Module 2: CLASS DIAGRAM

This supplementary material is provided by tdque@yahoo.com to Students D15 - PTIT (2019-2020) via the subject **Information System Analysis & Design** (A&D)

The greater our knowledge increases, the greater our ignorance unfolds

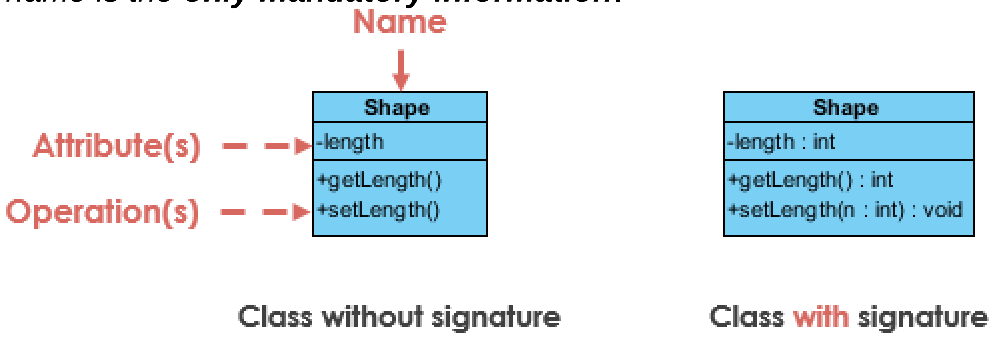
John F. Kennedy

==

Reference: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

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:

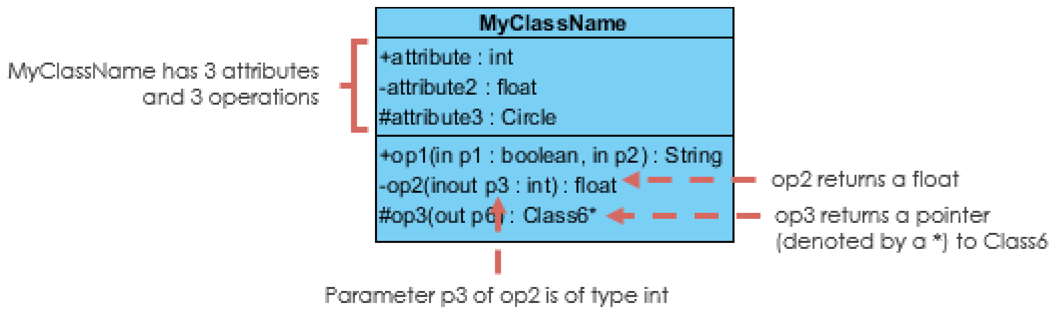
- The name of the class appears in the first partition.

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.

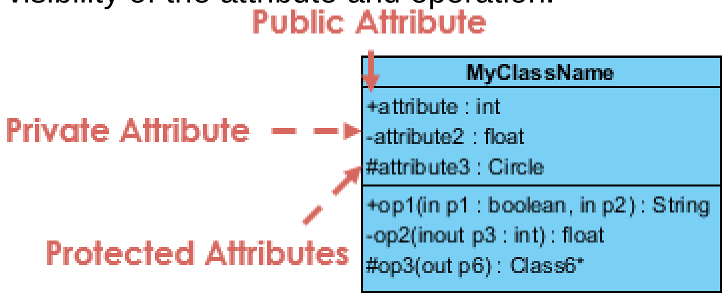
Class Operations (Methods):

- 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


MyClassName has 3 attributes and 3 operations
MyClassName
+attribute : int
-attribute2 : float
#attribute3 : Circle
+op1(in p1 : boolean, in p2) : String
-op2(inout p3 : int) : float
#op3(out p6) : Class6*
op2 returns a float
op3 returns a pointer (denoted by a *) to Class6
Parameter p3 of op2 is of type int

