Object-Oriented Software Development

Unified Modeling Language (UML)

Class Diagrams

©2012 Christopher Hield

UML Class Diagrams

UML

UML stands for Unified Modeling Language. The goal is for UML to become a common "language" for creating models of object oriented computer software. The UML is comprised of two major components: a meta-model and a notation.

- **Meta-model -** UML is unique in that it has a standard data representation. This representation is called the meta-model. The meta-model is a description of UML in UML. It describes the objects, attributes, and relationships necessary to represent the concepts of UML within a software application. This provides CASE manufacturers with a standard and unambiguous way to represent UML models. Hopefully it will allow for easy transport of UML models between tools. It may also make it easier to write ancillary tools for browsing, summarizing, and modifying UML models.

- **Notation -** The UML notation is rich and full bodied. It is comprised of two major subdivisions. There is a notation for modeling the static elements of a design such as classes, attributes, and relationships. There is also a notation for modeling the dynamic elements of a design such as objects, messages, and finite state machines. Static models are presented in diagrams called: Class Diagrams.

©2012 Christopher Hield

UML Class Diagrams

Classes

The purpose of the class diagram is to show the static structure of the system being modeled. The diagram specifically shows the entities in the system along with each entity's internal structure and relationships with other entities in the system. Because class diagrams only model the static structure of a system, only *types* of entities are shown on a class diagram; specific instances are not shown. For example, a class diagram would show an Employee class, but would not show actual employee instances such as John Smith, Jane Doe, etc.

Developers typically think of the class diagram as a diagram specifically meant for them, because they can use it to find out details about the system's coded classes or soon-to-be-coded classes, along with each class's attributes and methods. Class diagrams are particularly useful for business modeling, too. Business analysts can use class diagrams to model a business's current assets and resources, such as account ledgers, products, or geographic hierarchy.

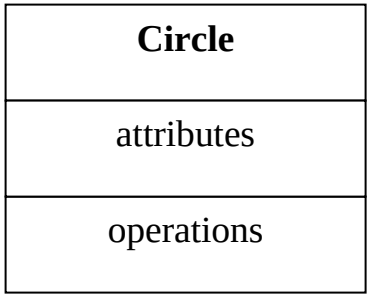
Modeling the static structure of classes, the class diagram shows each class's internal structure along with the relationship that the class has to other classes. In an object-oriented application, classes have attributes (member variables), operations (member functions) and relationships with other classes. The UML class diagram can depict all these things quite easily.

©2012 Christopher Hield

UML Class Diagrams

Classes

The fundamental element of the class diagram is an icon the represents a class.



A class icon is simply a rectangle divided into three compartments. The topmost compartment contains the name of the class. The middle compartment contains a list of attributes (member variables), and the bottom compartment contains a list of operations (member functions). In many diagrams, the bottom two compartments are omitted. Even when they are present, they typically do not show every attribute and operations. The goal is to show only those attributes and operations that are useful for the particular diagram.

This ability to abbreviate an icon is one of the hallmarks of UML. Each diagram has a particular purpose. That purpose may be to highlight on particular part of the system, or it may be to illuminate the system in general. The class icons in such diagrams are abbreviated as necessary. There is typically never a need to show every attribute and operation of a class on any diagram.

©2012 Christopher Hield

UML Class Diagrams

Class Diagrams

The following shows a typical UML description of a class that represents a circle.

Class
- radius:double # center:Point
+ area():double + circumference():double # setRadius(double):void # setCenter(Point):void

The attribute section of a class (the middle compartment) lists each of the class's attributes on a separate line. UML identifies four types of visibility: public, protected, private, and package. To display visibility on the class diagram, you place the visibility indicator in front of the attribute or operation name. UML visibility indicators are defined as: **“** + ” for public, “ # ” for protected, “ - ” for private, and “ ~ “ for package.

