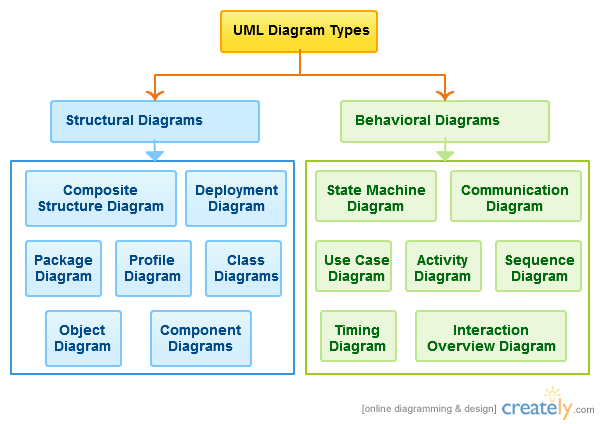
UML is just a graphical way of describing software systems. There are two types of UML namely:

1. **UML Sketching** - here you only want to communicate an idea and go through different alternative approaches. It's not specific but it only gives the basic ideas.
2. **UML Blueprint -** This is a very detailed type of modeling in which everything is well laid out and there is a definite time of completion.

<http://creately.com/blog/diagrams/uml-diagram-types-examples/>



Class diagrams are most used

### How to generate UML Diagrams from Java code in Eclipse - ObjectAid plugin

# [Design Documents (High Level and Low Level Design Documents)](http://stackoverflow.com/questions/10297869/design-documents-high-level-and-low-level-design-documents)

High level design(**HLD)** involves decomposing system into modules, and representating the interfaces and invocation relationships among modules. **A HLD is referred to as software architecture.**

LLD also known as detailed design is used to design internals of the individual modules identified during HLD i.e Data structure and algorithms of the modules are designed and documented.

Now, HLD and LLD are actually used in traditional Approach (Function-Oriented Software Design) whereas in OOAD, System is seen as a set of objects interacting with each other.

going by the above definitions, A high-level design document will usually include a high-level architecture diagram depicting the components, interfaces and networks that need to be further specified or developed. The document may also depict or otherwise refer to work flows and/or data flows between component systems.

**Class diagrams with all the methods and relation between classes comes under LLD**. Programs specs are covered under LLD.

LLD describes each and every module in an elaborate manner so that the programmer can directly code the program based on this. There will be at least 1 document for each module and there may be more for a module. The LLD will contain: - detailed functional logic of the module in pseudocode - database tables with all elements including their type and size - all interface details with complete API references(both requests and responses) - all dependency issues -error message listings - complete input and outputs for a module.

# UML basics: The class diagram

<https://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/>

## The basics

As mentioned earlier, the purpose of the class diagram is to show the types being modeled within the system. In most UML models these types include:

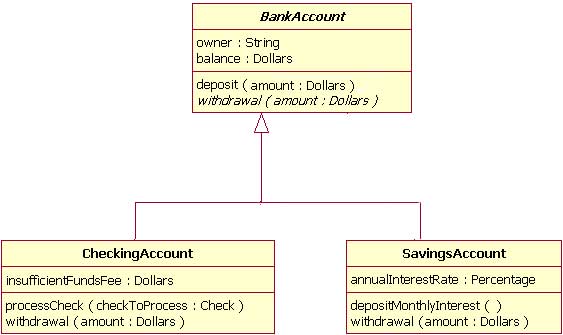
* a class
* an interface
* a data type
* a component.

### Class name

The UML representation of a class is a rectangle containing three compartments stacked vertically, as shown in Figure 1. The top compartment shows the class's name. The middle compartment lists the class's attributes. The bottom compartment lists the class's operations. When drawing a class element on a class diagram, you must use the top compartment, and the bottom two compartments are optional. (The bottom two would be unnecessary on a diagram depicting a higher level of detail in which the purpose is to show only the relationship between the classifiers.) Figure 1 shows an airline flight modeled as a UML class. As we can see, the name is Flight, and in the middle compartment we see that the Flight class has three attributes: flightNumber, departureTime, and flightDuration. In the bottom compartment we see that the Flight class has two operations: delayFlight and getArrivalTime.