Class Visibility

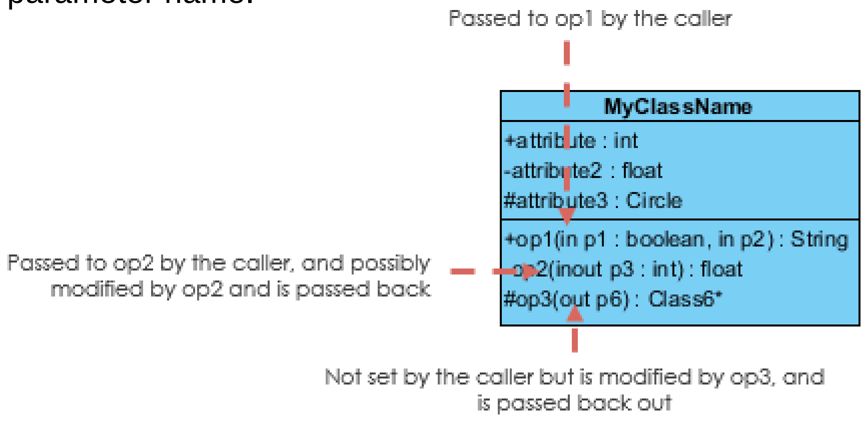
The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



- + 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.


Passed to op1 by the caller
MyClassName
+attribute : int
-attribute2 : float
#attribute3 : Circle
+op1(in p1 : boolean, in p2) : String
-op2(inout p3 : int) : float
#op3(out p6) : Class6*
Passed to op2 by the caller, and possibly modified by op2 and is passed back
Not set by the caller but is modified by op3, and is passed back out

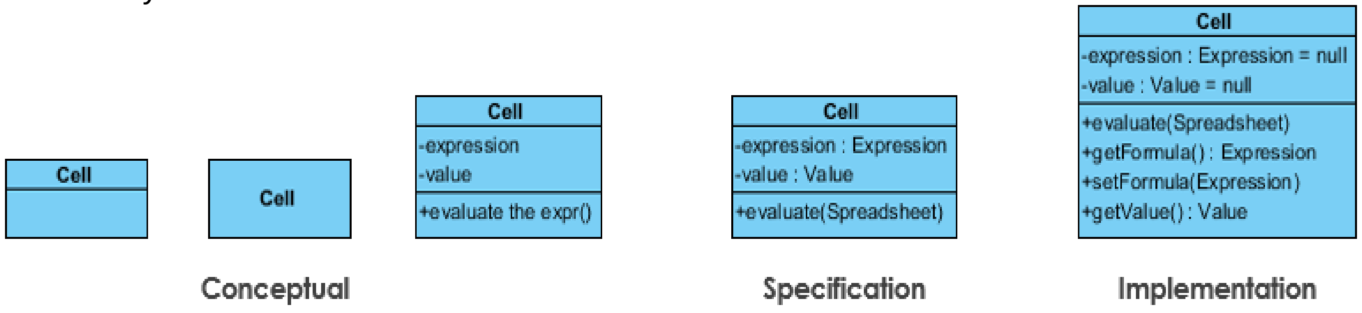
Perspectives of Class Diagram

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

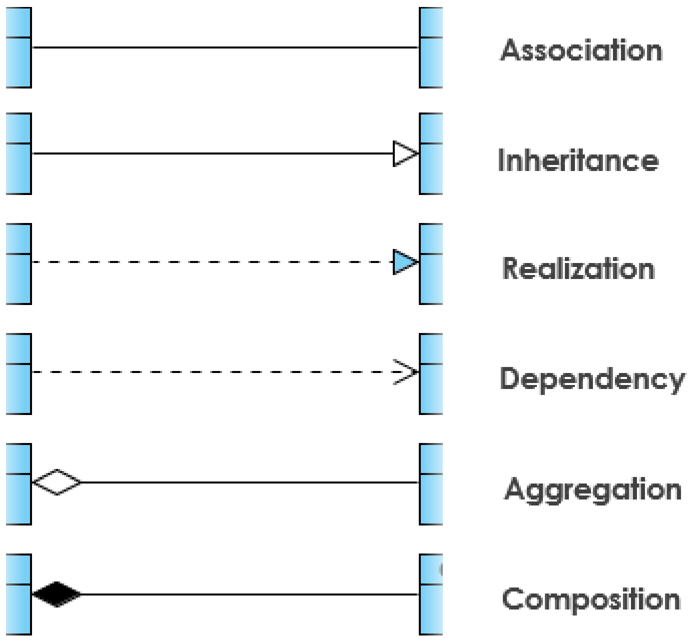
The perspective affects the amount of detail to be supplied and the kinds of relationships worth presenting. As we mentioned above, the class name is the only mandatory information.



