

---

**Username:** University of Illinois Chicago **Book:** UML for Mere Mortals®. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

---

## Logical Architecture

We just discussed the various scopes that an architecture can have (i.e., enterprise, system, and software architectures). Various levels of abstraction also exist for such architectures, each containing a different level of detail about the architecture. The two levels of abstraction we will discuss are logical architecture and physical architecture.

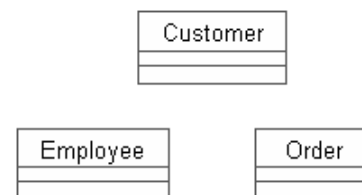
A logical architecture represents an architecture that is independent of the overall technology to be implemented. It is an interpretation of what that architecture should look like. It is not intended to show the implementation software, but rather an abstraction of it. In software, the logical architecture normally denotes a description of the software's architecture in plain language (nontechnical). It is not optimized for any specific technology (it is technology-independent). Architects, not typical developers, often create these logical architectures.

Contrasting the various logical architectures, the enterprise architecture shows how the organization works and the intended direction in which the company is going. The system architecture shows how the system to be developed will meet those business needs. And the software logical architecture depicts the structure of the software executing within that system.

## Class Diagrams

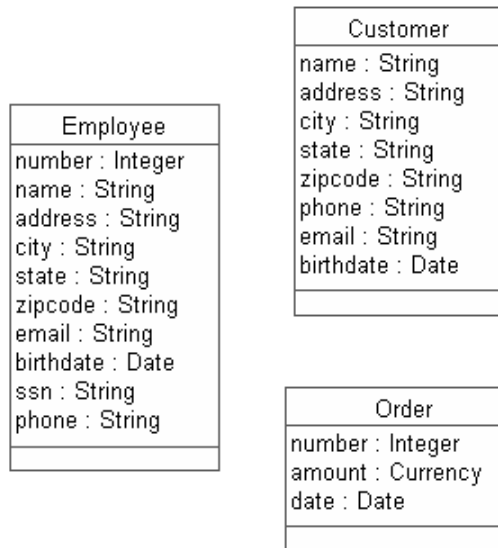
Using the UML, you design a logical architecture using classes in a class diagram. A class is visualized as a box containing two horizontal lines. Above the first line is the class's name, which is the logical description of the class and is typically a noun. Examples of a class name are Customer, Employee, and Order, as shown in [Figure 4-2](#).

**Figure 4-2. Classes on a class diagram.**



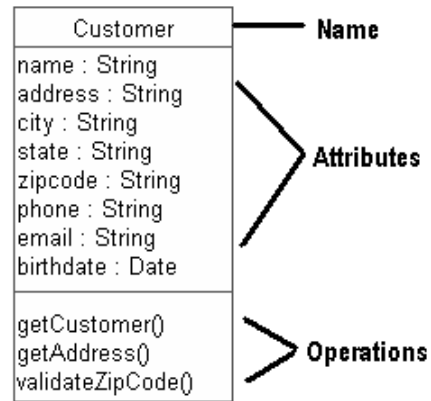
Attributes of a class are shown above the second line in the class and are used to describe the different properties of that class. They provide additional detail pertaining to that class. Attributes have additional properties attached to them that describe the type of data that we intend to capture within that attribute. Examples of some attribute types are Number, String (textual information), Boolean (yes or no), and Date. The type is displayed on a UML class to the right of the attribute name, as shown in [Figure 4-3](#).

**Figure 4-3. Classes with attributes.**









Operations, which you will see in figures later in this chapter (such as [Figure 4-11](#)), appear on a class below the second line and are used to indicate the behavior of a class. Operations define the logic that will execute in the system, both basic (e.g., returning an error) and sophisticated (e.g., a large algorithmic calculation). We will cover operations in more depth later in this chapter.

**Figure 4-11. Class with attributes and operations.**



You can use different types of associations in a class diagram to show relationships between classes or between classes and other modeling elements. [Figure 4-4](#) shows the different types of associations and gives a brief definition of each.

