UML Tutorial:
Part 1 -- Class Diagrams.

Robert C. Martin

My next several columns will be a running tutorial of UML. The 1.0 version of UML was released on the 13th of January, 1997. The 1.1 release should be out before the end of the year. This column will track the progress of UML and present the issues that the three amigos (Grady Booch, Jim Rumbaugh, and Ivar Jacobson) are dealing with.

Introduction

UML stands for Unified Modeling Language. It represents a unification of the concepts and notations presented by the three amigos in their respective books[1]. The goal is for UML to become a common language for creating models of object oriented computer software.

In its current form UML is comprised of two major components: a Meta-model and a notation. In the future, some form of method or process may also be added to; or associated with, UML.

The 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.

A deeper discussion of the metamodel is beyond the scope of this column. Interested readers can learn more about it by downloading the UML documents from the rational web site[2].

The 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.

In this article we will present some of the aspects of the static modeling notation. Static models are presented in diagrams called: Class Diagrams.

Class Diagrams.

The purpose of a class diagram is to depict the classes within a model. In an object oriented application, classes have attributes (member variables), operations (member functions) and relation-

---

1. Object Oriented Analysis and Design, Grady Booch, Benjamin Cummings, 1994.
   Object Oriented Modeling and Design, James Rumbaugh, et. al., Prentice Hall, 1991
   Object Oriented Software Engineering, Ivar Jacobson, et. al., Addison Wesley, 1992
2. http://www.rational.com

ships with other classes. The UML class diagram can depict all these things quite easily. The fundamental element of the class diagram is an icon the represents a class. This icon is shown in Figure 1.
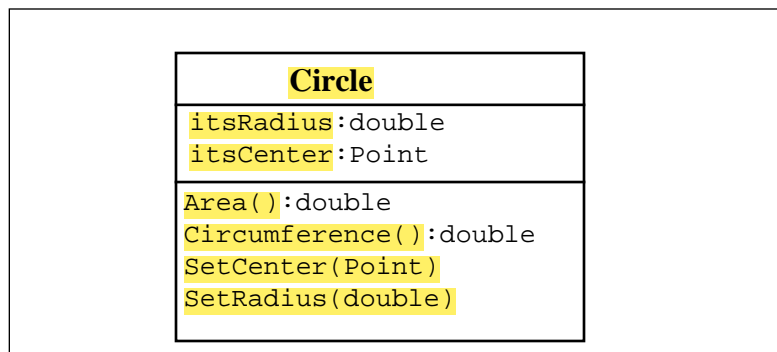
Figure 1: The Class Icon

Class
Attribute
operation()
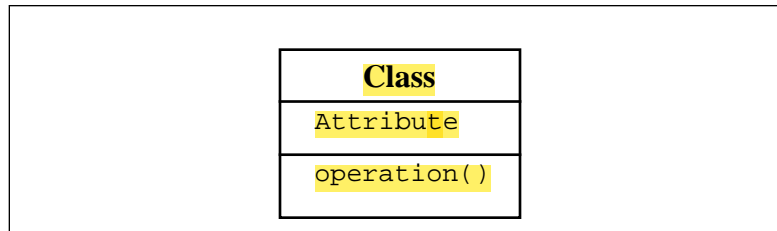
A class icon is simply a rectangle divided into three compartments. The topmost compartment contains the name of the class. The middle compartment contains a list of attributes (member variables), and the bottom compartment contains a list of operations (member functions). In many diagrams, the bottom 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. Figure 2 shows a typical UML description of a class that represents a circle.

Figure 2: Circle class

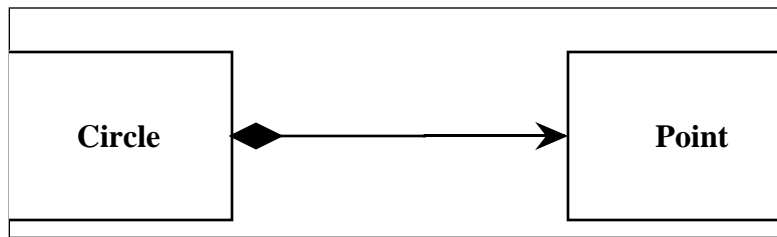| Circle |
| --- |
| itsRadius:double
itsCenter:Point |
| Area():double
Circumference():double
SetCenter(Point)
SetRadius(double) |

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. I could have named them too, and used colons to separate them from their types; or I could have omitted the arguments altogether.