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:

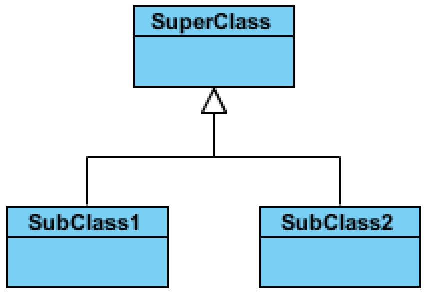


Inheritance (or Generalization):

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.

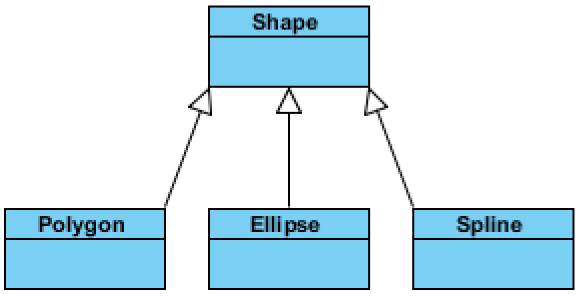
- Represents an "is-a" relationship.
- An abstract class name is shown in italics.
- SubClass1 and SubClass2 are specializations of SuperClass.

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.

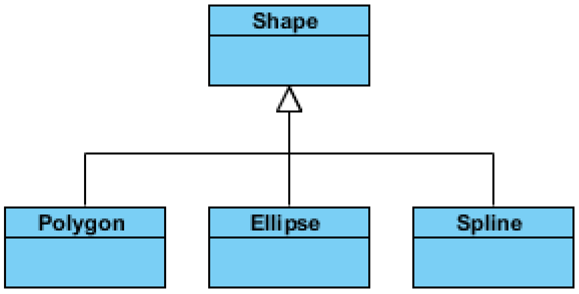


Inheritance Example - Shapes

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

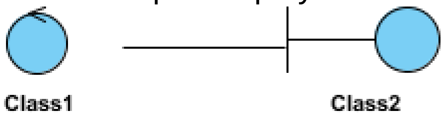
Association

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

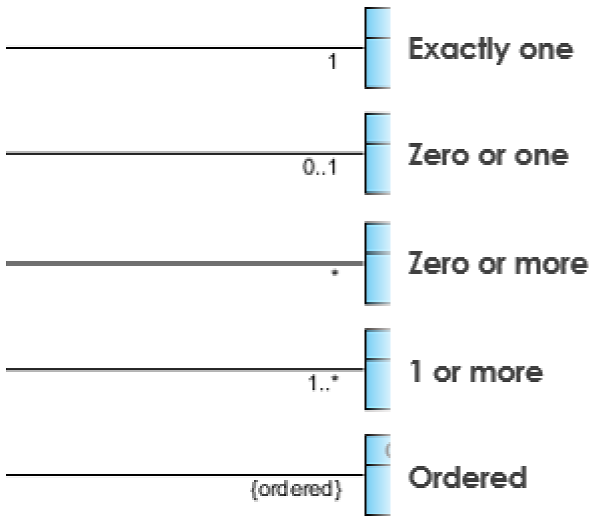
The figure below shows an example of simple association. There is an association that connects the <<control>> class Class1 and <<boundary>> class Class2. The relationship is displayed as a solid line connecting the two classes.



Cardinality

Cardinality is expressed in terms of:

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

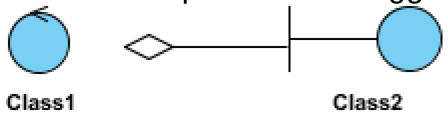


Aggregation

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.

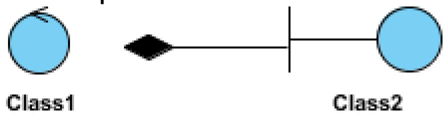
The figure below shows an example of aggregation. The relationship is displayed as a solid line with a unfilled diamond at the association end, which is connected to the class that represents the aggregate.



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.