**Figure 4-4. Association descriptions.**

Name	Graphic	Description
Undirectional Association		A relationship between two model elements that is navigated in primarily a single direction
Bidirectional Association		A relationship between two model elements that is navigated in both directions
Dependency		A relationship between two model elements where a change to one may cause a change to the other
Aggregation		A relationship between two model elements indicating that one is part of the other
Composite Aggregation		An aggregation where both model elements are tightly coupled to the point that the child cannot exist without the parent
Generalization		A relationship between model elements indicating that one element (subclass) is a "kind of" another element (superclass)

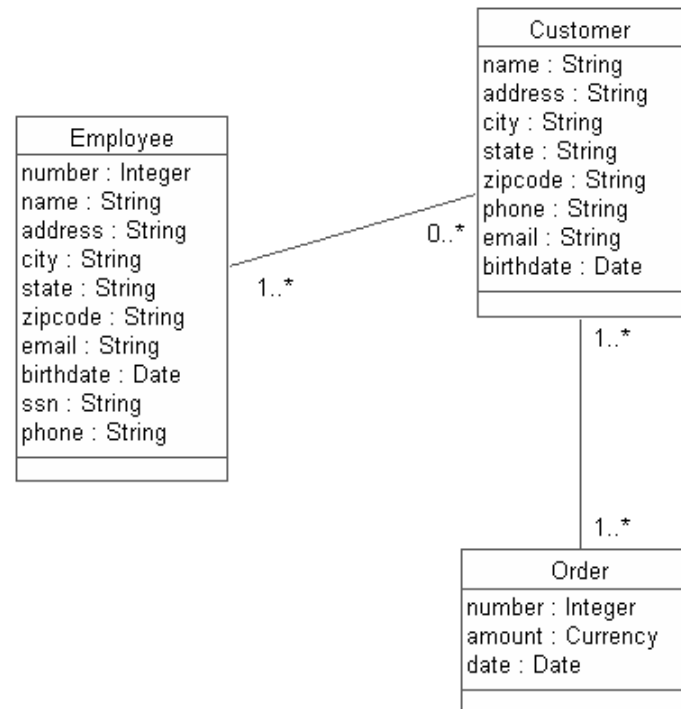
When defining an association, you can also designate each class's multiplicity. Multiplicity is used to show the number of objects that participate in the association. Multiplicity can be defined on an association and is shown on each end of the association. There are many combinations of multiplicity annotations. Here are some examples:

1	Exactly one
0..*	Zero or more
1..*	One or more
0..1	Zero or one
3..9	Specified range (3, 4, 5, 6, 7, 8, or 9)

Multiplicity can be defined as a specific number, as "many" (meaning an unlimited number can occur), or as a range, such

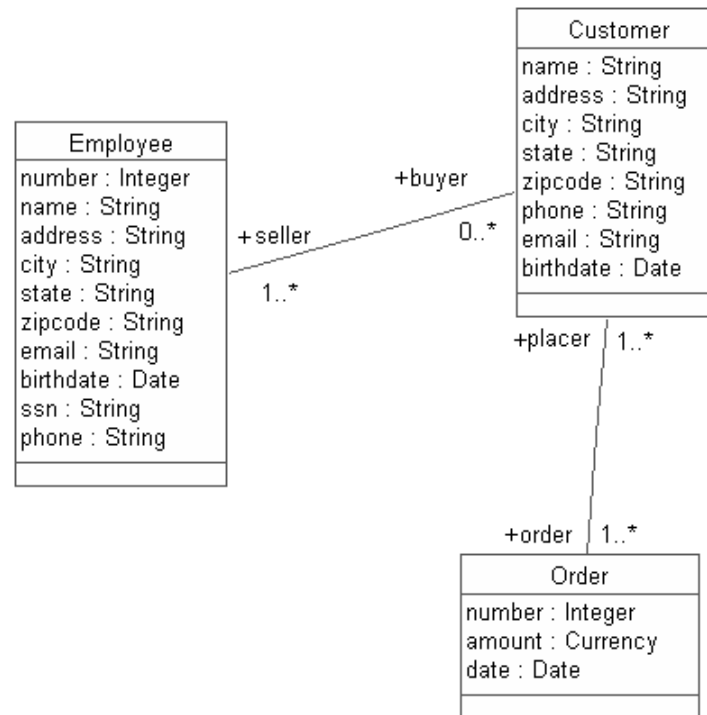
as 0 to 6. [Figure 4-5](#) shows the classes from [Figure 4-3](#) with associations and multiplicity added.

