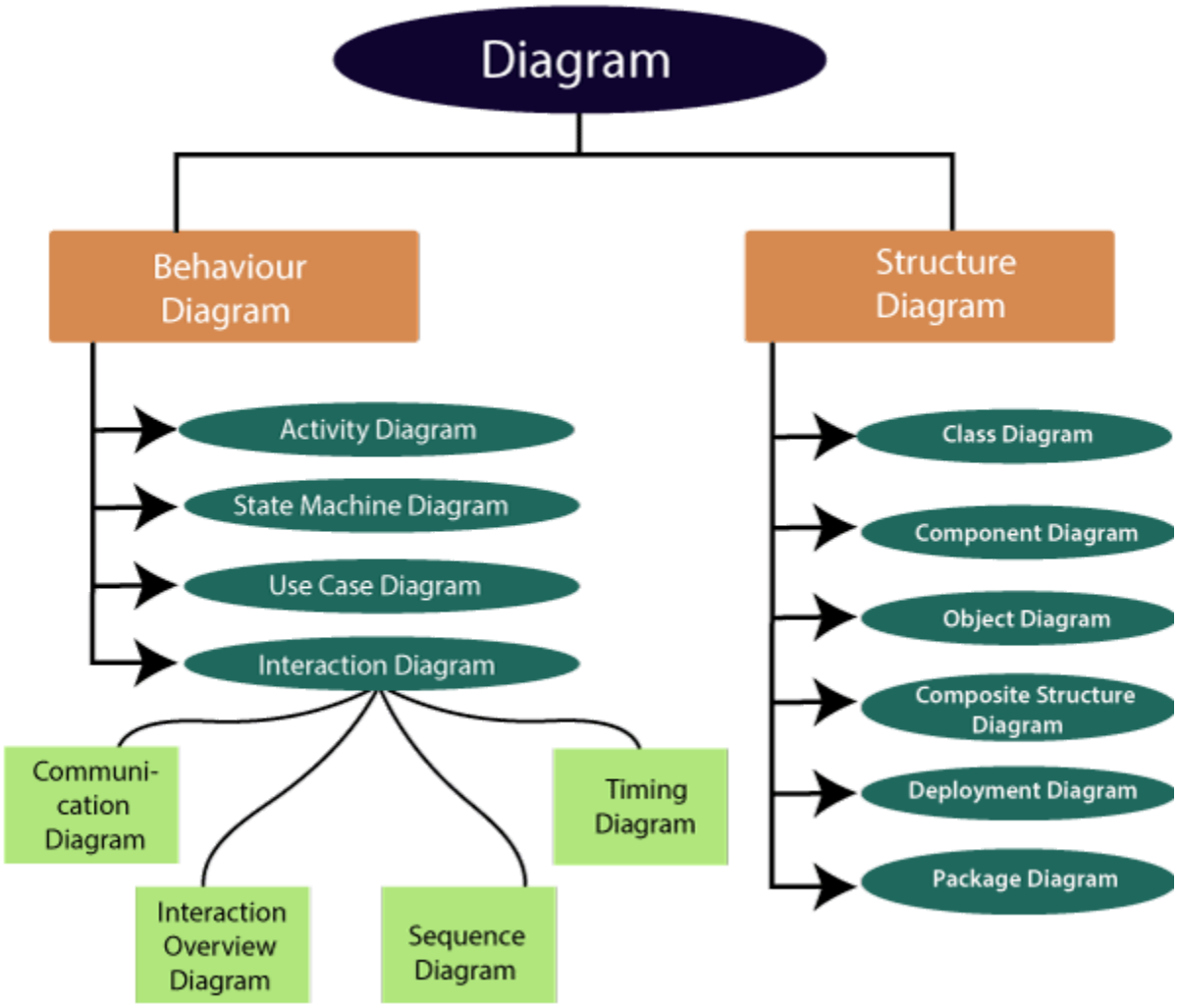
UML-Diagrams

The UML diagrams are categorized into structural diagrams, behavioral diagrams, and also interaction overview diagrams. The diagrams are hierarchically classified in the following figure:



1. Structural Diagrams

Structural diagrams depict a static view or structure of a system. It is widely used in the documentation of software architecture. It embraces class diagrams, composite structure diagrams, component diagrams, deployment diagrams, object diagrams,

and package diagrams. It presents an outline for the system. It stresses the elements to be present that are to be modeled.