Data attributes are defined using the following format:

```
visibility name : attribute type
```

For example:

```
- radius : double
# center : Point
```

©2012 Christopher Hield

UML Class Diagrams

Class Diagrams

Sometimes it is useful to show on a class diagram that a particular attribute has a default value. The UML specification allows for the identification of default values in the attribute list section by using the following notation:

```
visibility name : attribute type = default value
```

For example:

```
- radius : double = 1.0
# center : Point = new Point(0,0)
```

Notice that each member variable is followed by a colon and by the type of the variable. If the type is redundant, or otherwise unnecessary, it can be omitted. Notice also that the return values follow the member functions in a similar fashion. Again, these can be omitted.

Finally, notice that the member function arguments are just types. We could have named them too, and used colons to separate them from their types; or we could have omitted the arguments altogether.

©2012 Christopher Hield

UML Class Diagrams

Class Diagrams

Like attributes, the operations of a class are displayed in a list format, with each operation on its own line. As with data attributes, operation (method) visibility indicators are defined as: **“** + ” for public, “ # ” for protected, “ - ” for private, and “ ~ “ for package.

Operations are documented using the following notation:

```
visibility name(parameter list) : type of value returned
```

For example:

```
+ update(int) : void
# getSpeed() : double
-  createCenter(int, int) : Point
```

Data attributes and operations that are “static” are underlined to visually indicate that they are static.

<u>`+ getCount() : int`</u>

©2012 Christopher Hield

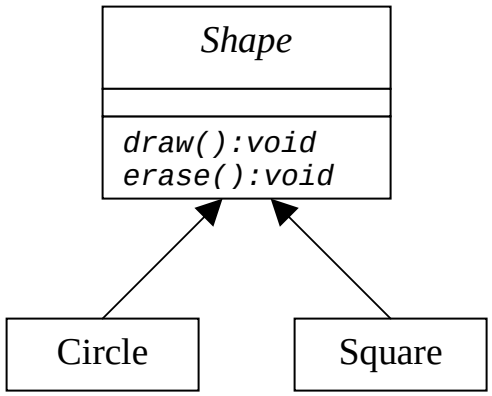
UML Class Diagrams

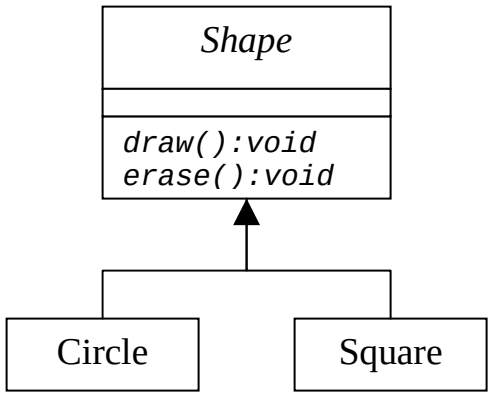
Generalization (Inheritance)

The inheritance relationship in UML is depicted by a triangular arrowhead. This arrowhead points the derived class to the base class.

The following figures show the inheritance relationship. We see that Circle and Square both derive from Shape. Note that the name of class Shape is shown in italics. This indicates that Shape is an "abstract" class. Note also that the operations, draw() and erase() are also shown in italics. This indicates that they are abstract as well.

(Note that both diagrams represent the same inheritance relationship – they are both presented to show variations in diagramming style.)



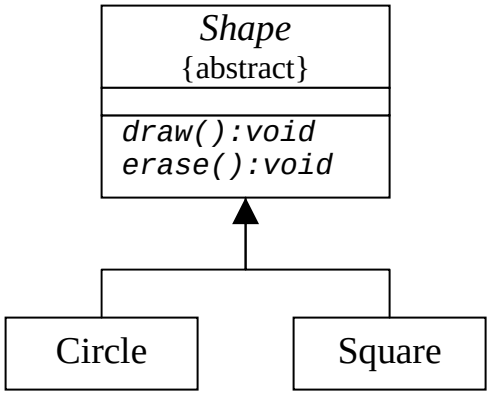


©2012 Christopher Hield

UML Class Diagrams

Generalization (Inheritance)

The italics used to show that a class is abstract are not always very easy to see. Therefore, an abstract class can also be marked with the {abstract} property.