##### Class diagram for the class FlightFigure 1: Class diagram for the class Flight

##### Figure 5: An example of inheritance using tree notation



### Abstract classes and operations

The observant reader will notice that the diagrams in Figures 4 and 5 use italicized text for the BankAccount class name and withdrawal operation. **This indicates that the BankAccount class is an abstract class and the withdrawal method is an abstract** operation. In other words, the BankAccount class provides the abstract operation signature of withdrawal and the two child classes of CheckingAccount and SavingsAccount each implement their own version of that operation.

**UML Building Blocks:**

As UML describes the real time systems it is very important to make a conceptual model and then proceed gradually. Conceptual model of UML can be mastered by learning the following three major elements:

 UML building blocks

 Rules to connect the building blocks

 Common mechanisms of UML

This chapter describes all the UML building blocks. The building blocks of UML can be defined as:

 Things

 Relationships

 Diagrams

<http://www.classdraw.com/help.htm>

# UML Class Diagram Help

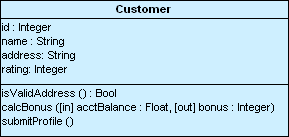
UML stands for *Unified Modeling Language* and is a visual modeling language, defined and maintained by the [Object Management Group](http://www.omg.org/). It can be used to create models of object-oriented software to help with design, although it can also be used to model other systems that have nothing to do with software, such as business models. UML specifies several diagrams for representing different aspects of a system, of which Class Diagrams are the most well known and widely used among software developers. Class diagrams represent the static elements of a system, rather than processes or snapshots of the system at a given time and describe the classes, the data and functionality they contain and the relationships between them.

Visual modeling techniques are important for large, complex systems, but they also play a very useful role in the development of smaller systems. The use of class diagrams can help to clarify ideas in the initial design stages of an object oriented system when analyzing the requirements and deciding on the fundamental classes and how they will relate to each other. The diagram also serves as a clear visual documentation of the overall design and can be adjusted as the system develops.

The following sections provide an overview of the main features of class diagrams to assist in using ClassDraw. Further resources are widely available which discuss various methodologies for designing a system using UML. You don't have to make use of all the features of class diagrams to benefit from them. It's up to you how much detail you provide to help with your design.

## The Class Icon

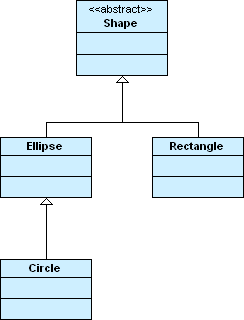
A class is represented visually using an *icon*, which is a rectangle split into three sections. The first section contains the class name, the second contains its *attributes* (member variables) and the third contains its *operations* (member functions). The class icon is intended to provide information that is useful in understanding the role of the class in the context of the rest of the diagram. It does not have to contain every attribute and operation of the class.



In UML notation, an attribute or operation ends with a colon, followed by its type. The colon and type can be omitted if the type is redundant. The type names should be chosen so that they are meaningful to those who will be looking at the diagram. Parameters can also be specified as input or output as shown. Some software developers prefer to specify attributes and operations in the more familiar format of a programming language instead of using the standard UML notation. That's fine, as long as it makes sense to everyone concerned.

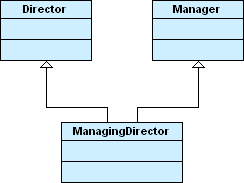
## Inheritance Relationships

The inheritance relationship, also known as the *generalization* relationship, is used to indicate that one class is a specialization of another. In the following example, the Ellipse and Rectangle classes are derived from the Shape class, because they both share characteristics that are common to all shapes in the system. The Shape class is the base class in both relationships. Inheritance relationships are shown in a hierarchy with the base classes at the top and the derived classes below them.



