Common UML Symbols and Relationships

4002-218

Paul Walter

Table of Contents

Common UML Symbols

Term	Criteria	UML Example	Implementation

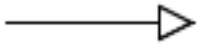
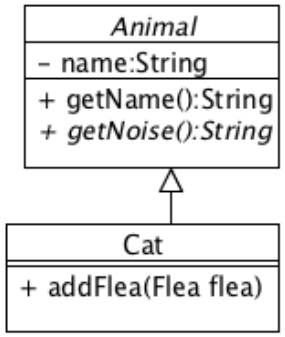
Class
 	
Class Name
attribute1
…
attributeN
method1
…
methodN
	A concept you identified in during the Discovery phase as an object you wish to have in your software system.	Storage
		- value: int
		+ setValue(int)
		+ getValue(): int
		
		<u>Notes</u>:
		
		"-" Means Private
		"+" Means Public
		"#" Means Protected
			`public class Storage`
			`{`
			`  private int value;`
			`  public void setValue(int i)`
			`  {`
			`  value = i;`
			`  }`
			`  public int getValue()`
			`  {`
			`  return value;`
			`  }`
			`}`

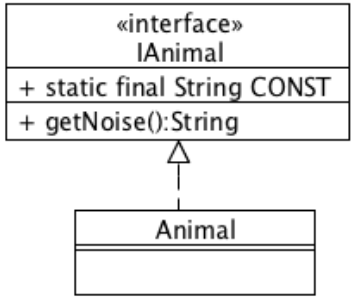
Abstract Class
 	
Abstract Class
- attribute 1
…
- attribute N
+ regularMethod():String
+ *abstractMethod():String*
 	
Notice the Italics in the title and method name that indicate an abstract class/method.
	1) When you have a generalized form of a concept that doesn't make sense to instantiate (like Animal).
	
	2) When you have a candidate class that you wish to define some default behavior for, but not all of it: the class that extends it should implement some.
		Animal: notice that it's name is italic, as well as the method "getNoise()". This means they are both abstract.
		
		Animal
		- name:String
		+ getName():String
		+ *getNoise():String*
			`public abstract class Animal`
			`{`
			`  private String name;`
			`  public String getname()`
			`  {`
			`  return name;`
			`  }`
			
			`  public abstract String getNoise();`
			
			`}`

Interface
 	
Same format as Class, but has **<<iinterface>>** at the top of the box.
	Use this to ensure particular behavior exists in any object that implements it.	«interface»
		IAnimal
		+ static final String CONST
		+ getNoise():String
			`/* Interface */`
			`public interface IAnimal`
			`{`
			`  public static final String CONST =`
			`"const";`
			
			`  public String getNoise();`
			`}`

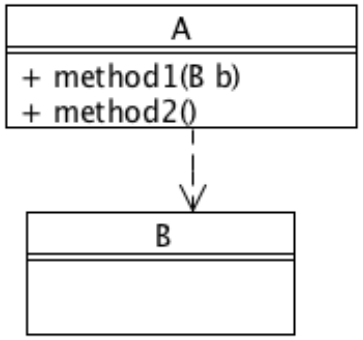
Relationships

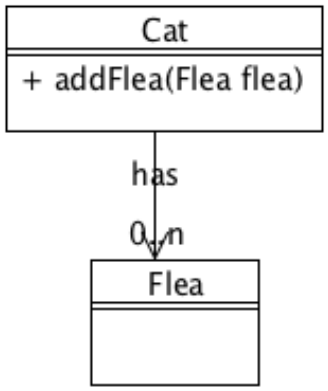
Term	Criteria	UML Example	Implementation

Generalization
(Inheritance)
 	
"Is a"
 	
 
	Used to describe one class as being a more specialized form of another; when one class extends another:
	
	"A Cat **is a**N Animal"
			`/* Superclass */`
			`public abstract class Animal`
			`{`
			`  private String name;`
			`  public String getName()`
			`  {`
			`  return name;`
			`  }`
			`  //abstract class`
			`  public abstract String`
			`  getNoise();`
			`}`
			
			`/* Subclass */`
			`public class Cat extends Animal`
			`{`
			`  public String getNoise()`
			`  {`
			`  return "meow!";`
			`  }`
			`}`

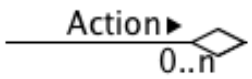
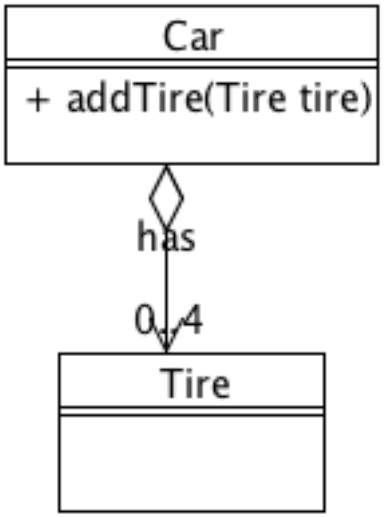
Realization
(Implementing an Interface)
 	
"Is a"
 	
 
 	
"Realization" is the UML specific term when an object implements an Interface.
	When you want to ensure functionality in an object without implementing it.		`/* Interface */`
			`public interface IAnimal`
			`{`
			`  public static final String CONST =`
			`"const";`
			