Composition Relationships

Each instance of type `Circle` seems to contain an instance of type `Point`. This is a relationship known as *composition*. It can be depicted in UML using a class relationship. Figure 3 shows the composition relationship.

The black diamond represents composition. It is placed on the `Circle` class because it is 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, `Point` does not know about `Circle`. In UML relationships are presumed to be bidirectional unless the arrowhead is present to restrict them. Had I omitted the arrowhead, it would have meant that `Point` knew about `Circle`. At the code level, this would imply a `#include "circle.h"` within `point.h`. For this reason, I tend to use a *lot* of arrowheads.

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. For those of you who are familiar with the Booch-94 notation, this is the Has-by-value relationship.

In C++ we would represent this as shown in Listing 1.

Listing 1: Circle class

```
class Circle
{
  public:
    void SetCenter(const Point&);
    void SetRadius(double);
    double Area() const;
    double Circumference() const;
  private:
    double itsRadius;
    Point itsCenter;
};
```

In this case we have represented the composition relationship as a member variable. We could also have used a pointer so long as the destructor of `Circle` deleted the pointer.

Inheritance

The inheritance relationship in UML is depicted by a peculiar triangular arrowhead. This arrowhead, that looks rather like a slice of pizza, points to the base class. One or more lines proceed from the base of the arrowhead connecting it to the derived classes.

Figure 4 shows the form of the inheritance relationship. In this diagram 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 pure virtual.

Figure 4: Inheritance
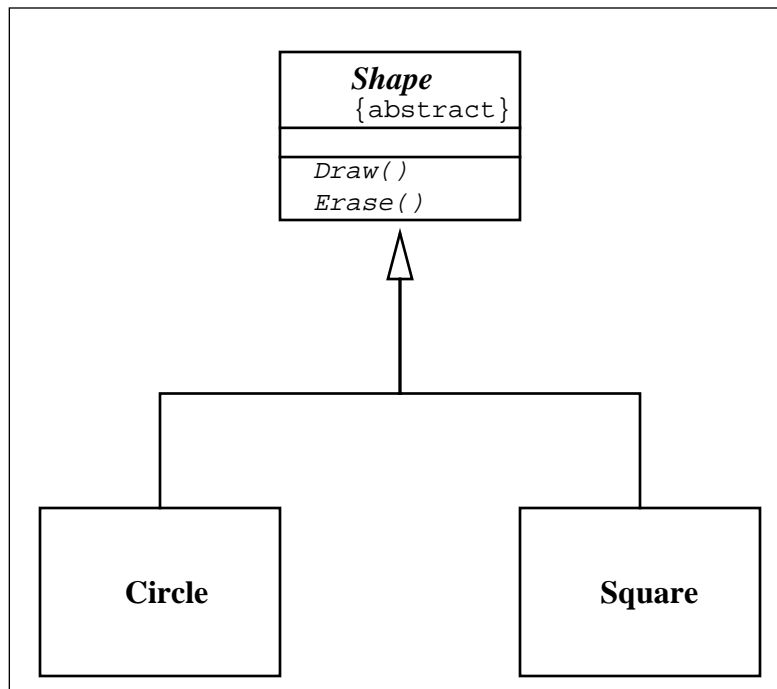
Italics are not always very easy to see. Therefore, as shown in Figure 4, an abstract class can also be marked with the `{abstract}` property. What's more, though it is not a standard part of UML, I will often write `Draw()=0` in the operations compartment to denote a pure virtual function.

Aggregation / Association

The weak form of aggregation is 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.
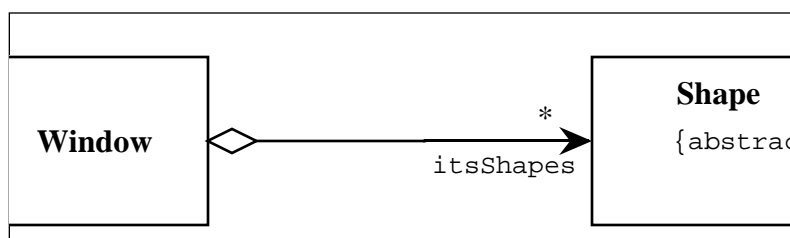

Figure 5: Aggregation

Figure 5 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 role at the `Shape` end of the aggregation is marked with a "`*`". This 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`.