In the above example, Shape is also an *abstract class*. An abstract class is one for which an instance cannot be created, because it only makes sense to create an instance of a class derived from it. Abstract classes are represented by placing the "<<abstract>>" stereotype above the class name or by showing the class name in italics.

It is also possible for a class to inherit from multiple base classes, although some programming languages do not support multiple inheritance. In the example, a managing director is both a manager and a director.



## Associations

An association is a relationship between two classes represented by a solid line. Associations are bi-directional by **default**, so both classes know about each other and about the relationship between them. Either end of the line can have a *role name* and *multiplicity*. In the example, Student has the role of "tenant" in relation to Apartment and Apartment has the role of "accommodation" in relation to Student. Also, any instance of Apartment can be associated with up to four students and any student could be associated with 0 or 1 Apartment (a student either has an apartment to live in or does not).

Bi-directional Association

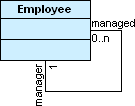
Associations can also be unidirectional, where one class knows about the other class and the relationship but the other class does not. Such associations require an open arrowhead to point to the class that is known and only the known class can have a role name and multiplicity. In the example, the Customer class knows about any number of products purchased but the Product class knows nothing about any customer. The multiplicity "0..\*" means zero or more.

Uni-directional Association

An alternative to using role names is to provide a single name for an association centered between the two classes. A direction indicator can also be used to show the direction of the name, but is not necessary if the direction is obvious:

Named Association

An association can also link a class to itself. Such an association is *reflexive*:



## Aggregation

Aggregation is a relationship where one class is part of another class. In basic aggregation, the class that forms part of the whole class can exist independently, so the life of an instance of the part class is not determined by the whole class. Basic aggregation is represented using an empty diamond symbol next to the whole class. In the example, a computer in a warehouse contains a motherboard, but although the motherboard is part of the computer, it can exist as a separate item. In this system, Computer knows about Motherboard but Motherboard doesn't know about Computer, so the aggregation is unidirectional. In a program, this relationship could be implemented as a member variable in the Computer class which is a reference to a Motherboard class.

Basic Aggregation

Association differs from aggregation only in that it does not imply any containment.

## Composition

Composition is a strong type of aggregation where the whole class contains the instance of the part class. The lifetime of the part class depends on the existence of the whole class. Composition relationships are represented using a filled diamond symbol next to the whole class. In the example, a building contains a number of rooms and a room cannot exist without a building. If a Building class instance is destroyed, its Room class instances are destroyed too. In a program, this relationship could be implemented as a member variable in the Building class which is an array of Room classes and the Room class might contain a pointer to a Building class.

Composition

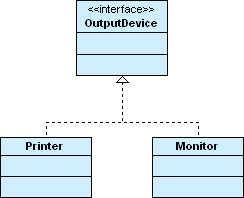
## Dependencies

A dependency is a weak relationship between two classes and is represented by a dotted line. In the example, there is a dependency between Point and LineSegment, because LineSegment's draw() operation uses the Point class. It indicates that LineSegment has to know about Point, even if it has no attributes of that type. This example also illustrates how class diagrams are used to focus in on what is important in the context, as you wouldn't normally want to show such detailed dependencies for all your class operations.

Dependency

## Interfaces

An interface is represented in the same way as a class, except that the <<interface>> stereotype appears above the name. You can never create an instance of an interface and it must be related to at least one other class that can implement it. An interface defines a set of things that an implementing class can *do* rather than a set of things that the implementing class *is*. The links between the interface and its implementing classes are dotted, but they appear in other respects like inheritance relationships between classes. Some programming languages such as C++ do not have the concept of an interface built in, but an abstract class with no member variables and only pure virtual functions could be used to represent one. In the example, the OutputDevice interface is implemented by the Printer and Monitor classes.

