UML

Diagram Types

- structural-gives us the structural relationships between system
- behavioral-concerned with the execution of the system

Class Model Diagram

- most popular
- also called static model
- it is an example of a structured diagram (shows system structures
- shows classes and relationships

UML classes

- 3 compartments separated by horizontal lines

Counter
- count : int
+increment():void +display():void +set(aCount : int=0):void

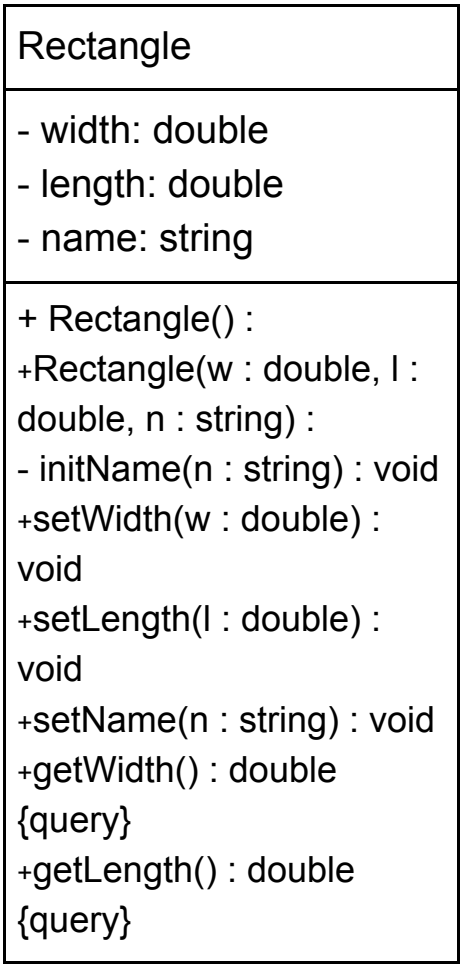
Name of the class
attributes of the class (instance variables)
methods and operations the class provides

another formal example:

An ADT representing a rectangle:

```
class Rectangle
{
 private:
```

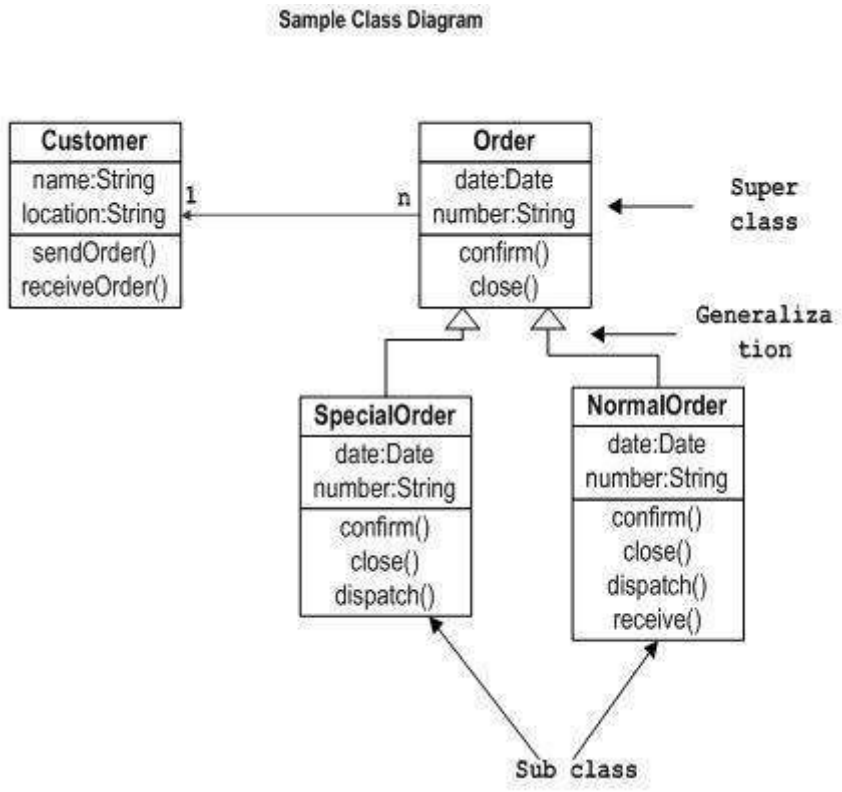
```
   double width;
   double length;
   string name;
   void initName(string n);
 public:
 //constructors
 Rectangle();
 Rectangle(double, double,
             string);
 //destructor
   ~Rectangle();
   void setWidth(double);
   void setLength(double);
   void setName(string);
   double getWidth() const;
   double getLength() const;
};
```