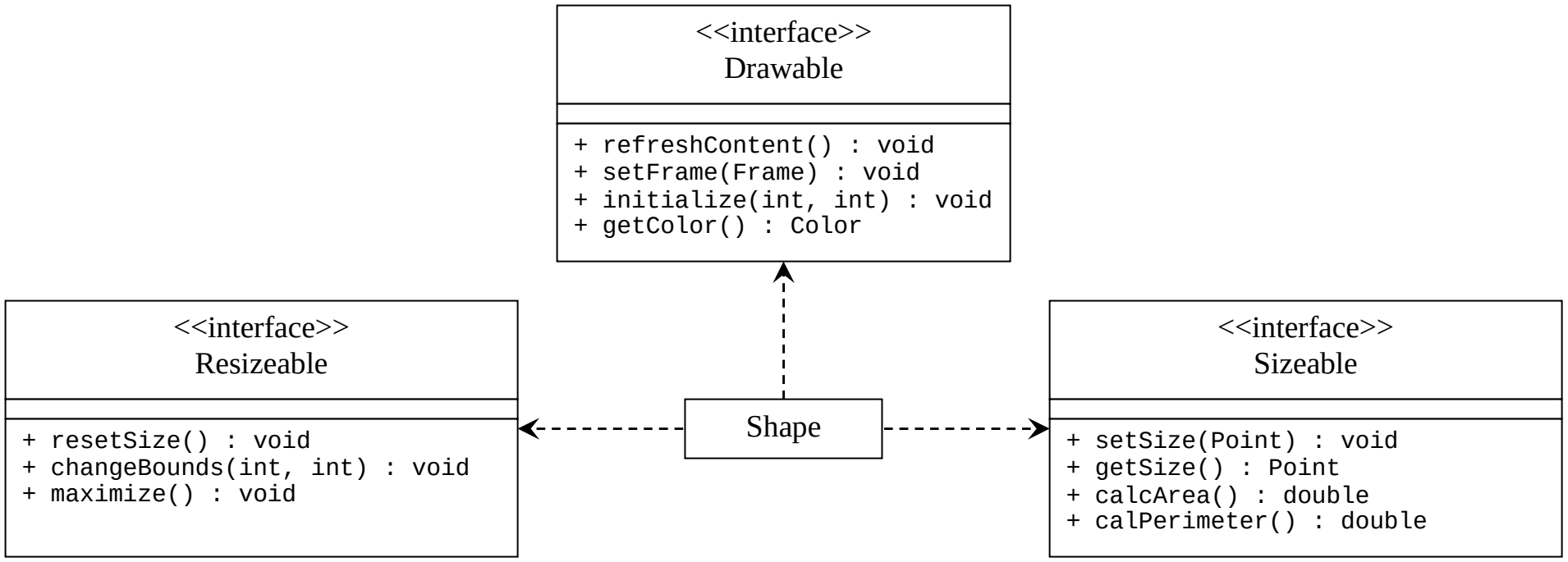
©2012 Christopher Hield

UML Class Diagrams

Realization (also known as Implementation)

A “realization” relationship indicates that one class implements a behavior specified by another class (i.e., an interface). An interface can be realized by many classes and a class can realize many interfaces.

The realization relationship notation is similar to generalization, though the relationship line is dashed. The Interface is represented much like a Class, with operations specified as shown below. The Shape class implements the Resizeable, Drawable, and Sizeable interfaces.