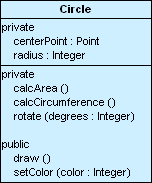


## Visibility

Visibility markers indicate if an attribute or operation in a class can be accessed only from within that class (private), from within the class and any derived classes (protected), from within the package that the class is part of (package) or from anywhere (public). Visibility markers are placed at the start of the relevant attribute or operation, for example: +setConnectionStatus().

|  |  |
| --- | --- |
| **Visibility** | **Symbol** |
| private | - |
| protected | # |
| package | ~ |
| public | + |

An alternative to using the above markers is to group attributes and operations by visibility:



# [What's the best UML diagramming tool? [closed]](http://stackoverflow.com/questions/15376/whats-the-best-uml-diagramming-tool)

Some context: Recently for graduate school I researched UML tools for usability and UML comprehension in general for an independent project. I also model/architect for a living.

The previous posts have too many answers and not enough questions. A common misunderstanding is that UML is about creating diagrams. Sure, diagrams are important, but really you are creating a model. Here are the questions that should be answered as each vendor product/solution does some things better than others. Note: The listed answers are my view as the best even if other products support a given feature or need.

* Are you modeling or drawing? (Drawing - [ArgoUML](http://argouml.tigris.org/), free implementations, and [Visio](http://office.microsoft.com/visio/))
* Will you be modeling in the future? (For basic modeling - Community editions of pay products)
* Do you want to formalize your modeling through profiles or meta-models? OCL? ([Sparx](http://www.sparxsystems.com/), RSM, [Visual Paradigm](http://www.visual-paradigm.com/product/vpuml/))
* Are you concerned about model portability, XMI support? ([GenMyModel](http://www.genmymodel.com/), [Sparx](http://www.sparxsystems.com/), [Visual Paradigm](http://www.visual-paradigm.com/product/vpuml/), [Altova](http://www.altova.com/umodel.html))
* Do you have an existing \set of documents that you need to work with? (Depends on the documents)
* Would you want to generate code stubs or full functioning code?([GenMyModel](http://www.genmymodel.com/), [Visual Paradigm](http://www.visual-paradigm.com/product/vpuml/), [Sparx](http://www.sparxsystems.com/),[Altova](http://www.altova.com/umodel.html))
* Do you need more mature processes such as use case management, pattern creation, asset creation, RUP integration, etc? (RSA/RSM/IBM Rational Products)

Detailed Examples: IBM Rational Software Architect did not implement UML 2.0 all the way when it comes to realizes type relationships when creating a UML profile, but Visual Paradigm and Sparx got it right.   
Ok, that was way too detailed, so a simpler example would be [ArgoUML](http://argouml.tigris.org/), which has no code generation features and focuses on drawing more than the modeling aspect of UML.   
[Sparx](http://www.sparxsystems.com/) and [Visual Paradigm](http://www.visual-paradigm.com/product/vpuml/) do UML really well and generate code well, however, hooking into project lifecycles and other process is where RSM/RSA is strong.   
Watch out for closed or product specific code generation processes or frameworks as you could end up stuck with that product.

This is a straight brain dump so a couple details may not be perfect, however, this should provide a general map to the questions and solutions to looking into.

NEW - Found a good list of many UML tools with descriptions. [Wiki UML Tool List](http://en.wikipedia.org/wiki/List_of_UML_tools)

Interview questions

<http://java-success.blogspot.com/2012/07/uml-diagrams-interview-questions-and.html>

<http://stackoverflow.com/questions/731802/what-is-the-difference-between-composition-and-association-relationship>

**COMPOSITION**

Imagine a software firm that is composed of different Business Units (or departments) like Storage BU, Networking BU. Automobile BU. The life time of these Business Units is governed by the lifetime of the organization. In other words, these Business Units cannot exist independently without the firm. This is COMPOSITION. (ie the firm is COMPOSED OF business units)

**ASSOCIATION**

The software firm may have external caterers serving food to the employees. These caterers are NOT PART OF the firm. However, they are ASSOCIATED with the firm. The caterers can exist even if our software firm is closed down. They may serve another firm! Thus the lifetime of caterers is not governed by the lifetime of the software firm. This is typical ASSOCIATION

