OOP 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.
- Let 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.

reinventing the wheel** is a phrase that means to duplicate a basic method that has already previously been created or optimized by others

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
- Cohesion
- Interfaces, Implementation?

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.
- A value we pass to an object when sending a message is called an argument of the message.

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.

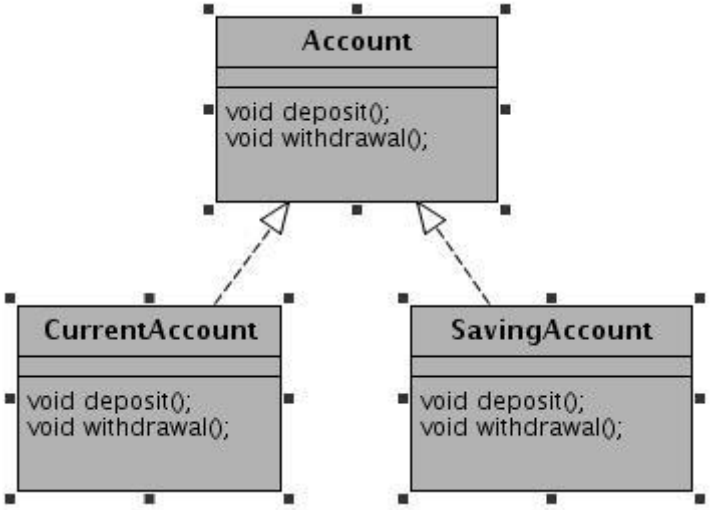
Association, Aggregation and Composition

- **Association →** Whenever two object are related with each other the relationship is called ***association*** between objects.
- **Aggregation →** ***Aggregation*** is specialized form of ***association.*** In ***aggregation objects*** have their own life-cycle but there is ownership and child object can not belongs to another parent object. But this is only an ownership not the life-cycle control of child control through parent object. **Ex:** Student and teacher, Person and address etc.
- **Composition →** ***Composition*** is again ***specialize*** form of ***aggregation*** and we can call this as a "***life and death***" relationship. It is a strong type of ***aggregation***. Child object dose not have their life-cycle and if parent object deletes all child object will also be deleted. **Ex:** House and room

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 future, especially when your assumption breaks (Using inheritance).
- Discussion is incomplete without discussion of Liskov substitution principle.

Composition/Inheritance…..

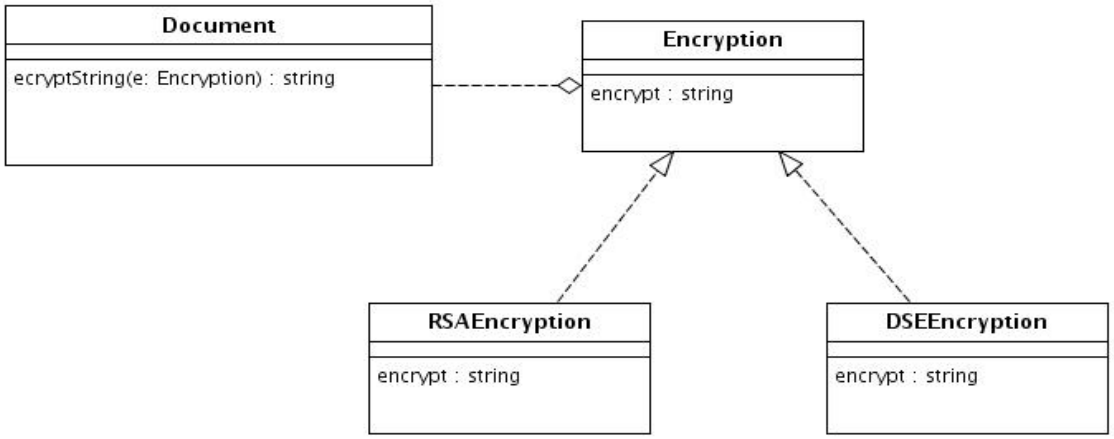
- Idea is to think twice while making decision.
- One has to have proper reason while choosing composition/inheritance.
- A car has “engine”.
- A car is a “vechicle”.
- Discussion?

Polymorphism

- Polymorphism indicates the meaning of “many forms”.
- Polymorphism presents 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 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 object oriented design (OOD).