UML Relationships

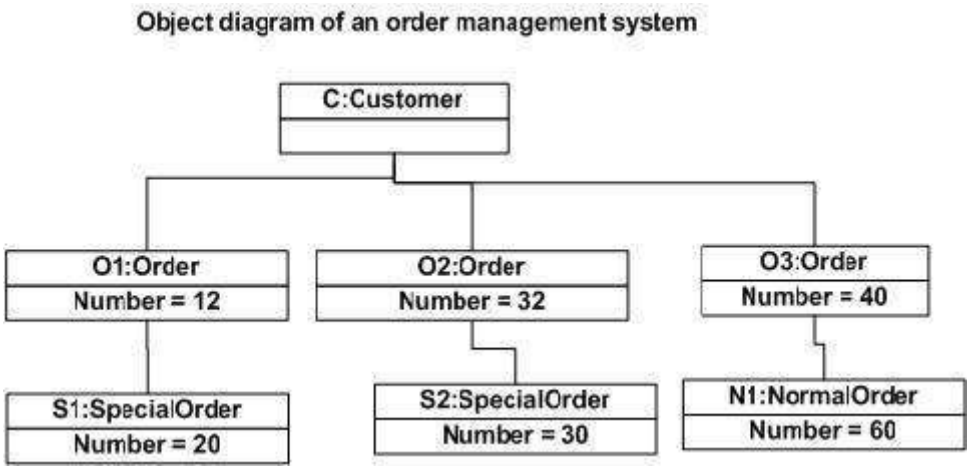
- dependency: x uses y - - - - - - > (dashed direct line)
- associations: x affects Y _____ (solid undirected line)
- generalization: x is a kind of y ____>(solid with open arrowhead)