Class Diagram: Class diagrams are one of the most widely used diagrams. It is the backbone of all the object-oriented software systems. It depicts the static structure of the system. It displays the system's class, attributes, and methods. It is helpful in recognizing the relation between different objects as well as classes.

Composite Structure Diagram: The composite structure diagrams show parts within the class. It displays the relationship between the parts and their configuration that ascertain the behavior of the class. It makes full use of ports, parts, and connectors to portray the internal structure of a structured classifier. It is similar to class diagrams, just the fact it represents individual parts in a detailed manner when compared with class diagrams.

Object Diagram: It describes the static structure of a system at a particular point in time. It can be used to test the accuracy of class diagrams. It represents distinct instances of classes and the relationship between them at a time.

Component Diagram: It portrays the organization of the physical components within the system. It is used for modeling execution details. It determines whether the desired functional requirements have been considered by the planned development or not, as it depicts the structural relationships between the elements of a software system.

Deployment Diagram: It presents the system's software and its hardware by telling what the existing physical components are and what software components are running on them. It produces information about system software. It is incorporated

whenever software is used, distributed, or deployed across multiple machines with dissimilar configurations.

Package Diagram: It is used to illustrate how the packages and their elements are organized. It shows the dependencies between distinct packages. It manages UML diagrams by making it easily understandable. It is used for organizing the class and use case diagrams.

2. Behavioral Diagrams:

Behavioral diagrams portray a dynamic view of a system or the behavior of a system, which describes the functioning of the system. It includes use case diagrams, state diagrams, and activity diagrams. It defines the interaction within the system.

State Machine Diagram: It is a behavioral diagram. it portrays the system's behavior utilizing finite state transitions. It is also known as the State-charts diagram. It models the dynamic behavior of a class in response to external stimuli.

Activity Diagram: It models the flow of control from one activity to the other. With the help of an activity diagram, we can model sequential and concurrent activities. It visually depicts the workflow as well as what causes an event to occur.

Use Case Diagram: It represents the functionality of a system by utilizing actors and use cases. It encapsulates the functional requirement of a system and its association with actors. It portrays the use case view of a system.

3. Interaction Diagrams

Interaction diagrams are a subclass of behavioral diagrams that give emphasis to object interactions and also depicts the flow between various use case elements of

a system. In simple words, it shows how objects interact with each other and how the data flows within them. It consists of communication, interaction overview, sequence, and timing diagrams.

Sequence Diagram: It shows the interactions between the objects in terms of messages exchanged over time. It delineates in what order and how the object functions are in a system.

Communication Diagram: It shows the interchange of sequence messages between the objects. It focuses on objects and their relations. It describes the static and dynamic behavior of a system.

Timing Diagram: It is a special kind of sequence diagram used to depict the object's behavior over a specific period of time. It governs the change in state and object behavior by showing the time and duration constraints.

Interaction Overview diagram: It is a mixture of activity and sequence diagram that depicts a sequence of actions to simplify the complex interactions into simple interactions.

UML Class Diagram

What is a Class?

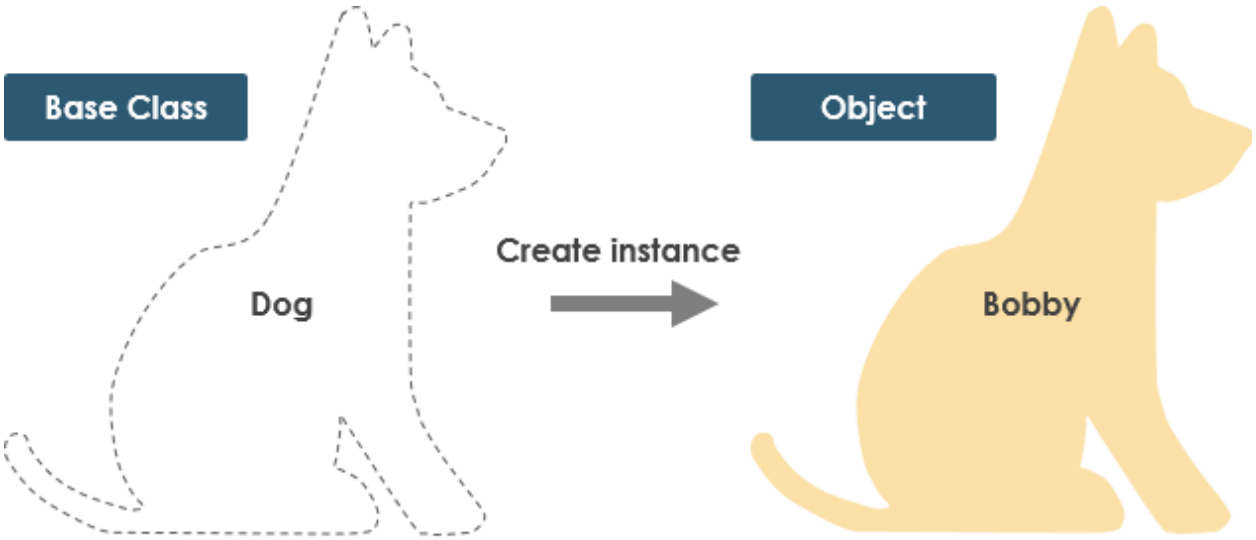
A Class is a blueprint for an object. Objects and classes go hand in hand. We can't talk about one without talking about the other. And the entire point of Object-Oriented Design is not about objects, it's about classes, because we use classes to create objects. So a class describes what an object will be, but it isn't the object itself.