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)

Object oriented programming design principles

- **Principles of class design:**
  - Single responsibility principle (SRP)
  - Open close principle (OCP)
  - Liskov substitution principle (LSP)
  - Dependency inversion principle (DIP)
  - Interface segregation principle (ISP)
- **Principles of package cohesion**
  - Reuse release equivalence principle (REP)
  - Common closure principle (CCP)
  - Common reuse principle (CRP)
- **Principles of package coupling**
  - Acyclic dependency principle (ADP)
  - Stable dependencies principle (SDP)
  - Stable abstractions principle (SAP)

Single responsibility principle

- Each responsibility should be a separate class, because each responsibility is an axis of change.
- A class have one and only one reason to change.

Open close principle

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
- In other words, (in an ideal world...) you should never need to change existing code or classes: All new functionality can be added by adding new subclasses or methods, or by reusing existing code through delegation.
- This prevents you from introducing new bugs in existing code. If you never change it, you can't break it.
- Ex. Draw shapes etc.

Open close principle …..

- When a single change to a program results in a cascade of changes to dependent modules, that program exhibits the undesirable attributes that we have come to associate with “bad” design. The program becomes fragile, rigid, unpredictable and un-reusable. The open-closed principle attacks this in a very straightforward way. It says that you should design modules that never change. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

OCP Example

Procedural solution for square/circle problem

```
enum ShapeType {circle, square};
struct Shape
{
      ShapeType itsType;
};
struct Circle
{
      ShapeType itsType;
      double itsRadius;
      Point itsCenter;
};
struct Square
{
      ShapeType itsType;
      double itsSide;
      Point itsTopLeft;
};
```

```
//
// These functions are implemented elsewhere
//
void DrawSquare(struct Square*)
void DrawCircle(struct Circle*);
typedef struct Shape *ShapePointer;
void DrawAllShapes(ShapePointer list[], int n)
{
      int i;
      for (i=0; i<n; i++)
      {
            struct Shape* s = list[i];
            switch (s->itsType)
            {
            case square:
                  DrawSquare((struct Square*)s);
                  break;
            case circle:
                  DrawCircle((struct Circle*)s);
                  break;
            }
      }
}
```

OOD solution to Square/Circle problem.

```
class Shape
{
public:
virtual void Draw() const = 0;
};
class Square : public Shape
{
public:
virtual void Draw() const;
};
class Circle : public Shape
{
public:
virtual void Draw() const;
};
void DrawAllShapes(Set<Shape*>& list)
{
for (Iterator<Shape*>i(list); i; i++)
(*i)->Draw();
}
```

Liskov substitution principle

- Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
- What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for **all** programs P defined in terms of T, the behavior of P is **unchanged** when o1 is substituted for o2 then S is a subtype of T.
- Ex. Rectangle and Square etc.

LSP Example

- Rectangle and Square (Violation of LSP).

```
class Rectangle
{
public:
virtual void SetWidth(double w)
{
                    itsWidth=w;
}
virtual void SetHeight(double h)
{
                    itsHeight=h;
}
double GetHeight() const
 {
                    return itsHeight;
}
double GetWidth() const {
                    return itsWidth;
}
private:
                    double itsHeight;
                    double itsWidth;
};
```

```
class Square : public Rectangle
{
public:
          virtual void SetWidth(double w);
           virtual void SetHeight(double h);
};
void Square::SetWidth(double w)
{
           Rectangle::SetWidth(w);
           Rectangle::SetHeight(w);
}
void Square::SetHeight(double h)
{
           Rectangle::SetHeight(h);
            Rectangle::SetWidth(h);
}
void g(Rectangle& r)
{
           r.SetWidth(5);
           r.SetHeight(4);
           assert(r.GetWidth() *
r.GetHeight()) == 20);
}
```

Dependency inversion principle

- High level modules should not depend upon low level modules. Both should depend upon abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.
- What is bad design?
  - *Rigid* (Hard to change due to dependencies. Especially since dependencies are transitive.)
  - *fragil* (Changes cause unexpected bugs.)
  - *immobile* (Difficult to reuse due to implicit dependence on current application code.)