Example of class diagram

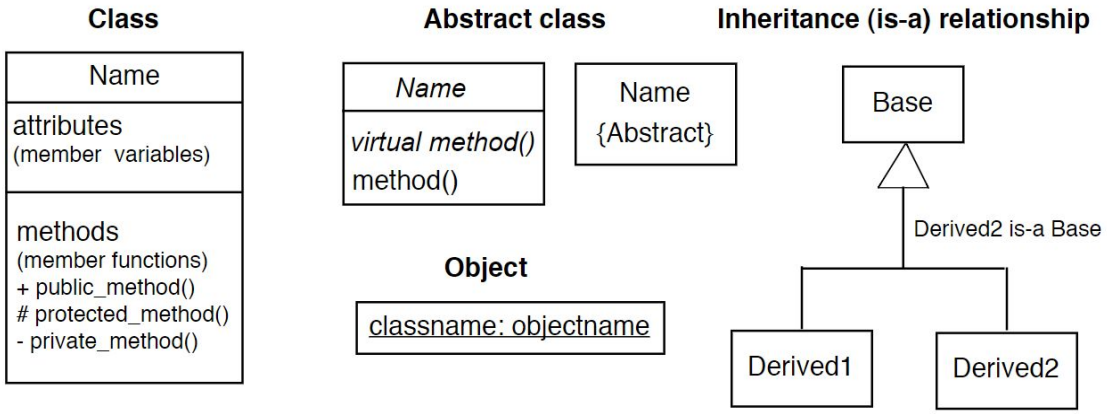


Object Diagram

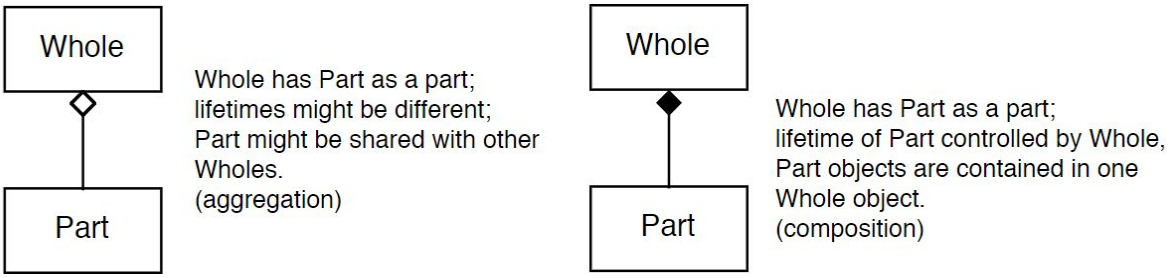
- conveys objects and links instead of classes and relationships
- name is composed by the class name and the specific instance
- for example in the first box the class is customer and the instance is c, they are separated by a colon
- some instances also have their attribute fields with values filled in (example in the class order,the instance o1 has the attribute number = 12



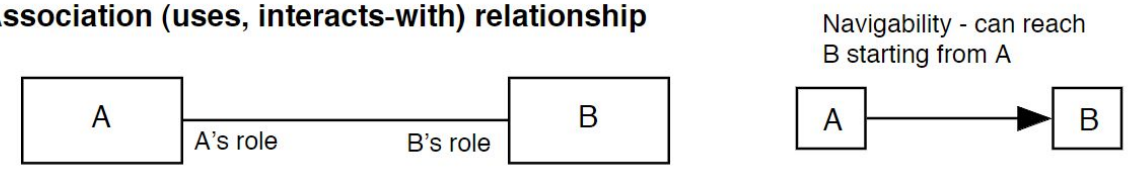
Class diagram summary



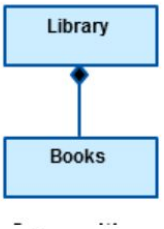
Aggregation and Composition (has-a) relationship



Association (uses, interacts-with) relationship



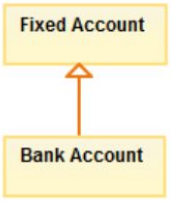
Composition



Composition

The composition relationship is very similar to the aggregation relationship. with the only difference being its key purpose of emphasizing the dependence of the contained class to the life cycle of the container class. That is, the contained class will be obliterated when the container class is destroyed. For example, a shoulder bag's side pocket will also cease to exist once the shoulder bag is destroyed.

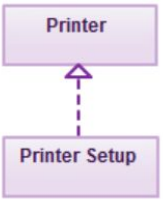
Inheritance / Generalization



Inheritance

refers to a type of relationship wherein one associated class is a child of another by virtue of assuming the same functionalities of the parent class. In other words, the child class is a specific type of the parent class. To show inheritance in a UML diagram, a solid line from the child class to the parent class is drawn using an unfilled arrowhead.

Realization



Realization

denotes the implementation of the functionality defined in one class by another class. To show the relationship in UML, a broken line with an unfilled solid arrowhead is drawn from the class that defines the functionality of the class that implements the function. In the example, the printing preferences that are set using the printer setup interface are being implemented by the printer.

Multiplicity in Aggregation, Composition, or Association

* - any number	0..1 - zero or one	Follow line from start class to end class,
1 - exactly 1	1..* - 1 or more	note the multiplicity at the end.
n - exactly *n*	*n* .. *m* - *n* through *m*	Say “Each <start> is associated with <multiplicity> <ends>”