In fact, classes describe the type of objects, while objects are usable instances of classes. Each Object was built from the same set of blueprints and therefore contains the same components (properties and methods). The standard meaning is that an object is an instance of a class and object - Objects have states and behaviors.

Example

A dog has states - color, name, breed as well as behaviors -wagging, barking, eating.

An object is an instance of a class.



Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Property Values	Methods
Color: Yellow	Sit
Eye Color: Brown	Lay Down
Height: 17 in	Shake
Length: 35 in	Come
Weight: 24 pounds	

UML Class Notation

A class represents a concept which encapsulates state (attributes) and behavior (operations). Each attribute has a type. Each operation has a signature. *The class name is the only mandatory information.*

Class Name:

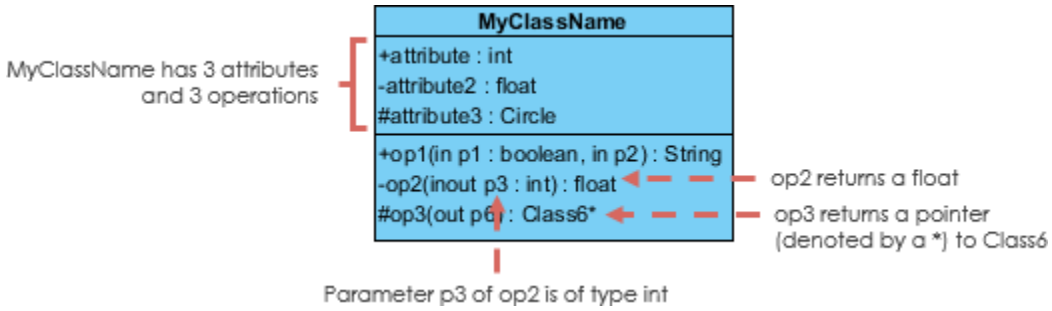
- The name of the class appears in the first partition.

Class Attributes:

- Attributes are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

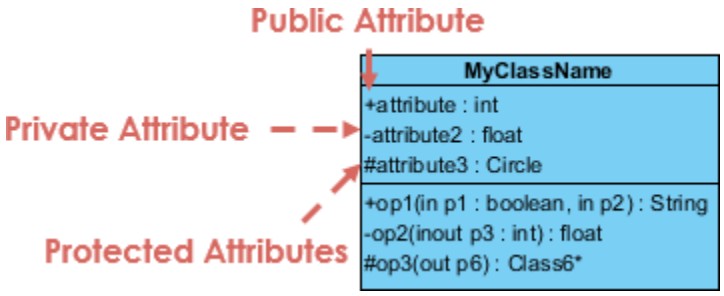
Class Operations (Methods):

- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code



Class Visibility

The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.