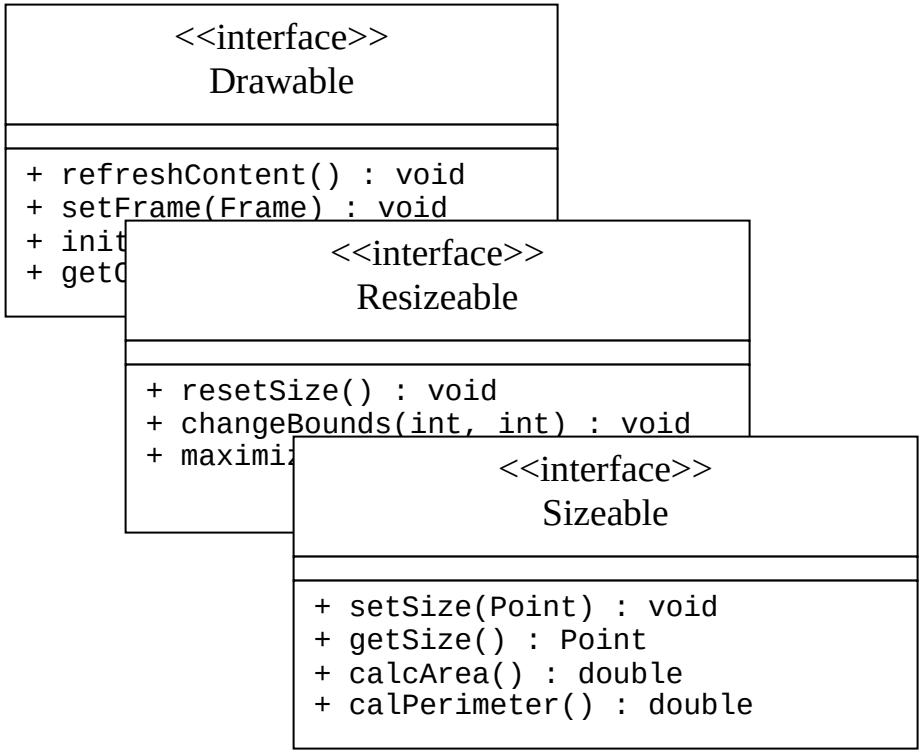


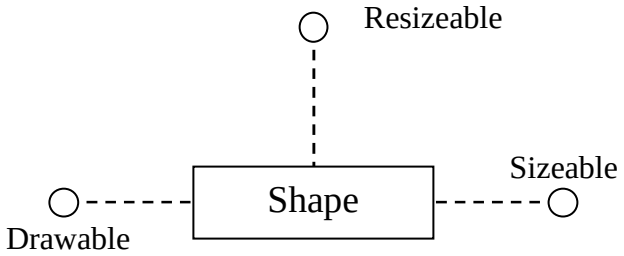
©2012 Christopher Hield

UML Class Diagrams

Realization (cont.)

The UML 2.0 specification introduced an alternate representation for interfaces and interface realization. The alternate representation is commonly known as the “Lollipop” notation. You apply the “Lollipop” notation to indicate that a class “realizes” an interface. This notation makes larger (more real-sized) UML Class Diagrams less cluttered by removing the proliferation of interface realization lines/arrows. The following shows the same realization relationships as the previous page, note the lack of interface realization lines/arrows.



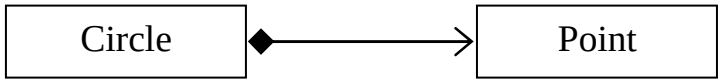


©2012 Christopher Hield

UML Class Diagrams

Composition Relationships

Assume that each instance of the Circle class contains a data attribute of type "Point". The Circle must have a Point (representing the center) to exist. This type of relationship is known as composition. It can be depicted in UML as follows:



The black diamond on the "Circle" end of the line represents a composition. It is placed on the Circle's side of the line shows that the Circle that is *composed* of a Point. The arrowhead on the other end of the relationship denotes that the relationship is navigable in only one direction. That is, Circle knows about it's Point, but the Point does not know about the Circle. At the code level this would imply a reference to the Point class from within the Circle class.
In UML, relationships are presumed to be bidirectional unless the arrowhead is present to restrict them, in which case they would be unidirectional. Were the arrowhead omitted, it would have meant that the Point knew about the Circle.