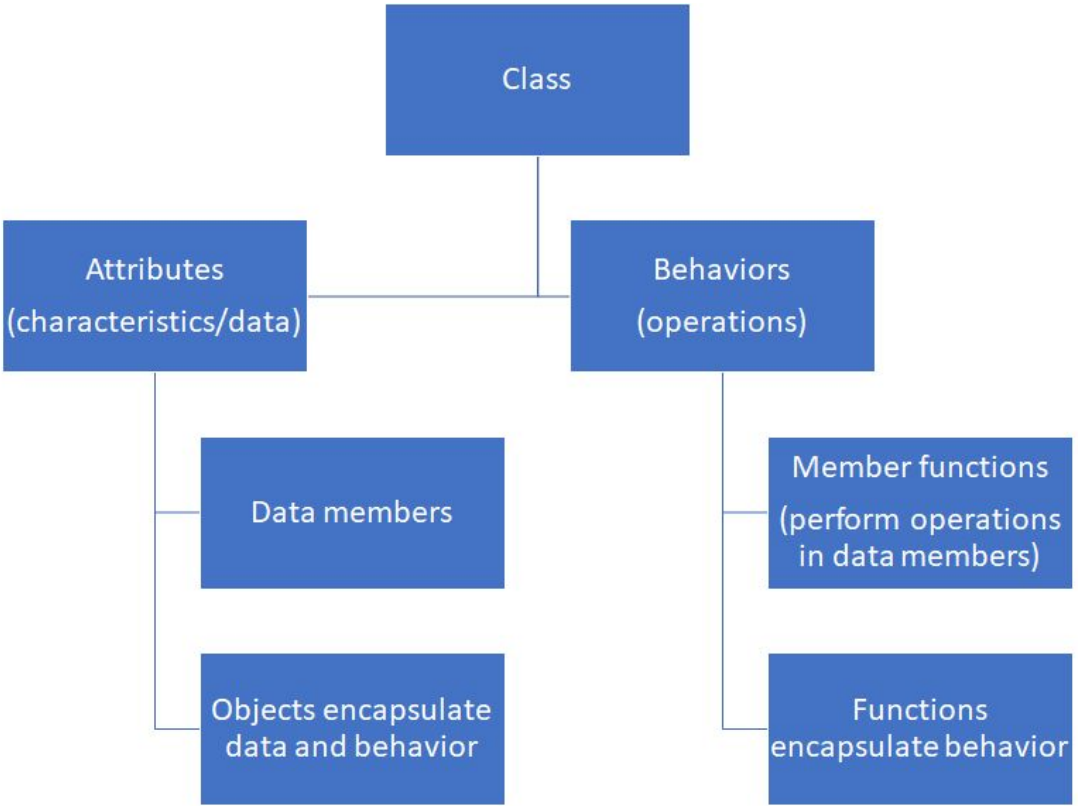
Review on terminology

- A solution is said to be efficient if it solves a problem within required resource constraints (example total space or time to perform a task
- the steps to select a data structure are:
    - Analyze your problem to determine the basic operations that must be supported.Examples of basic operations include inserting a data item into the data structure, deleting a data item from the data structure, and finding a specified data item.
    - Quantify the resource constraints for each operation.
    - Select the data structure that best meets these requirements.

- the questions regarding the operation constraints should be:
    - Are all data items inserted into the data structure at the beginning, or are insertions interspersed with other operations? Static applications (where the data are loaded at the beginning and never change) typically require only simpler data structures to get an efficient implementation than do dynamic applications.
    - Can data items be deleted? If so, this will probably make the implementation more complicated

    - Are all data items processed in some well-defined order, or is search for specific data items allowed? “Random access” search generally requires more complex data structures

- A type is a collection of values (for example an int is a simple type because its value contains no subparts)
    - an aggregate/composite type contains several pieces of data (example a bank account-address, number, name, etc)
- A data item is a piece of info whose value is drawn from a type (it is a member of a type)
- A data type is a type together with a collection of operations to manipulate the type. For example, an integer variable is a member of the integer data type. Addition is an example of an operation on the integer data type.
- An ADT is the realization of a data type as a software component. The interface of the ADT is defined in terms of a type and a set of operations on that type. The behavior of each operation is determined by its inputs and outputs. An ADT does not specify how the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as encapsulation.
- A data structure is the implementation for an ADT
    - ADT+implementation=class
    - operations associated to ADT are implemented by member functions
    - variables that define space required by data item are data members
    - an object is an instance of a class
- Object oriented analysis: identifying the problem and coming up with a solution that consists of the interaction between classes of objects
- A module is a unit of code, it can be a function/method/class