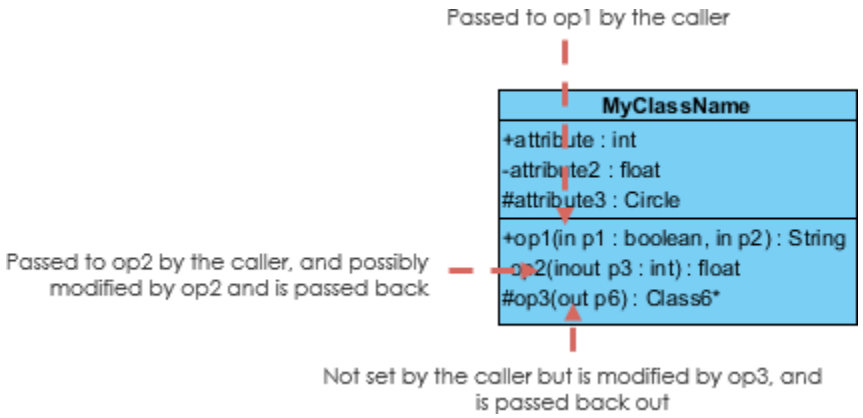


- + denotes public attributes or operations
- - denotes private attributes or operations

- # denotes protected attributes or operations

Parameter Directionality

Each parameter in an operation (method) may be denoted as in, out or inout which specifies its direction with respect to the caller. This directionality is shown before the parameter name.



Perspectives of Class Diagram

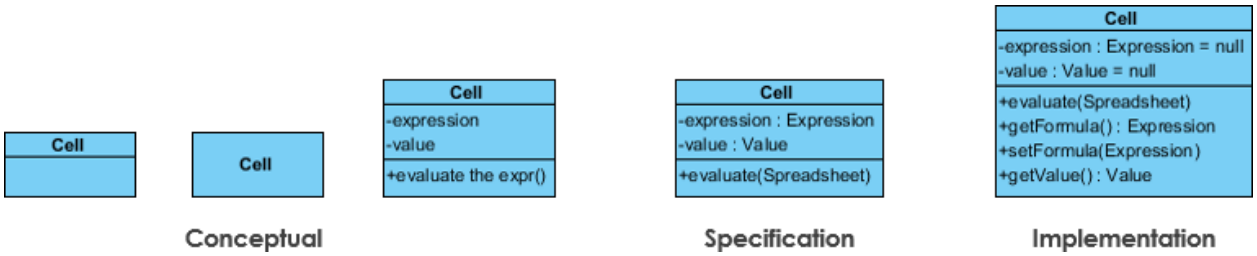
The choice of perspective depends on how far along you are in the development process. During the formulation of a domain model, for example, you would seldom move past the conceptual perspective. Analysis models will typically feature a mix of conceptual and specification perspectives. Design model development will typically start with heavy emphasis on the specification perspective, and evolve into the implementation perspective.

A diagram can be interpreted from various perspectives:

- Conceptual: represents the concepts in the domain

- Specification: focus is on the interfaces of Abstract Data Type (ADTs) in the software
- Implementation: describes how classes will implement their interfaces

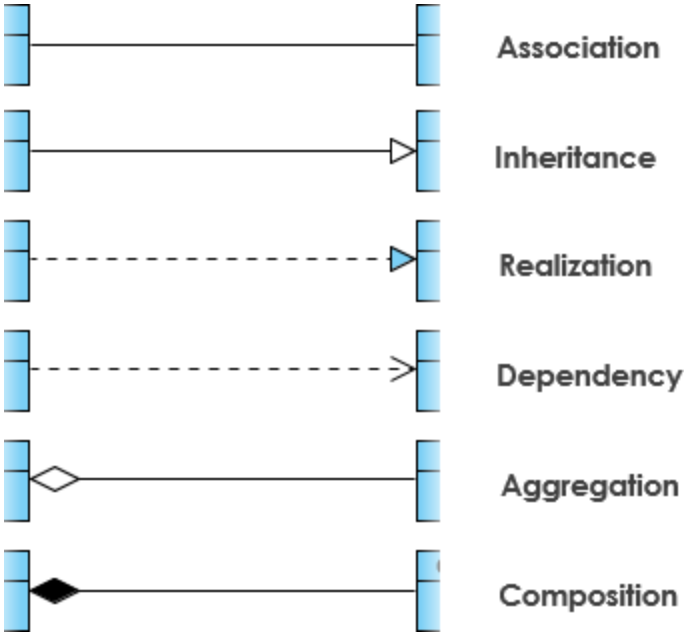
The perspective affects the amount of detail to be supplied and the kinds of relationships worth presenting. As we mentioned above, the class name is the only mandatory information.



Relationships between classes

UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. Can you describe what each of the relationships mean relative to your target programming language shown in the Figure below?