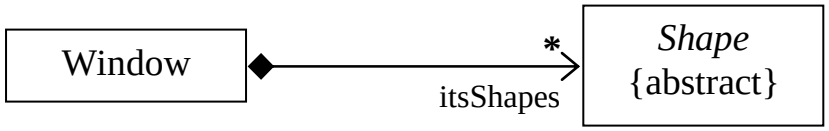
Composition relationships are a strong form of containment or aggregation. Aggregation is a whole/part relationship. In this case, Circle is the whole, and Point is part of Circle. However, composition is more than just aggregation. Composition also indicates that the lifetime of Point is dependent upon Circle. This means that if Circle is destroyed, Point will be destroyed with it.

©2012 Christopher Hield

UML Class Diagrams

Aggregation

Aggregation is a weak form of composition, denoted with an "open" diamond. This relationship denotes that the aggregate class (the class with the white diamond touching it) is in some way the "whole", and the other class in the relationship is somehow "part" of that whole.



This figure shows an aggregation relationship. In this case, the Window class contains many Shape instances. In UML the ends of a relationship are referred to as its "roles". Notice that the "Shape" end of the aggregation is marked with a "*". This is the "multiplicity" indicator, which indicates that the Window contains *many* Shape instances. Notice also that the role has been named. This is the name that Window knows its Shape instances by. (i.e. it is the name of the instance variable within Window that holds all the Shapes).

Multiplicity indicators can take on the following types of values: 1, *N*, *, 0..1, 0..*, 1..*N*, $N_1..N_2$, etc.

(Note "" is equivalent to "0..*" – which means "any number is valid")*

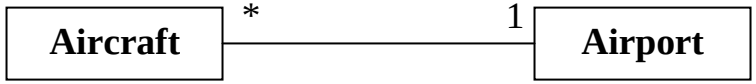
©2012 Christopher Hield

UML Class Diagrams

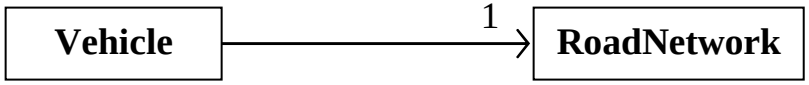
Association

An association is a linkage between two classes where no ownership is implied. Associations are assumed to be bi-directional - in other words, both classes are aware of their relationship and of the other class - unless you qualify the association as some other type of association.

A solid line between the two classes indicates a bi-directional association where each class knows about (i.e., “refers to”) the other. At either end of the line, we may place a role name and a multiplicity value.



A unidirectional association shows that two classes are related, but only one class knows that the relationship exists. A unidirectional association is drawn as a solid line with an open arrowhead (not the closed arrowhead, or triangle, used to indicate inheritance) pointing to the known class.



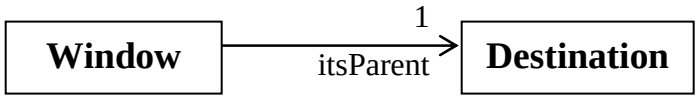
Like standard associations, the unidirectional association includes a role name and a multiplicity value, but unlike the standard bi-directional association, the unidirectional association only contains the role name and multiplicity value for the known class.

©2012 Christopher Hield

UML Class Diagrams

Association

The next figure shows how we draw an unidirectional association with a “multiplicity” and a “role” indicator.



What is the Difference Between an Aggregation and an Association?

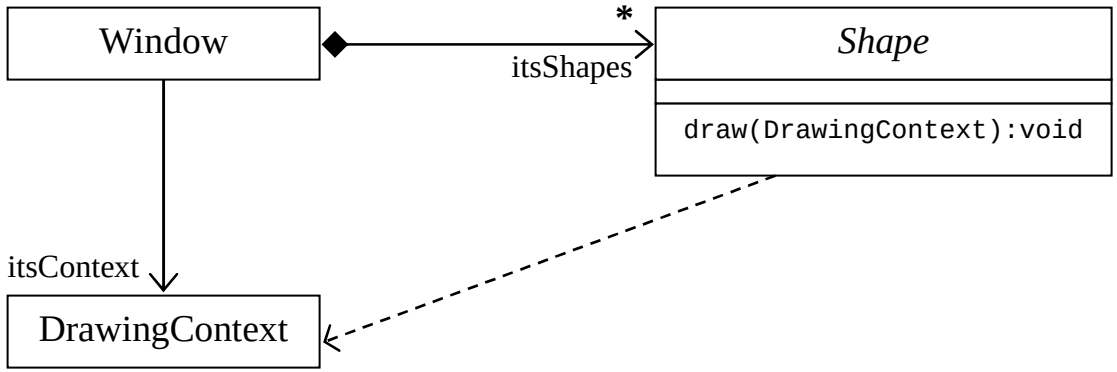
The difference is one of implication. Aggregation denotes whole/part relationships whereas associations do not. However, there is not likely to be much difference in the way that the two relationships are implemented. That is, it would be very difficult to look at the code and determine whether a particular relationship ought to be aggregation or association. For this reason, many designers choose to ignore the aggregation relationship altogether.