Problems, Algos and Programs

- problem: task to be performed
- function: matching between inputs(domain) and outputs (range)
- values of the input are parameters of a function
- specific selection of values for parameters are the instance of the problem
- an algorithm is a method or a process followed to solve a problem, if the problem is viewed as a function, then the algorithm is an implementation of the function that transforms an input to the corresponding output

Something is an algorithm if it has the following properties:
1. It must be correct. In other words, it must compute the desired function, converting each input to the correct output. Note that every algorithm implements some function, because every algorithm maps every input to some output (even if that output is a program crash). At issue here is whether a given algorithm implements the intended function.
2. It is composed of a series of concrete steps. Concrete means that the action described by that step is completely understood — and doable — by the person or machine that must perform the algorithm. Each step must also be doable in a finite amount of time. Thus, the algorithm gives us a “recipe” for solving the problem by performing a series of steps, where each such step is within our capacity to perform. The ability to perform a step can depend on who or what is intended to execute the recipe. For example, the steps of

a cookie recipe in a cookbook might be considered sufficiently concrete for instructing a human cook, but not for programming an automated cookiemaking factory.
3. There can be no ambiguity as to which step will be performed next. Often it is the next step of the algorithm description. Selection (e.g., the if statement in C++) is normally a part of any language for describing algorithms. Selection allows a choice for which step will be performed next, but the selection process is unambiguous at the time when the choice is made.
4. It must be composed of a finite number of steps. If the description for the algorithm were made up of an infinite number of steps, we could never hope to write it down, nor implement it as a computer program. Most languages for describing algorithms (including English and “pseudocode”) provide some way to perform repeated actions, known as iteration. Examples of iteration in programming languages include the while and for loop constructs of C++. Iteration allows for short descriptions, with the number of steps actually performed controlled by the input.
5. It must terminate. In other words, it may not go into an infinite loop.

Writing Pseudocode

```
Program MakeTea:
      Organize needed supplies;
      Plug kettle;
      Put teabag in cup;
      Put water in kettle;
      Add water to cup;
      Remove teabag;
      if (sugar is required)
            then add sugar;
      else
            don't add sugar;
      Serve;
```

Sequence

When writing a program we assume that the computer executes the program starting at the beginning and working to the end- IT IS CALLED SEQUENCE

```
statement1;
statement2;
statement3;
```

Selection

making a cup of tea with or without sugar

```
if (sugar is required)
      then add sugar;
else
      don't add sugar;
```

checking which number is biggest

```
if (a>b)
      then print a + "is bigger";
else
      print b + "is bigger";
```

Iteration

keep doing something until a condition is verified-loop

```
While (kettle is not full)
      do keep filling kettle;
```

print numbers 1 to 5

```
while (a<5)
      do print a;
            a=a+1;
```

read a number, check if it is odd or even

```
Program oddOrEven:
Read A;
If (A/2 gives remainder)
      then print "it is odd)
else
      print "it is even"
```

check if number is prime

```
Program Prime
      Read A;
      B = A-1;
      isPrime=True;
```

```
while (b!=)
do if(a/b gives no remainder)
      then isPrime=false;
      b =b-1;
if (isPrime==true)
      then print "prime
else
      print "not prime"
```