If you can't yet recognize them, no problem this section is meant to help you to understand UML class relationships. A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:

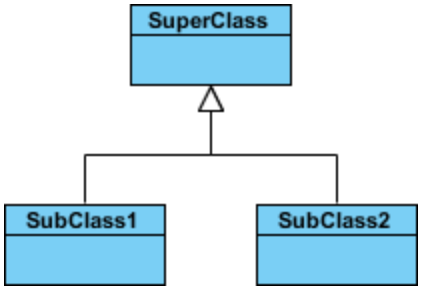


Inheritance (or Generalization):

A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier.

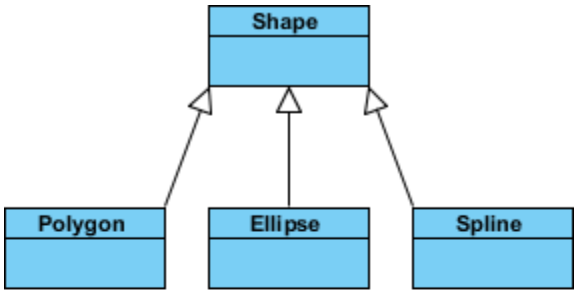
- Represents an "is-a" relationship.
- An abstract class name is shown in italics.
- SubClass1 and SubClass2 are specializations of SuperClass.

The figure below shows an example of inheritance hierarchy. SubClass1 and SubClass2 are derived from SuperClass. The relationship is displayed as a solid line with a hollow arrowhead that points from the child element to the parent element.

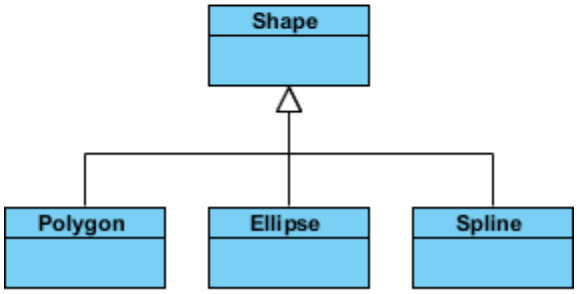


Inheritance Example - Shapes

The figure below shows an inheritance example with two styles. Although the connectors are drawn differently, they are semantically equivalent.



Style 1: Separate target



Style 2: Shared target

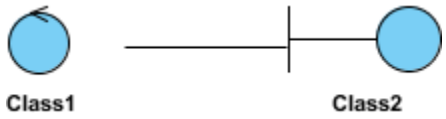
Association

Associations are relationships between classes in a UML Class Diagram. They are represented by a solid line between classes. Associations are typically named using a verb or verb phrase which reflects the real world problem domain.

Simple Association

- A structural link between two peer classes.
- There is an association between Class1 and Class2

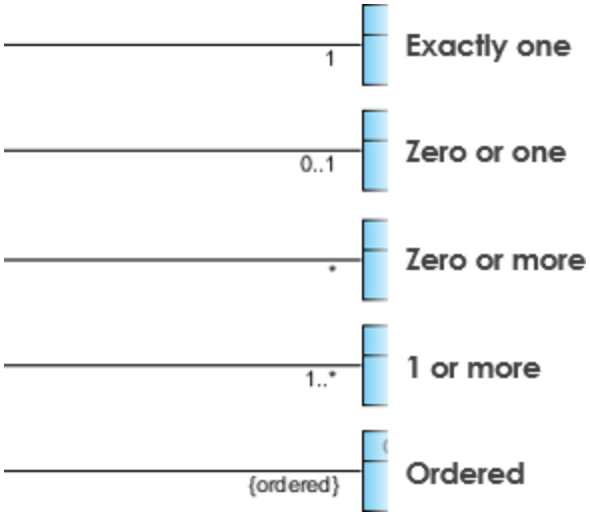
The figure below shows an example of simple association. There is an association that connects the <<control>> class Class1 and <<boundary>> class Class2. The relationship is displayed as a solid line connecting the two classes.



Cardinality

Cardinality is expressed in terms of:

- one to one
- one to many
- many to many

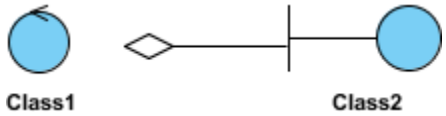


Aggregation

A special type of association.

- It represents a "part of" relationship.
- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.