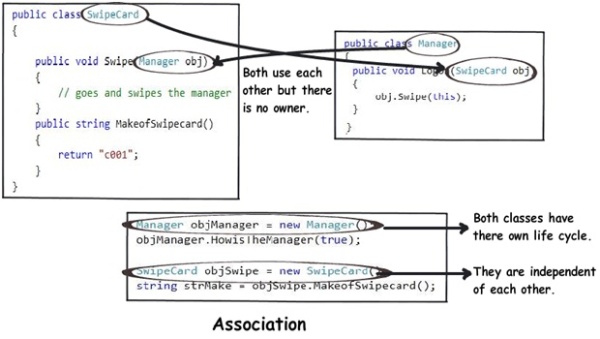
**AGGREGATION**

Consider a Car manufacturing unit. We can think of Car as a whole entity and Car Wheel as part of the Car. (at this point, it may look like composition..hold on) The wheel can be created weeks ahead of time, and it can sit in a warehouse before being placed on a car during assembly. In this example, the Wheel class's instance clearly lives independently of the Car class's instance. Thus, unlike composition, in aggregation, life cycles of the objects involved are not tightly coupled.

JAVA EXAMPLES:

## Requirement 2: The Using relationship: Association

Requirement 2 is an interesting requirement (Manager uses a swipe card to enter XYZ premises). In this requirement, the manager object and the swipe card object use each other but they have their own object life time. In other words, they can exist without each other. The most important point in this relationship is that there is no single owner.



The above diagram shows how the SwipeCard class uses the Manager class and the Manager class uses the SwipeCard class. You can also see how we can create objects of the Manager class and SwipeCard class independently and they can have their own object life time.

This relationship is called the “Association” relationship

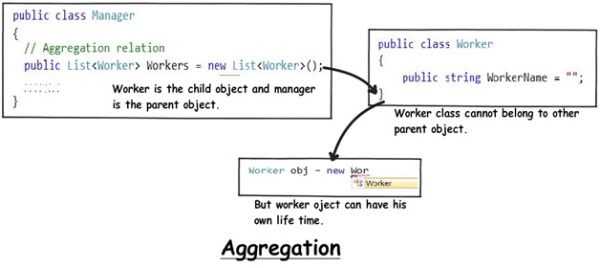
## Requirement 3: The Using relationship with Parent: Aggregation

The third requirement from our list (Manager has workers who work under him) denotes the same type of relationship like association but with a difference that one of them is an owner. So as per the requirement, theManager object will own Worker objects.

The child Worker objects can not belong to any other object. For instance, a Worker object cannot belong to a SwipeCard object.

But… the Worker object can have its own life time which is completely disconnected from the Manager object. Looking from a different perspective, it means that if the Manager object is deleted, the Worker object does not die.

This relationship is termed as an “Aggregation” relationship.



**Composition :** Since Engine is part-of Car, relationship between them is Composition. Here is how they are implemented between Java classes.

**public** **class** **Car** {

//final will make sure engine is initialized

**private** **final** Engine engine;

**public** **Car**(){

engine = **new** Engine();

}

}

**class** **Engine** {

**private** String type;

}

In Java, you can use [final keyword](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) to represent Composition. Since in Composition, Owner object expect part object to be available and functions, by making it final, your provide guarantee that, when Owner will be created, this part object will exist. This is actually a *Java idiom to represent strong form of association* i.e. composition between two objects.  
  
Read more: <http://javarevisited.blogspot.com/2014/02/ifference-between-association-vs-composition-vs-aggregation.html#ixzz3OZFEpnX4>

Read more: <http://javarevisited.blogspot.com/2014/02/ifference-between-association-vs-composition-vs-aggregation.html#ixzz3OZ1PhwzT>

<http://1.bp.blogspot.com/-VL_9cjhwEE4/UvJN__IvaBI/AAAAAAAABCc/IkDmShgM-Yc/s1600/Association,+Composition+UML.JPG>