©2012 Christopher Hield

UML Class Diagrams

Dependency

Sometimes the relationship between a two classes is very weak. They are not implemented with member variables at all. Rather they might be implemented as member function arguments. Consider, for example, the Draw function of the Shape class. Suppose that this function takes an argument of type DrawingContext.

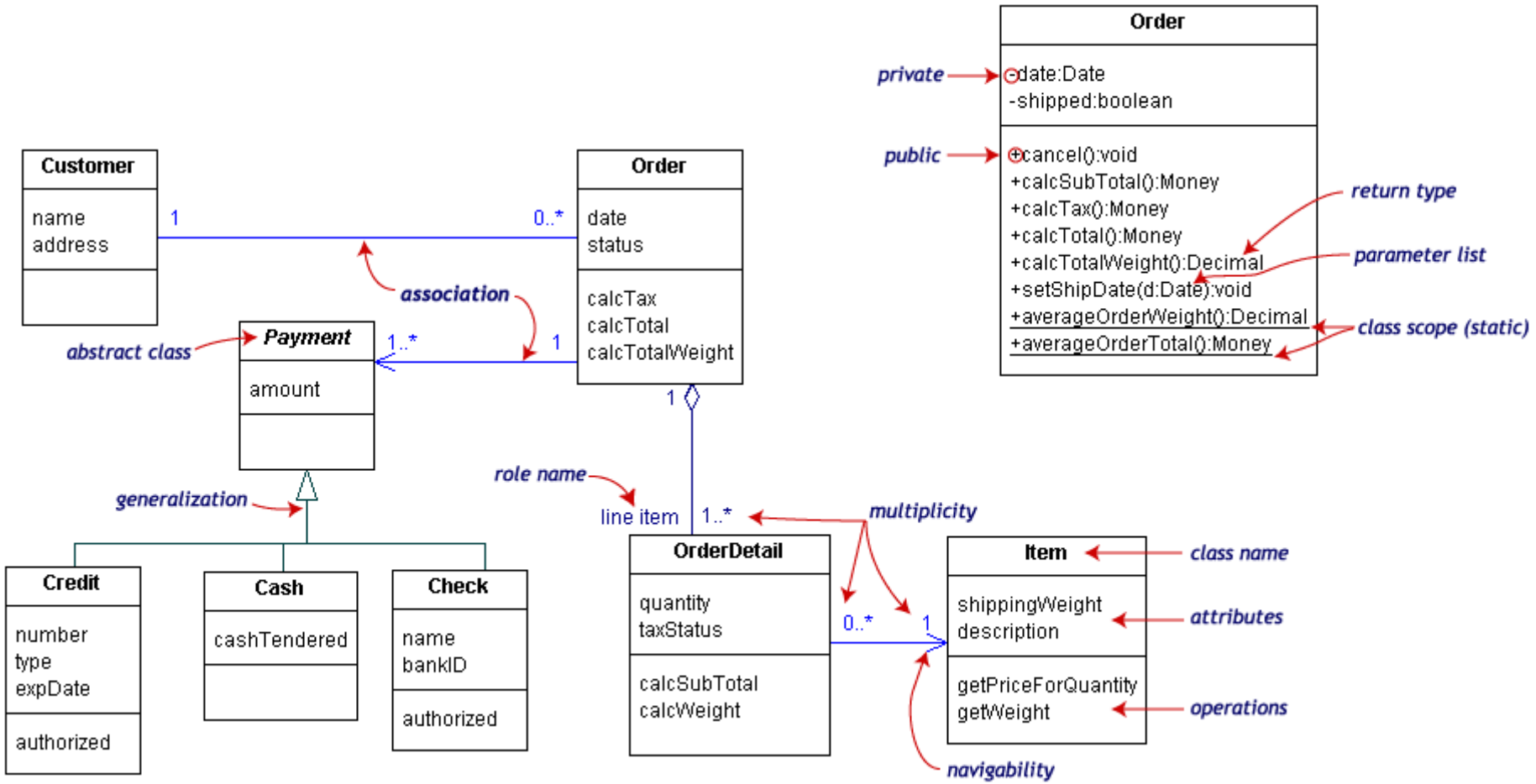


The dashed arrow between the Shape class and the DrawingContext class is the dependency relationship. This is sometimes called a “using” relationship. This relationship simply means that Shape somehow depends upon (contains some reference to) DrawingContext.