New, Delete, ->, this, notes

```
/*
 * for successful use of dynamic memory allocation
 * keep the following in mind
 *
 * use new/delete for single variables of all types
 * use new[]/delete[] for array variables of all types
 * new allocates memory and delete deallocates it
 * clean up all pointer member variables in the destructor
 *
 * new first allocates memory and then calls the constructor
 * delete first calls the destructor and then deallocates memory
 *
 * cant pass arguments to constructor with new[]
 *
 *
 */

 /*
  * example 1
  *
  * //floatDynamic is a pointer variable dynamically allocated to a float
  * float *floatDynamic = new float (23.3);
  * cout << "dynamically assigned float has value = "<< *floatDynamic<<endl;
  * delete floatDynamic;
  *
  */

 /*
  * example 2
  *
  * //dynamically allocate memory and construct an object
  * ComplexNumber *cDynamic = new ComplexNumber(10,15);
  * cout << "Printing out dynamically allocated object " << endl;
  * //use arrow operator to access members of a class through pointer
```

```
* //it dereferences a pointer and used the dot operator on that address
* //so in this case this is the same as:
* //(*cDynamic).print();
* cDynamic->print()
* delete cDynamic;
*
*
* reminder for the ComplexNumber constructor
*
* ComplexNumber(double c, double r):realPart(r), complexPart(c)
* {
*     cout << "inside 2 argument constructor"<< endl;
*  }
*
*  the this keyword can be used inside any member function of an object
*  to refer to itself as if that object were any other variable
*
*  example 3
*
*  //using this
*  void printThis()
*  {
*     cout << "The address of this object is: "<< this << endl;
*     cout << "real part "<< this -> realPart<< "imaginary part ="<< this->complexPart<< endl;
*  }
*
*  so, if I had this:
*  cPlaceNew->printThis();
*  cout<<cPlacedNew<<endl;
*
*  it would just print the address of the variable in printThis()
*  this points to the same address in both cases
*
*
*/
```

References

- can be said to be pointers in disguise
- no need to dereference references
- no need to allocate or deallocate memory (no need to use new or delete)
- references can’t be null
- a reference must always be initialized
- there can be multiple references to the same value
- reference reassignments

```
#include <iostream>
using namespace std;

int main() {

int x=5;
int z=3;

//create reference y and assign x to it
int& y=x;
cout << "x= "<< x<<flush<< " y= "<<y<<flush<<" z= "<<z<<endl;

//references can't be reassigned so it is assigning the value
//of z to (3) to x via the reference y
y=z;

//change value of y
y=10;
cout << "x= "<< x<<flush<< " y= "<<y<<flush<<" z= "<<z<<endl;
//x will be the reassigned variable and not z

cout << "x= "<< x<<flush<< " y= "<<y<<flush<<" z= "<<z<<endl;

return 0;
}
/*
x= 5 y= 5 z= 3
x= 10 y= 10 z= 3
x= 10 y= 10 z= 3

*/
```

example:

```
#include <iostream>
using namespace std;

int main() {
 int x = 5;

 //when the value in y is changed, the value in x will also change
 int& y = x;//int reference or reference to an int

 cout << "this is the initial value of x: "<< x << " and y: "<< y <<endl;

 y=10;
```

```
 cout<< "changing value of y to "<< y <<endl;

 cout<<"this makes the value of x: "<< x <<endl;

 return 0;
}
/*
output
this is the initial value of x: 5 and y: 5
changing value of y to 10
this makes the value of x: 10
*/
```

swapping 2 values using references

```
#include <iostream>
using namespace std;

//swapping two values using references
double swap(int& a, int& b)
{
 int temp = a;
 a=b;
 b=temp;
}
int main() {

 int a = 5;
 int b = 10;

 cout << "value of a: "<<a<<" and b: "<<b<<" before calling swapper"<<endl;

 swap(a,b);
 cout << "value of a: "<<a<<" and b: "<<b<<" after calling swapper"<<endl;

 return 0;
}
/*
output
value of a: 5 and b: 10 before calling swapper
value of a: 10 and b: 5 after calling swapper
*/
```

Local memory

```
// Local storage example
int Square(int num) {
     int result ;
     result = num * num;
     return result;
}
```

- both num and result are locals
- lifetime refers to the time between memory allocation and deallocation of a variable
- lifetime of a local variable is tied to the function it is local to
- parameters (line num here) start out with a value copied from the caller
- local variables start with random initial values

Algorithm Analysis

- the most critical resource is often running time but running time alone should not be the focus
    - space required to run
    - time required for instantiation of an algo into a program
    - space required for the data structure