**Figure 4-5. Class diagram with associations.**



In [Figure 4-5](#), the association between Employee and Customer is read as “an Employee is associated with 0 or more Customers, and a Customer is associated with 1 or more Employees.” You can also add a role to each end of the association to better define it. In [Figure 4-6](#), roles are added to our example.

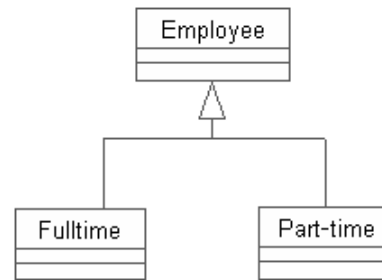
**Figure 4-6. Class diagram with association and roles.**



As you may notice, there is a plus sign (+) before the rolename. No, that isn't a typo; it does belong there. The plus sign defines the role as public when used in the definition of code. A role can be public (+), protected (#), private (-), or implementation where no adornment exists. Public means that the role is accessible to all elements that need to access it, protected is only available to those elements that are part of the relation and elements it is related to, private is only available to that association, and implementation means it is only available to the overall package that implements that role.

When creating a logical architecture model, generalizations are used to show how classes have a hierarchy of definition. For example, an employee can be either fulltime or part-time. When defining an Employee class, you might want to provide additional attributes to capture these different types of employees. But, an Employee class might be more clearly understood by nontechnical reviewers as being various types of employees, as seen in [Figure 4-7](#). This indicates that both fulltime and part-time employees inherit all the attributes and operations of the parent Employee class.

**Figure 4-7. Generalization.**

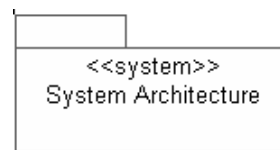


## Systems and Subsystems

In this section, we will introduce a new UML construct called a *stereotype*. A stereotype enables you to extend the UML to fit your modeling needs more specifically. A stereotype is a UML modeling element that extends the existing elements. Stereotyping a UML element causes it to act as something else that has specific properties. A stereotype is represented as `<<stereotype>>` on the element being stereotyped.

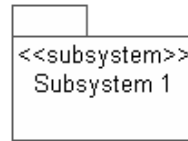
A system is represented as a package with the stereotype of `<<system>>`, as seen in [Figure 4-8](#). The system represents all the model elements that pertain to the particular project. You can also break a system into `<<business systems>>` and `<<application systems>>` when building more detailed models to make them smaller and more workable.

**Figure 4-8. System.**



A system usually is broken into multiple subsystems. Subsystems, like systems, are stereotyped packages with the stereotype of `<<subsystem>>`, as you can see in [Figure 4-9](#). A subsystem is a grouping of model elements that are part of the overall system.

**Figure 4-9. Subsystem.**



Because a system or subsystem is a stereotyped package, it has all the properties and rules of a package. This means that model elements that are contained by the system and subsystem are owned by that package and can only be part of them and no other. The subsystem gives the project team an easy way to partition the system. Since a system contains multiple subsystems, everything contained within the subsystems is owned by the system that they roll up into. [SCOTT1] A diagram can display the logical architecture of a system (see [Figure 4-10](#)).

**Figure 4-10. System architecture.**