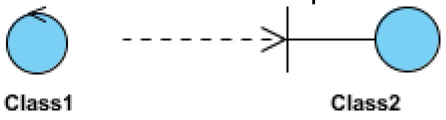


Dependency

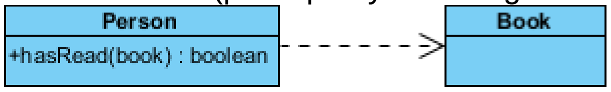
An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship.

- 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.



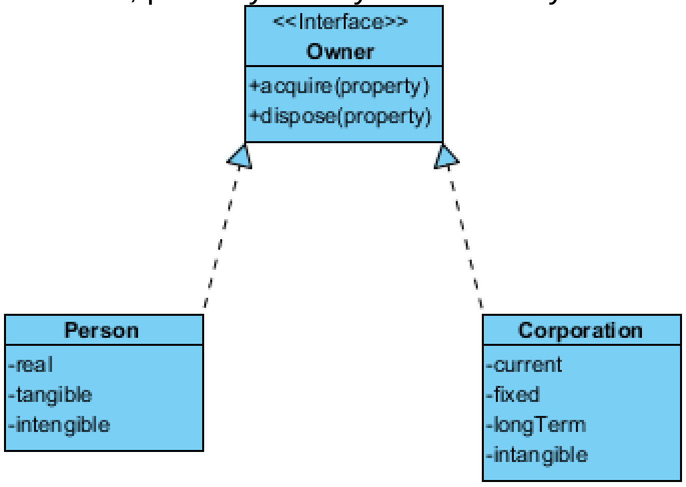
The figure below shows another example of dependency. The Person class might have a hasRead method with a Book parameter that returns true if the person has read the book (perhaps by checking some database).



