* pTires;     public:         classTruck();         virtual ~classTruck(); };                     </pre>
<p><b>UML visualization</b></p> <hr/>



## CONCLUSION

In this paper, we have seen the different view to represent the relationship between the classes and their UML view. This paper is giving a point through which, we can come to know that how we build the relationship between the classes. How the objects of different classes will interact to one another. In the review paper, we will show how these relationships play an important role in the field of development.

## REFERENCES

1. S. Barbey, M. Ammann, and A. Strohmeier. Open issues in testing Object Oriented software. In K. F. (Ed.), editor, ECSQ '94 (European Conference on Software Quality), pages 257–267, vdf Hochschulverlag AG an der ETH Zürich, Basel, Switzerland, October 1994. Also available as Technical Report (EPFLDI-LGL No 94/45).
2. G. Booch. Object Oriented Design. Benjamin/Cummings Publ., USA, 1991.
3. G. Booch, I. Jacobson, and J. Rumbaugh. Unified Modeling Language User Guide. Addison-Wesley, 1997.
4. T. J. Cheatham and L. Mellinger. Testing Object-Oriented Software Systems. In Proceedings of the Eighteenth Annual Computer Science Conference, pages 161–165. ACM, Feb. 1990.
5. P. Coad and E. Yourdon. Object-Oriented Analysis. Prentice Hall, London, 2 edition, 1991.
6. A. Coen-Provini, L. Lavazza, and R. Zicari. Assuring type safety of objectoriented

languages. Journal of Object-Oriented Programming, 5(9):25–30, February 1994.

7. D. Coleman, F. Hyes, and S. Bear. Introducing objectcharts or how to use statecharts in object-oriented design. IEEE Transactions on Software Engineering, 18(1):9–18, January 1992.
8. W. Cook. A proposal for making eiffel type-safe. The Computer Journal, 32(4), 1989.
9. R. Doong and P. Frankl. The astoot approach to testing object-oriented programs. ACM Transactions on Software Engineering and Methodology, 3(2):101–130, April 1994.
10. R.-K. Doong and P. G. Frankl. Case Studies on Testing Object-Oriented Programs. In Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4), pages 165–177, Victoria, CDN, Oct. 1991. ACM SIGSOFT, acm press.