Listing 2 shows how Figure 5 might be implemented in C++

Listing 2: Window contains Shapes

```
class Window
{
  public:
    //...
  private:
    vector<Shape*> itsShapes;
};
```

There are other forms of containment that do not have whole / part implications. For example, Each `Window` refers back to its parent `Frame`. This is not aggregation since it is not reasonable to consider a parent `Frame` to be part of a child `Window`. We use the association relationship to depict this.
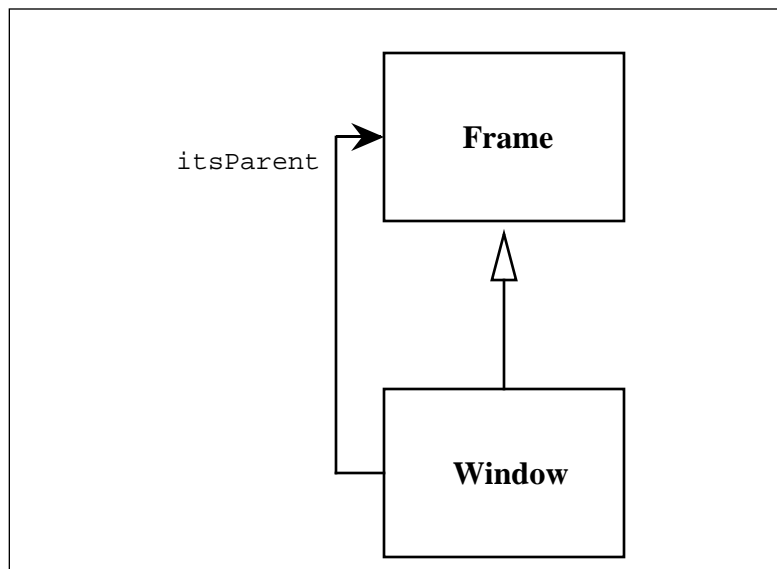
Figure 6: Associations

Figure 6 shows how we draw an association. An association is nothing but a line drawn between the participating classes. In Figure 6 the association has an arrowhead to denote that `Frame` does not know anything about `Window`. Once again note the name on the role. This relationship will almost certainly be implemented with a pointer of some kind.

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, it is pretty safe to ignore the aggregation relationship altogether. As the amigos said in the UML 0.8 document: "...if you don't understand [aggregation] don't use it."

Aggregation and Association both correspond to the Has-by-reference relationship from the Booch-94 notation.

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`.
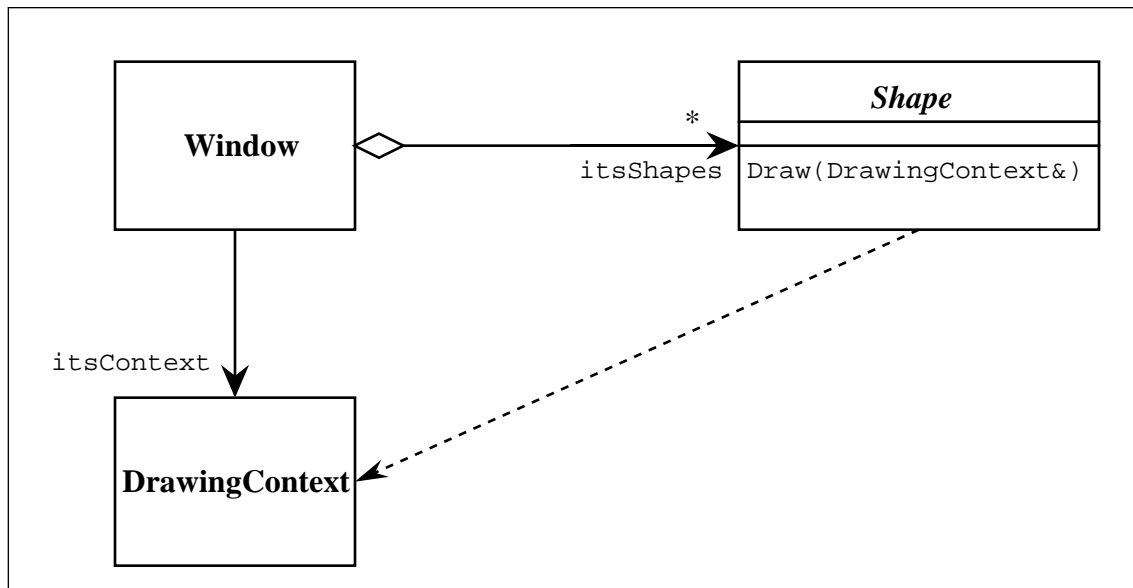
Figure 7: Dependency