DIP Example

Copy Program

```
void Copy()
{
    int c;
    while ((c =
ReadKeyboard()) != EOF)
        WritePrinter(c);
}
```

Enhanced Copy program

```
void Copy(outputDevice dev)
{
    int c;
    while ((c = ReadKeyboard())
!= EOF)
        if (dev == printer)
            WritePrinter(c);
        else
            WriteDisk(c);
}
```

The OO Copy Program

```
class Reader
{
public:
    virtual int Read() = 0;
};
class Writer
{
public:
    virtual void Write(char) = 0;
};
void Copy(Reader& r, Writer& w)
{
    int c;
    while((c=r.Read()) != EOF)
        w.Write(c);
}
```

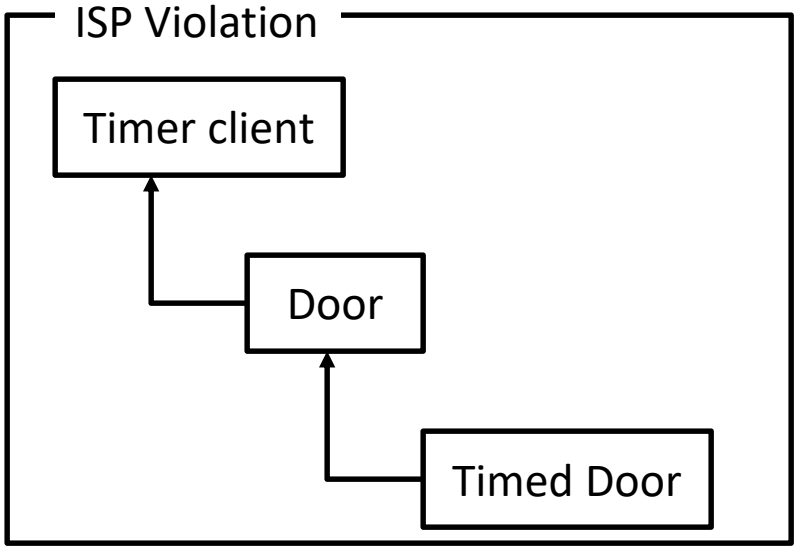
Interface segregation principle

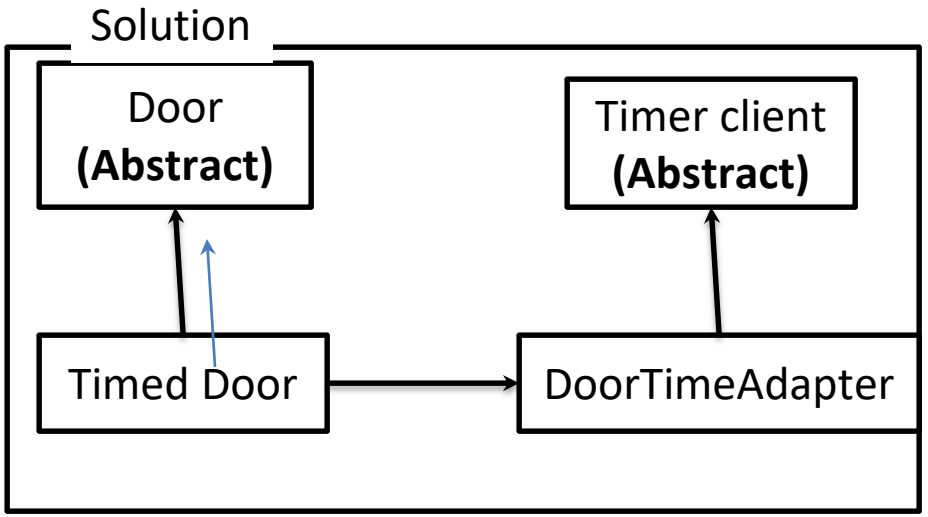
- Client should not forced to depend on methods that they do not use.
- The ISP says that once an interface has become too 'fat‘ it needs to be split into smaller and more specific interfaces so that any clients of the interface will only know about the methods that pertain to them

ISP Example

```
class Door
{
public:
    virtual void Lock() = 0;
    virtual void Unlock() = 0;
    virtual bool IsDoorOpen()
= 0;
};
```

```
class Timer
{
public:
    void Regsiter(int timeout,
          TimerClient* client);
};
class TimerClient
{
public:
      virtual void TimeOut() = 0;
};
```





References

- For more principles visit http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign

Questions?

Thanks!!!
Happy Learning!!!