Advance OOPS concepts

Agenda

- Programming
- Procedural programming
- Object oriented programming.
- Features of OOP
- OOP concepts
- Object oriented programming design principles

Programming

- *Programming* is the craft of transforming requirements into something that computer can execute.

Procedural programming

- Programmer implements requirement by breaking down them to small steps (functional decomposition).

- Programmer creates the “recipe” that computer can understand and execute.

Procedural programming …..

- What’s wrong with procedural programming language?
- When requirements change
  - It hard to implement new feature that were not planned in the beginning.
  - Code blocks gets bigger and bigger.
  - Changes in code introduce many bugs.
  - Code gets hard to maintain.

Worst thing is that

Requirement **always change**

Object oriented programming

- Break down requirements into objects with responsibilities, not into functional steps.
- Embraces change of requirements.
  - By minimizing changes in code.
- Lets you think about object hierarchies and interactions instead of program control flow.
- A completely different programming paradigm.

Why OOPS?

- To modularize software development, just like any other engineering discipline.
- To make software projects more manageable and predictable.
- For better maintainability, since software maintenance costs were more than the development costs.
- For more re-use code and prevent ‘reinvention of wheel’** every time.

Features of OOP

- Emphasis on data rather on procedure.
- Programs are divided into what are known as "objects".
- Functions that operate on data of an object are tied together in a data structure.
- Object may communicate with each other through functions.
- New data and functions can be added easily whenever necessary.

OOPS Concepts

- Classes and Objects
- Message and Methods
- Encapsulation
- Association, Aggregation and Composition
- Inheritance
- Polymorphism
- Abstraction
- Modularity
- Coupling

Classes and Objects

- Object oriented programming uses objects.
- An object is a thing, both tangible and intangible. Account, Vehicle, Employee etc.
- To create an object inside a compute program we must provide a definition for objects – how they behave and what kinds of information they maintain – called a class.
- An object is called an instance of a class.
- Object interacts with each other via message.

Message and Methods

- To instruct a class or an object to perform a task, we send message to it.
- You can send message only to classes and objects that understand the message you sent to them.
- A class or an object must posses a matching method to be handle the received message.
- A method defined for a class is called class method, and a method defined for an object is called an instance method.

Message Passing

- The process by which an object:
  - Sends data to other objects
  - Asks the other object to invoke the method.
- In other words, object talks to each other via messages.

Encapsulation

- Encapsulation is the integration of data and operations into a class.
- Encapsulation is hiding the functional details from the object calling it.
- **Can you drive the car?**
  - Yes, I can!
- **So, how does acceleration work?**
  - Huh?
- Details encapsulated (hidden) from the driver.

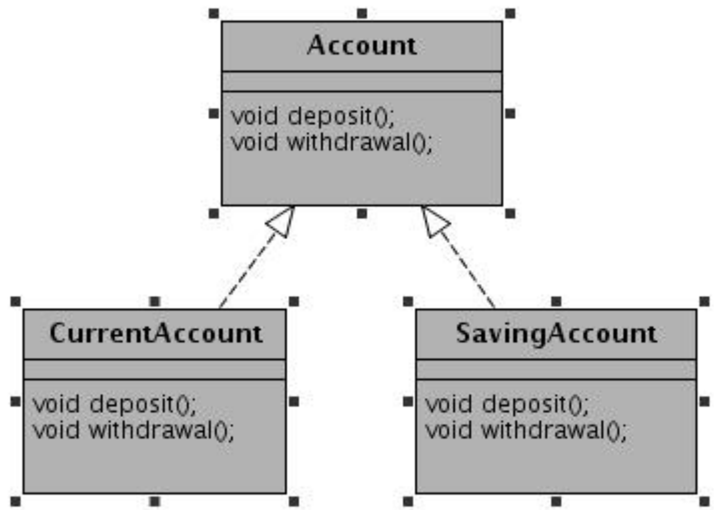
Association, Aggregation and Composition

- **Association →** Whenever two object are related with each other the relationship is called ***association*** between objects.
- **Aggregation**
  → ***Aggregation*** is specialized

Inheritance

- Inheritance is a mechanism in OOP to design two or more entities that are different but share many common features.
  - Feature common to all classes are defined in the superclass.
  - The classes that inherit common features from the superclass are called subclasses.

Inheritance Example



Why inheritance?

- Classes often share capabilities.
- We want to avoid re-coding these capabilities.
- Reuse of these would be best to
  - Improve maintainability
  - Reduce cost
  - Improve “real world” modeling.

Why Inheritance? Benefits

- No need to re-invent the wheel.
- Allow us to build on existing codes without having to copy it, paste it or rewrite it again, etc.
- To create the subclass, we need to program only the differences between the superclass and subclass that inherits from it.
- Make class more flexible.

Composition(has-a)/Inheritance(is-a)

- Prefer composition when not sure about inheritance.
- Prefer composition when not all the superclass functions were re-used by subclass.
- Inheritance leads to tight coupling b/w subclass with superclass. Harder to maintain.
- Inheritance hides some of compilation error which must be exposed.
- Inheritance is easier to use than composition.
- Composition make the code maintainable in

Composition/Inheritance…..