Figure 7 shows a dashed arrow between the `Shape` class and the `DrawingContext` class. This is the dependency relationship. In Booch94 this was called a 'using' relationship. This relationship simply means that `Shape` somehow depends upon `DrawingContext`. In C++ this almost always results in a `#include`.
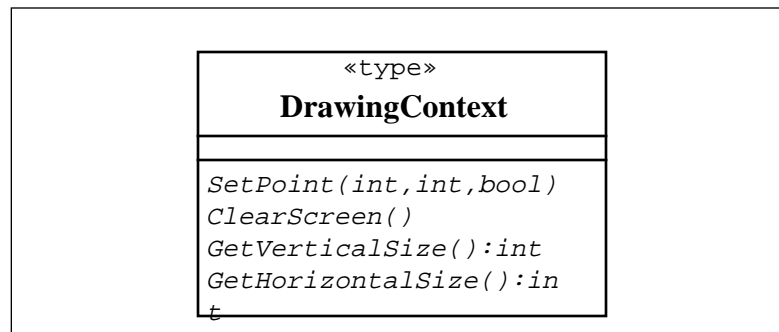
Interfaces

There are classes that have nothing but pure virtual functions. In Java such entities are not classes at all; they are a special language element called an `interface`. UML has followed the Java example and has created some special syntactic elements for such entities.

The primary icon for an interface is just like a class except that it has a special denotation called a stereotype. Figure 8 shows this icon. Note the «`type`» string at the top of the class. The two surrounding characters "«»" are called guillemets (pronounced Gee-**may**). A word or phrase surrounded by guillemets is called a "stereotype". Stereotypes are one of the mechanisms that can be used to extend UML. When a stereotype is used above the name of a class it indicates that this class is a special kind of class that conforms to a rather rigid specification.

The «`type`» stereotype indicates that the class is an interface. This means that it has no member variables, and that all of its member functions are pure virtual.
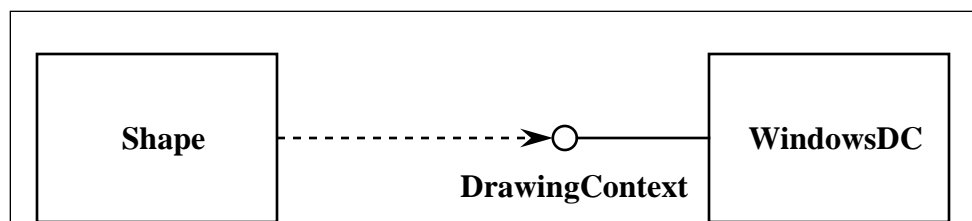
UML supplies a shortcut for «type» classes. Figure 9 shows how the "lollypop" notation can be used to represent an interface. Notice that the dependency between `Shape` and `DrawingContext` is shown as usual. The class `WindowsDC` is derived from, or conforms to, the `Drawingcontext` interface. This is a shorthand notation for an inheritance relationship between

Figure 8: Type class

```
              «type»
          DrawingContext

  SetPoint(int,int,bool)
  ClearScreen()
  GetVerticalSize():int
  GetHorizontalSize():in
  t
```

WindowsDC and DrawingContext.

Figure 9: Interface Lollypop

```
  ┌───────────┐                        ┌───────────┐
  │           │                        │           │
  │   Shape   │ - - - - - - - - ->O───│ WindowsDC │
  │           │      DrawingContext    │           │
  └───────────┘                        └───────────┘
```

Conclusion

In this article we have explored a few of the notational elements that UML supplies for static software design. In future columns we will expand upon this notation by showing how to use it for solving some real software problems. We will also examine UML's contingent of tools for modeling dynamic design.