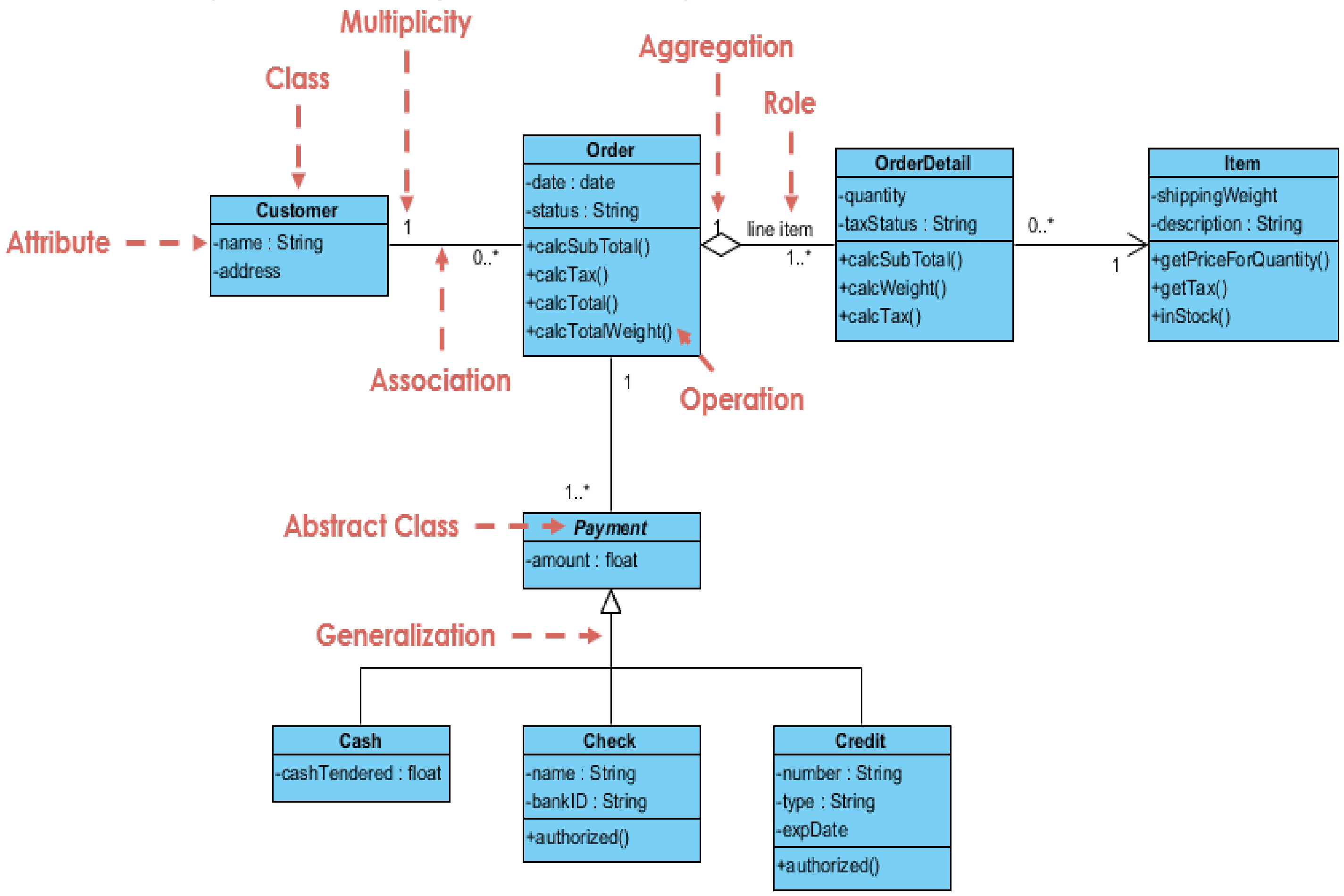
Realization

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.