The figure below shows an example of aggregation. The relationship is displayed as a solid line with a unfilled diamond at the association end, which is connected to the class that represents the aggregate.



Composition

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.

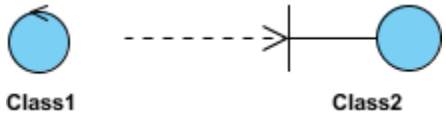
The figure below shows an example of composition. The relationship is displayed as a solid line with a filled diamond at the association end, which is connected to the class that represents the whole or composite.

Dependency

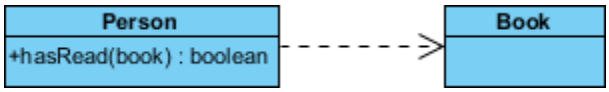
An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship.

- A special type of association.
- Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).
- Class1 depends on Class2

The figure below shows an example of dependency. The relationship is displayed as a dashed line with an open arrow.



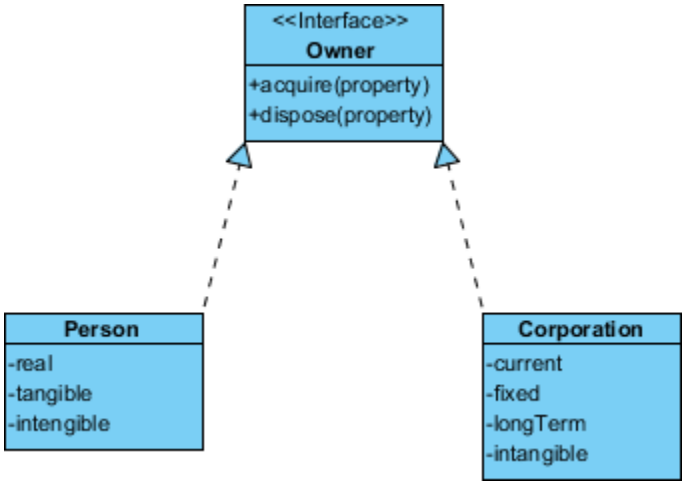
The figure below shows another example of dependency. The Person class might have a hasRead method with a Book parameter that returns true if the person has read the book (perhaps by checking some database).