			`  public String getNoise();`
			`}`
			`/* Class */`
			`public class Cat implements IAnimal`
			`{`
			`  public String getNoise()`
			`  {`
			`  return "meow!";`

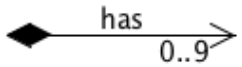
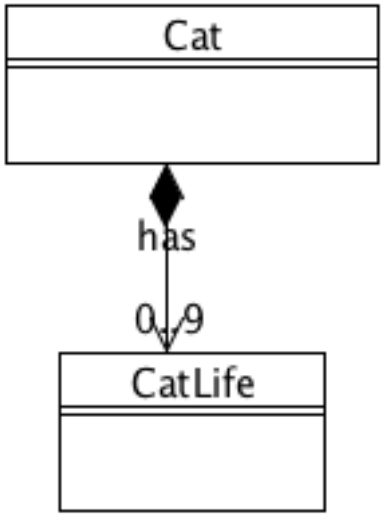
Notice it is an arrow with the dashed line.			`}` `}`

Dependency
 	
"Uses"
 	
Notice the arrow points to the object that **DOSEN'T** have the reference (see Uni and Bi-Directional relationships, below, for reasons why).
	Use this when you don't need a permanent reference to an object
	
	Example: when you pass it into a method only to use it as a local variable.
	
	Another example is when you use objects (like System.out) in the main method; they aren't referenced, they are just being called.
	
	"RunCat **uses** System.out.println to print out messages ."
			`import B;`
			
			`public class A {`
			
			`public void method1(B b) {`
			`// B will be used locally`
			`...`
			`}`
			
			`public void method2() {`
			`// local var`
			`B tempB = new B();`
			`...`
			`}`
			`}`

Association
 	
"Has a" / "References"
 	
Action ▸
0..n 0..1
	Use this when you need a permanent object reference;
	
	When the objects being grouped don't **conceptually** create a whole;
	
	And that the objects can exist independently.
		
		Note: the arrow at the end of the
			`import java.util.*;`
			
			`public class Cat`
			`{`
			`private List<Flea> fleas;`
			`public Cat(String name)`
			`{`
			`super(name);`
			`fleas = new`
			`ArrayList<Flea>();`
			`}`
			
			`public void addFlea(Flea flea)`
			`{`
			`fleas.add(flea);`
			`}`

		association line indicates that this relationship is one way: Cat knows about Flea but not vice versa.	`}`

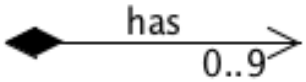
Aggregation
 	
"Has a"
 	
 
 	
	Use this when you need a permanent object reference;
	
	When the objects being grouped **conceptually** create a whole;
	
	And that those objects can exist independently.
		
		
		Note: the little arrow opposite the clear diamond on the Aggregation symbol which means this is a uni-directional relationship.
			`import java.util.*;`
			`public class Car`
			`{`
			`  private List<Tire> tires= new ArrayList<Tire>();`
			`  public void addTire(Tire tire)`
			`  {`
			`  tires.add(tire);`
			`  }`
			`}`
			
			`public class Tire`
			`{`
			`}`

Composition
 	
"Has a"
 	
 
	Use this when you need a permanent object reference;
	
	When the objects being grouped **conceptually** create a whole;
	
	And that those objects **cannot** exist independently. When the host dies, so do the objects that are grouped with it.
			`import java.util.*;`
			`public class Cat`
			`{`
			`private List<CatLife> lives;`
			`public Cat()`
			`{`
			`lives= new ArrayList<CatLife>();`
			`lives.add(new CatLife(1));`
			`...`
			`lives.add(new CatLife(9));`
			`}`
			`}`
			
			`public class CatLife`
			`{`
			`...`
			`}`

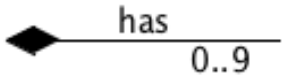
Gotchas

Unidirectional vs. Bidirectional relationships

Uni-directional



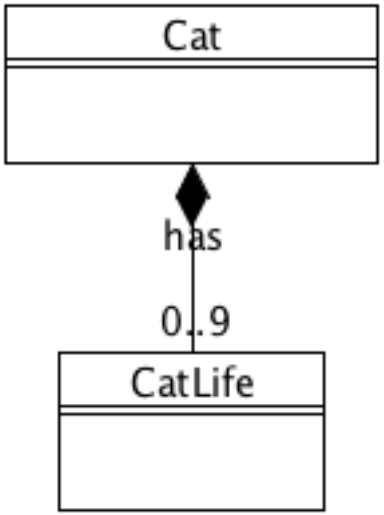
Bi-directional



Notice that the right hand side of the relationship line has an arrow (opposite the black diamond): This indicates that the relationship is only one way: there is only one reference.

If the relationship doesn't have this distinction, then it is assumed that both objects participating in the relationship have a reference to each other.

In next column over we are going to explore what a bi-directional relationship looks like.



Note: This is a bi-directional relationship: notice the lack of an arrow opposite the black diamond.

```
 import java.util.*;
public class Cat
{
   private List<CatLife> lives;
   public Cat()
   {
     lives= new
ArrayList<CatLife>();
     lives.add(new CatLife(this));
     ...
     lives.add(new CatLife(this));
   }
}

public class CatLife
{
   private Cat cat;
   public CatLife(Cat cat)
   {
      this.cat = cat;
   }
}
```