©2012 Christopher Hield

UML Class Diagrams

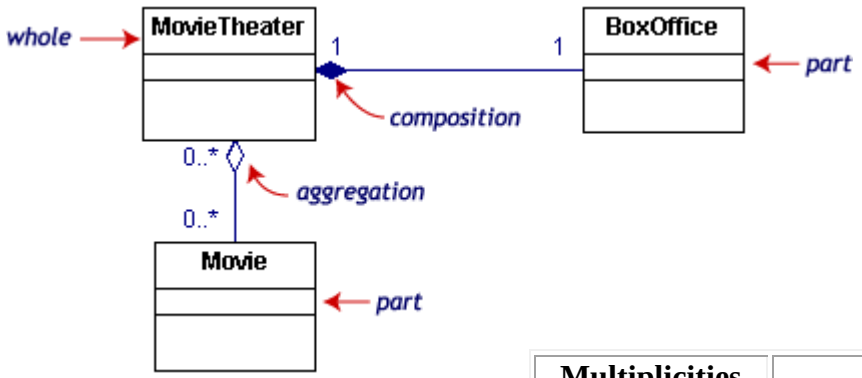
Notation Summary – The Basics



©2012 Christopher Hield

UML Class Diagrams

Notation Summary – The Aggregation & Composition

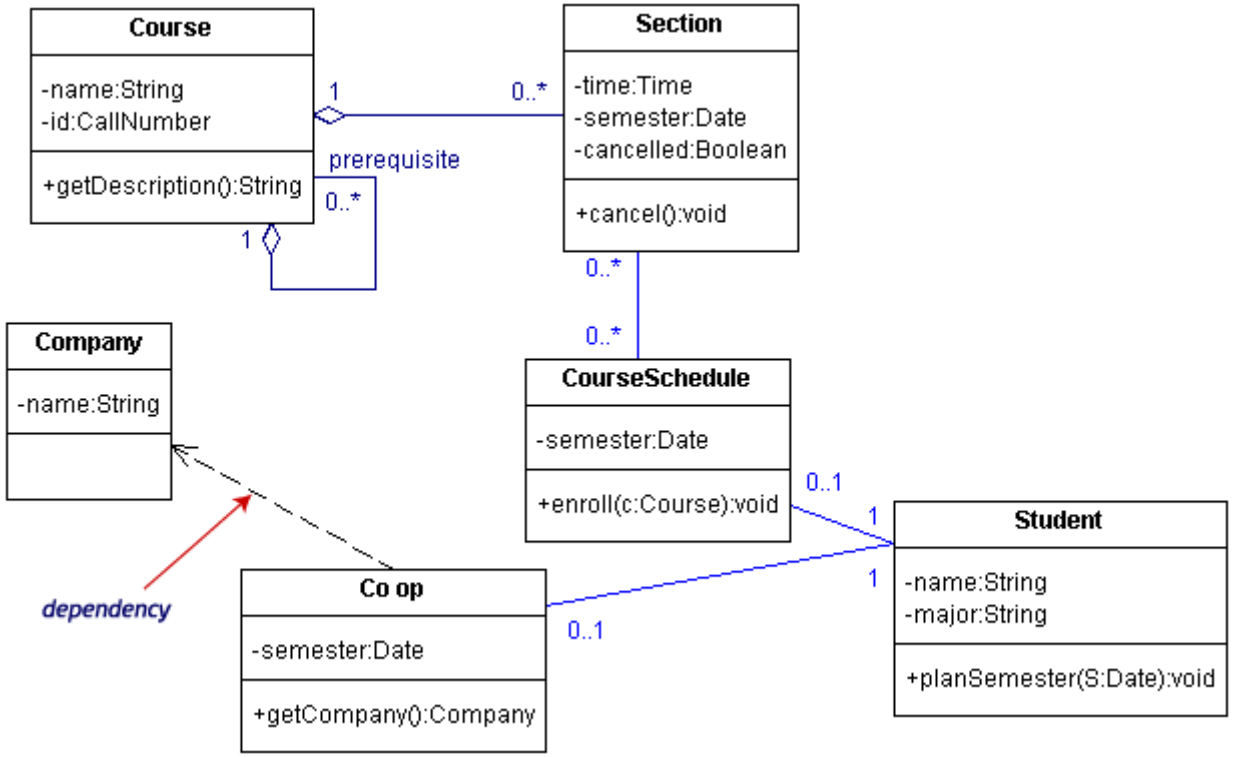


Multiplicities	Meaning
1	exactly one instance (zero instances not allowed)
n	exactly "*n*" instances
0..1	zero or one instance.
n..m	indicates ***n*** to ***m*** instances (i.e., 2..5, 4..10, etc.)
0..* *or* *****	no limit on the number of instances (including none).
1..*	at least one instance but no limit on the number of instances

©2012 Christopher Hield

UML Class Diagrams

Notation Summary – Additional Details



©2012 Christopher Hield

UML Class Diagrams

Large class diagrams

Inevitably, if you are modeling a large system or a large business area, there will be numerous entities you must consider. Instead of modeling every entity and its relationships on a single class diagram, it is better to use multiple class diagrams. Dividing a system into multiple class diagrams makes the system easier to understand, especially if each diagram is a graphical representation of a specific part of the system.

Placing Notes and Comments on Class Diagrams

Typically, class diagrams are accompanied by comments or notes about each class, and/or its relationships, attributes, and operations. These notes can be placed directly on the class diagram, or, better yet, associated with each level (e.g. class, attribute, etc.).

©2012 Christopher Hield

UML Class Diagrams

Summary

UML class diagrams model static class relationships that represent the fundamental architecture of the system. Note that these diagrams describe the relationships between classes, not those between specific objects instantiated from those classes. Thus the diagram applies to all the objects in the system. A class diagram consists of the following features:

- Classes: These titled boxes represent the classes in the system and contain information about the name of the class, fields, methods and access specifiers. Abstract roles of the class in the system can also be indicated.
- Interfaces: These titled boxes represent interfaces in the system and contain information about the name of the interface and its methods.
- Generalization: a solid line with a solid arrowhead that points from a sub-class to a superclass or from a sub-interface to its super-interface.
- Realization: a dotted line with a solid arrowhead that points from a class to the interface that it implement
- Association: a solid line with an open arrowhead that represents a "has a" relationship. The arrow points from the containing to the contained class. Associations can be one of the following two types or not specified.
- Composition: Represented by an association line with a solid diamond at the tail end. A composition models the notion of one object "owning" another and thus being responsible for the creation and destruction of another object.
- Aggregation: Represented by an association line with a hollow diamond at the tail end. An aggregation models the notion that one object uses another object without "owning" it and thus is not responsible for its creation or destruction.
- Dependency -- a dotted line with an open arrowhead that shows one entity depends on the behavior of another entity. Typical usages are to represent that one class instantiates another or that it uses the other as an input parameter.

©2012 Christopher Hield