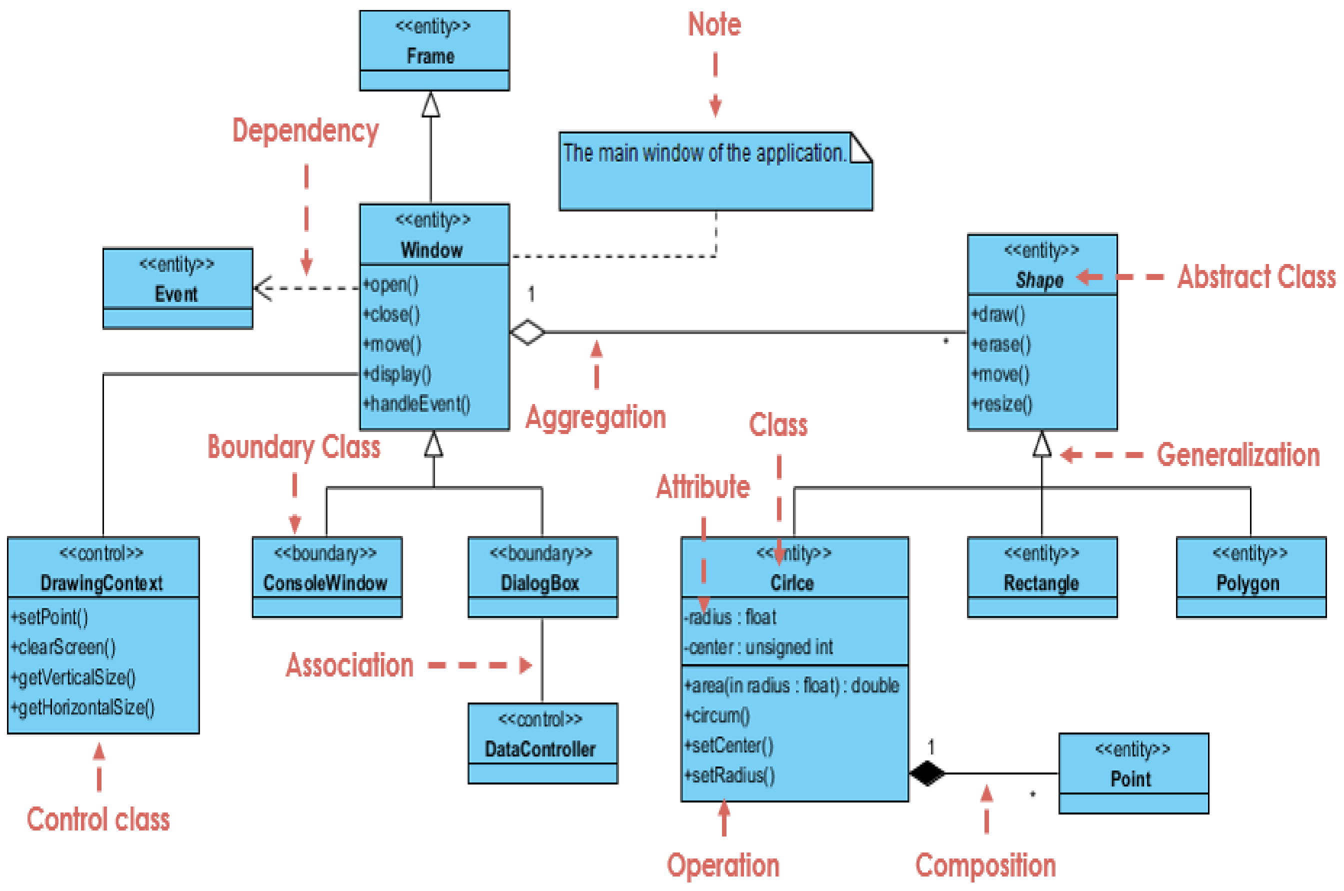


Class Diagram Example: Order System



Class Diagram Example: GUI

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



Note
<<entity>> Frame
Dependency
The main window of the application.
<<entity>> Window
+open()
+close()
+move()
+display()
+handleEvent()
<<entity>> Event
1
*
<<entity>> Shape
Abstract Class
+draw()
+erase()
+move()
+resize()
Aggregation
Boundary Class
Class
Generalization
Attribute
<<control>> DrawingContext
+setPoint()
+clearScreen()
+getVerticalSize()
+getHorizontalSize()
<<boundary>> ConsoleWindow
<<boundary>> DialogBox
<<entity>> Cirlce
-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

All the UML you need to know

By Paul Gestwicki

http://www.cs.bsu.edu/~pvgestwicki/misc/uml/

Introduction

This page describes the elements of the UML that I expect my students to know. By no means does this document attempt to portray all of the UML. Those elements described herein are those that I have found useful in practice and those that I have seen featured in formal and informal written communication. That is, these are the minimum features that I consider to represent UML literacy.

Sequence Diagrams

This document is being written in Fall 2011, and we already talked about these in class. I'm not going to invest the time in reiterating the same content here, since you have it in your notes already.

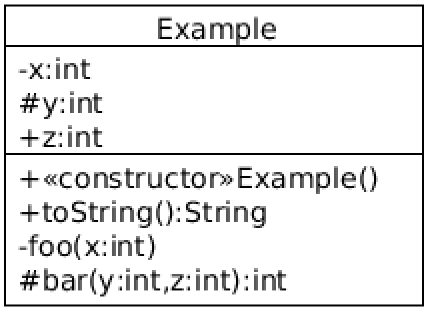
Class Diagrams

Classes and interfaces

A **class** is represented by a box with up to three sections: the top contains the class name; the middle contains the fields; the bottom contains the methods. Consider the following Java class definition, a ridiculously-designed example that will serve to demonstrate core UML data representations. (Note that if you're one of my students and you ever turn in programs as nonsensical as this, expect to be harassed.)

```
public class Example {
  private int x;
  protected int y;
  public int z;
  public Example() { ... }
  public String toString() { ... }
  private void foo(int x) { ... }
  protected int bar(int y, int z) { ... }
}
```

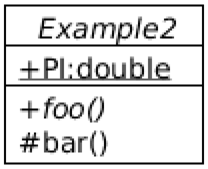
This can be represented with the following class diagram.



The fields and methods are annotated to indicate their access level: plus (+) for public, minus (-) for private, and hash (#) for protected. UML conventially uses Algol-style naming, so variables are given as *name*:*type* and methods are given as *name*(*params*):*type*, where each parameter is, of course, a variable. In UML, metadata is often represented through stereotypes, which are always listed in guillemet. For example, the fact that our `Example()` method is a constructor is identified via the «constructor» stereotype.

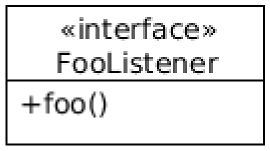
Static members in class diagrams are underlined, and abstract elements are italicized. Here is another code and diagram example.

```
public abstract class Example2 {
  public static final double PI = 3.14;
  public abstract void foo() { ... }
  protected void bar() { ... }
}
```



Interfaces are given the «interface» annotation, as shown below.