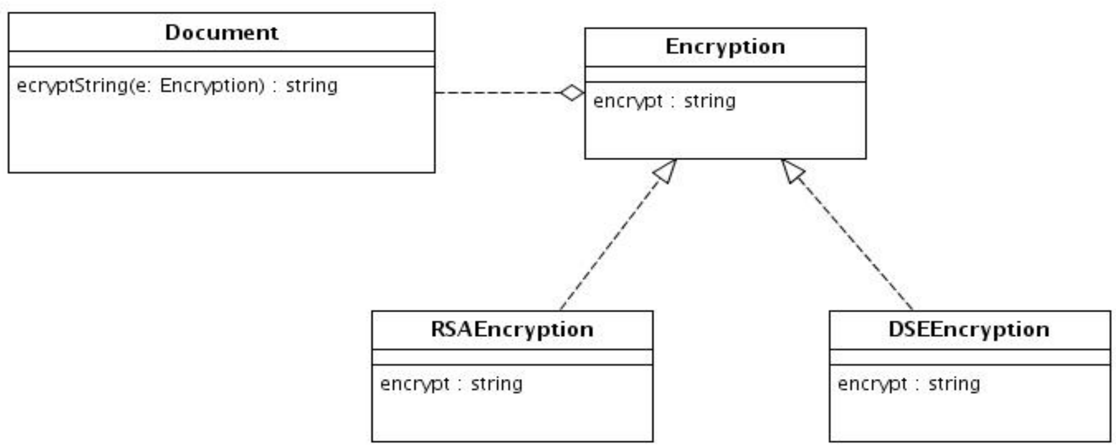
- Idea is to think twice while making decision.
- One has to have proper reason while choosing composition/inheritance.
- A car has “engine”.
- A car is a “vechicle”.
- Discussion?

Polymorphism

- Polymorphism indicates the meaning of "many forms".
- Polymorphism present a method that can have many definitions. Polymorphism is related to "overloading" and "overriding".
- Overloading indicates a method can have different definitions by defining different type of parameters.
  - getPrice() : void
  - getPrice(string name) : void

Polymorphism….

- Overriding indicates subclass and the parent class has the same methods, parameters and return type(namely to redefine the methods in parent class).



Abstraction

- Abstraction is the process of modeling only relevant features
  - Hide unnecessary details which are irrelevant for current for current purpose (and/or user).
- Reduces complexity and aids understanding.
- Abstraction provides the freedom to defer implementation decisions by avoiding commitments to details.

Abstraction example

```
#include <iostream>
using namespace std;
class Adder{
public:
      // constructor
     Adder(int i = 0)
    {
         total = i;
    }
    // interface to outside world
   void addNum(int number)
   {
       total += number;
   }
   // interface to outside world
   int getTotal()
   {
        return total;
   };
private:
    // hidden data from outside world
    int total;
};
```

```
int main( )
{
      Adder a;

     a.addNum(10);
     a.addNum(20);
     a.addNum(30);

     cout << "Total " << a.getTotal()
<<endl;
     return 0;
}
```

Modularity

- The modularity means that the logical components of a large program can each be implemented separately. Different people can work on different classes. Each implementation task is isolated from the others.
- This has benefits, not just for organizing the implementation, but for fixing problems later.

Coupling

- Coupling defines how dependent one object on another object (that is uses).
- Coupling is a measure of strength of connection between any two system components. The more any one components knows about other components, the tighter(worse) the coupling is between those components.

Tight coupling

```
class Traveler
{
   Car c=new Car();
   void startJourney()
   {
     c.move();
   }
}
```

```
class Car
{
  void move()
  {
    // logic...
  }
}
```

Loose coupling

```
class Traveler
{
    Vehicle v;
    public void setV(Vehicle v)
    {
      this.v = v;
    }

    void startJourney()
    {
      v.move();
    }
}

Interface Vehicle
{
   void move();
}
```

```
class Car implements Vehicle
{
    public void move()
    {
        // logic
    }
}

class Bike implements Vehicle
{
    public void move()
    {
        // logic
    }
}
```

Cohesion

- Cohesion defines how narrowly defined an object is. Functional cohesion refers measures how strongly objects are related.
- Cohesion is a measure of how logically related the parts of an individual components are to each other, and to the overall components. The more logically related the parts of components are to each other higher (better) the cohesion of that component.
- Low coupling and tight cohesion is good

Interface

- An interface is a contract consisting of group of related function prototypes whose usage is defined but whose implementation is not:
  - An interface definition specifies the interface's member functions, called methods, their return types, the number and types of parameters and what they must do.
  - These is no implementation associated with an interface.

Interface Example

```
class shape
 {
public:
    virtual ~shape();
    virtual void move_x(distance x) = 0;
    virtual void move_y(distance y) = 0;
    virtual void rotate(angle rotation) =
0;
    //...
};
```

Interface implementation

- An interface implementation is the code a programmer supplies to carry out the actions specified in an interface definition.

Implementation Example

```
class line : public shape
{
public:
     virtual ~line();
    virtual void move_x(distance x);
    virtual void move_y(distance y);
    virtual void rotate(angle rotation);
private:
    point end_point_1, end_point_2;
   //...
};
```

Interface vs. Implementation

- Only the services the end user needs are represented.
  - Data hiding with use of encapsulation
- Change in the class implementation should not require change in the class user’s code.
  - Interface is still the same
- Always provide the minimal interface.
- Use abstract thinking in designing interfaces
  - No unnecessary steps
  - Implement the steps in the class implementation

How to determine minimum possible interface?

- Only what user absolutely needs
  - Fewer interfaces are possible
  - Use polymorphism
- Starts with hiding everything (private)
  - Only use public interfaces (try not to use public attributes, instead get/set).
- Design your class from users perspective and what they need (meet the requirements)