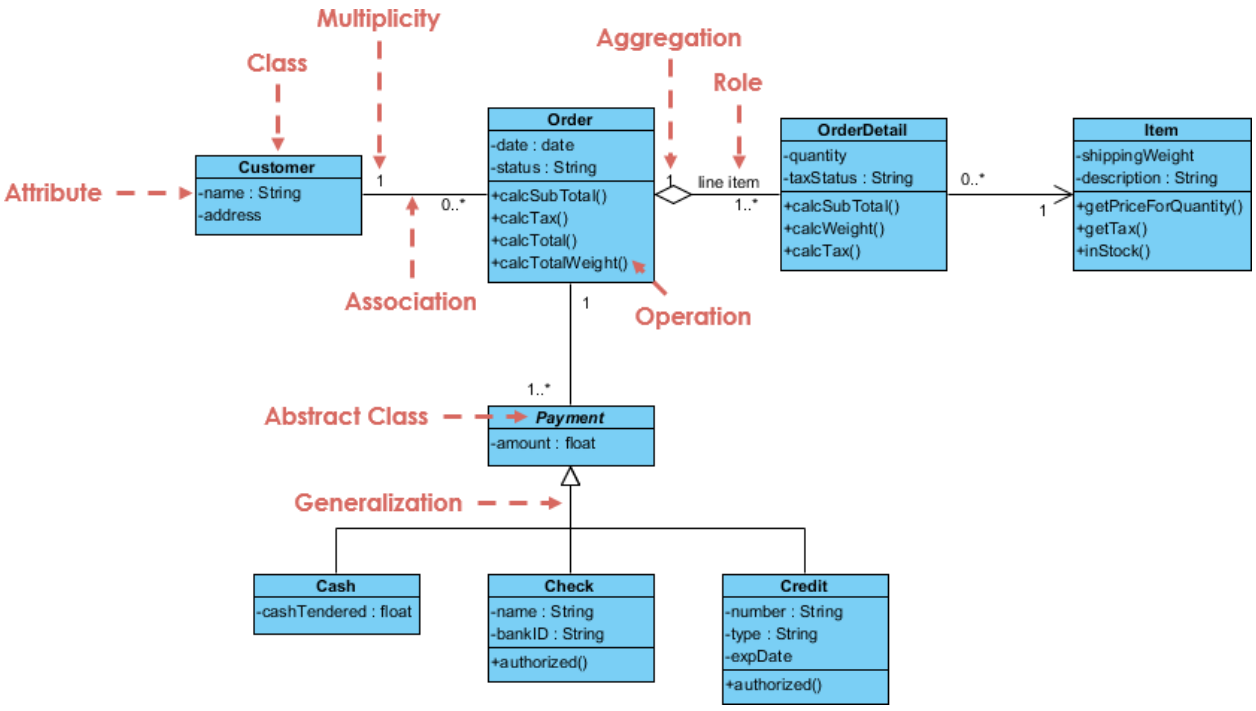
Realization

Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.

For example, the Owner interface might specify methods for acquiring property and disposing of property. The Person and Corporation classes need to implement these methods, possibly in very different ways.

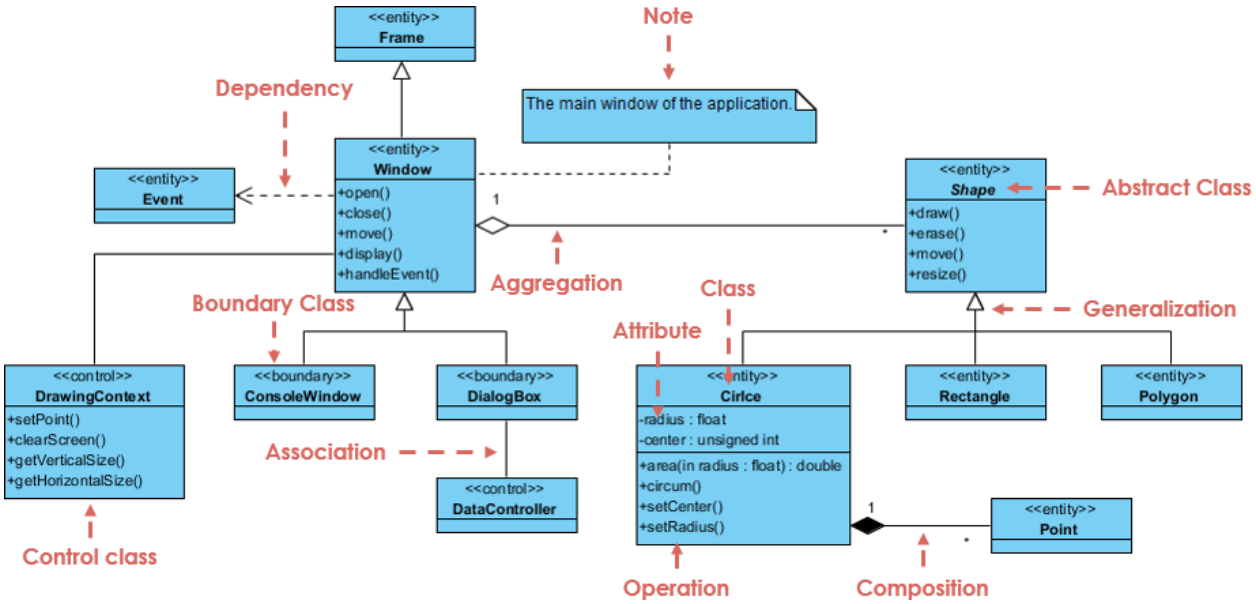


Class Diagram Example: Order System



Class Diagram Example: GUI

A class diagram may also have notes attached to classes or relationships.


<<entity>>
Frame
Note
Dependency
The main window of the application.
<<entity>>
Window
+open()
+close()
+move()
+display()
+handleEvent()
<<entity>>
Event
1
<<entity>>
Shape
+draw()
+erase()
+move()
+resize()
Abstract Class
Aggregation
Boundary Class
Class
Attribute
Generalization
<<control>>
DrawingContext
+setPoint()
+clearScreen()
+getVerticalSize()
+getHorizontalSize()
<<boundary>>
ConsoleWindow
<<boundary>>
DialogBox
<<entity>>
Circle
-radius : float
-center : unsigned int
+area(in radius : float) : double
+circum()
+setCenter()
+setRadius()
<<entity>>
Rectangle
<<entity>>
Polygon
Association
<<control>>
DataController
1
<<entity>>
Point
Control class
Operation
Composition