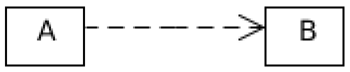
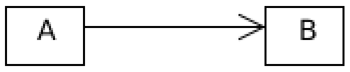
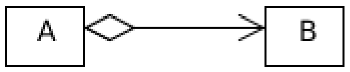
```
public interface FooListener {
  public void foo();
}
```

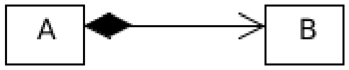


Keep in mind that UML is a communication tool, and you can omit details that are not necessary for expressing your message. For example, I frequently skip the middle box in UML classes since they deal with data representation, and I'm usually more interested in capturing the relationships among classes.

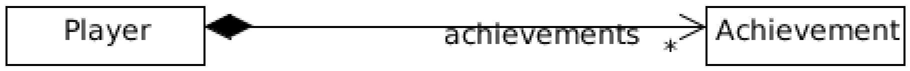
Relationships

It is the messages sent among objects that give a system dynamic behavior, and these are represented in UML through the relationships among classes. There are four kinds of relationships that I use regularly, shown in the following table in order of increasing specificity. That is, the relationships lower on the table subsume all those above them.

Relationship	Depiction	Interpretation
Dependency		A depends on B This is a very loose relationship and so I rarely use it, but it's good to recognize and be able to read it.
Association		An A sends messages to a B Associations imply a direct communication path. In programming terms, it means instances of A can call methods of instances of B, for example, if a B is passed to a method of an A.
Aggregation		An A is made up of B This is a part-to-whole relationship, where A is the whole and B is the part. In code, this essentially implies A has fields of type B.

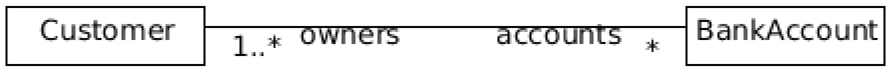
Composition		An A is made up of B with lifetime dependency That is, A aggregates B, and if the A is destroyed, its B are destroyed as well.

A useful annotation on these relationships is *multiplicity*, which tells you how many of each object is involved in the relationship. This can be a constant ("1"), unbounded ("*", *i.e.*, zero or more), or a range ("2..*"). Where multiplicity is not explicit, "1" is assumed. You can also annotate a relationship with *roles* to further describe the relationship; these roles may translate into fields in the implementation. For example, the relationship between a player and its achievements might be represented as follows:

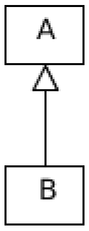
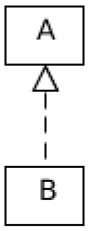


(The reader may wonder what actual data structure is used to hold the achievements: is it an array, or a linked list, or something else? At this level of modeling, that's probably not important. You could use UML to show that Player aggregates a java.util.LinkedList, and that this list aggregates Achivement objects, but unless that's essential to your reader, you're best to skip it.)

Directionality is another important aspect of relationships. All of my examples above have been unidirectional, but relationships may also be bidirectional. This is shown by omitting the arrowhead. For example, the following diagram shows a case where a customer may have any number of bank accounts, and a bank account can be owned by one or more customers. Note that you could use the open diamond annotation to show aggregation if you were interpreting this as a part-to-whole on either or both sides. (This proves the fact that you cannot write a UML tutorial without a bank account example. QED.)



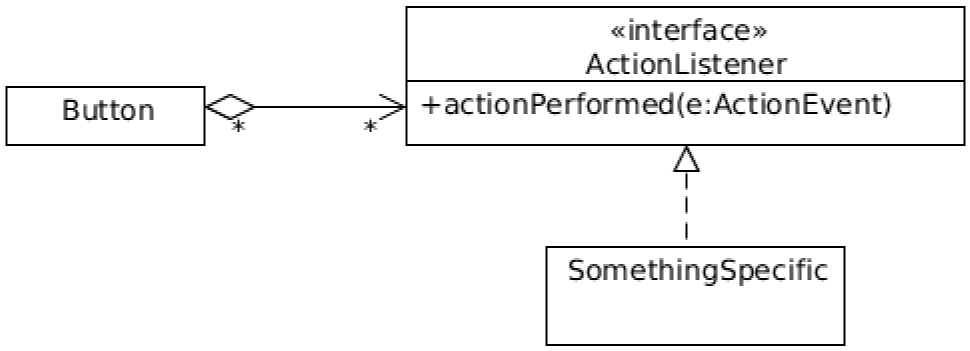
Two other important relationships deal with the relationship among classes, shown in the table below.

Relationship	Depiction	Interpretation
Generalization		A generalizes B Equivalently, B is a subclass of A. In Java, this is extends.
Realization		B realizes (the interface defined in) A As the parenthetical name implies, this is used to show that a class realizes an interface. In Java, this is implements, and so it would be common for A to have the «interface» stereotype.

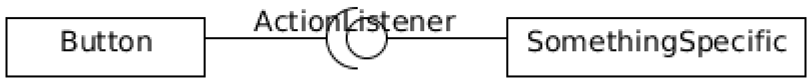
Note that it's not mandatory to draw these with vertical alignment, but I do recommend it to improve readability. Most readers will conceptualize the upper class as more general and the lower class as more specific; if you were to invert your relationship, you would be causing unnecessary cognitive dissonance.

Multiple representations, plus collaboration notation

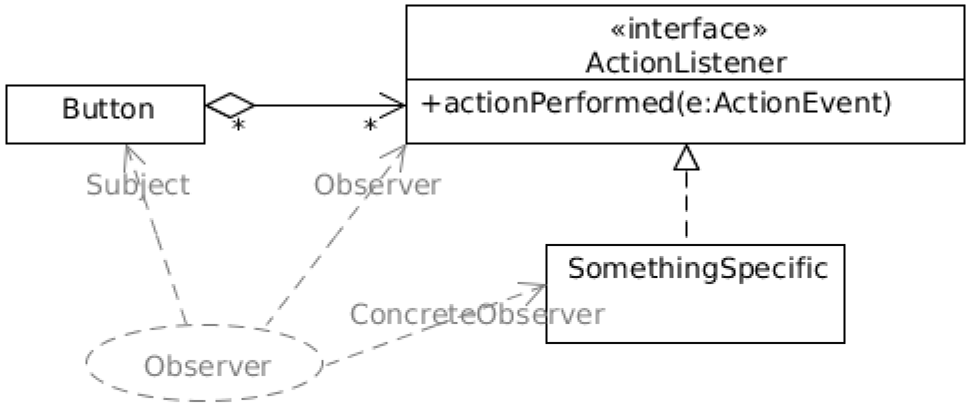
In a good OO design, you have cross-module dependencies on interfaces, not implementations. As a result, you might frequently encounter cases as shown below, where a Button sends messages through the ActionListener interface



If what you're trying to express is that information gets to SomethingSpecific from Button, then you can use the ball-and-socket notation instead, as shown below. This says that the communication between Button and SomethingSpecific happens through the ActionListener interface.

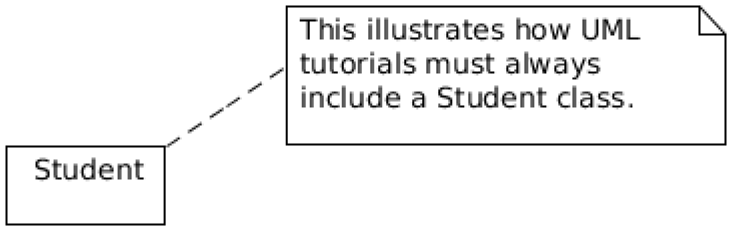


On the other hand, if the important thing to show is that this is a reification of the observer design pattern, then you can use a UML collaboration to show this. As always, it all comes down to knowing what you want to say and then using the notation to your advantage.



Miscellaneous

You can put a note on any part of a UML diagram. Connect the note to the relevant bit with a dashed line, as shown in the example below.


This illustrates how UML tutorials must always include a Student class.
Student

Concluding Remarks

The UML is a massive specification, and I've only showed one or two kinds of diagrams above. These are the diagrams I encounter most often in research, on the Web, and in print. For your next steps, I would recommend learning state machine diagrams, activity diagrams, and use case diagrams. One of the most useful UML resources I have found online is Allen Holub's UML Quick Reference—great for getting a refresher or a birds-eye view on a diagram type.

All of the diagrams above were created with UMLet, an amazing and free tool for rapidly creation of UML diagrams.