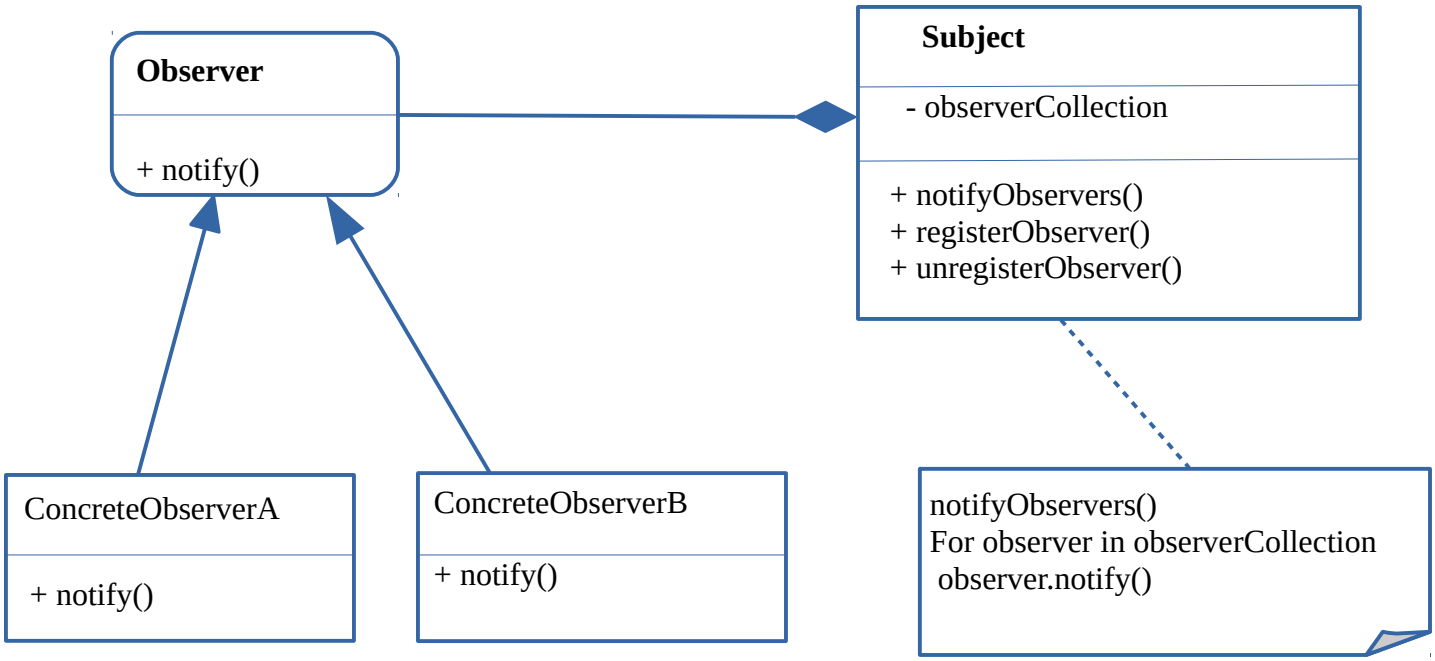
Observer Pattern

It defines one to many dependency between objects so that when one object changes state,all its dependents are notified and updated automatically.

Use-case:-

* A change to one object implies a change to another object.
* One abstraction depends on the state of the another abstraction.
* Objects should be notified of the state changes of another object without being tightly coupled.



Subject
* Manages its collection of observers
* Allows the observers to register and unregister them self.

Observer
* Defines an interface to notify the observers

Concrete Observer
* Implements the interface
* Is notified by the subject

```
#include <iostream>
#include <string>
#include <list>

// Class Observer
class Observer{
    public:
        virtual void notify() const
        {

        }
        virtual ~Observer()
        {
        }
};

// Class Subject

class Subject {
    public:
        void resgisterObserver(Observer* observer) {
            observers.push_back(observer);
        }
        void unregisterObserver(Observer* observer) {
            observers.remove(observer);
        }
        void notifyObserver() {
            for(auto x : observers)
            {
                x→notify();
            }
        }
    private:
        std::list<Observer *> observers;
};
```

// class ConcreteObserverA

```
class ConcreteObserverA : public Observer {
    private:
        Subject& m_subject;
    public:
        ConcreteObserverA(Subject& subject) : m_subject(subject){
            m_subject. resgisterObserver(this);
        }
        void notify() const {
          cout<<"ConcreteObserverA::notify"<<endl;
        }
};
```

// class ConcreteObserverB

```
class ConcreteObserverB : public Observer {
        private:
            Subject& m_subject;
        public:
            ConcreteObserverB(Subject& subject) : m_subject(subject) {
                m_subject.registerObserver(this);
            }
            void notify() const{
              cout<<"ConcreteObserverB::notify"<<endl;
            }
};

int main() {
  Subject subject;

   ConcreteObserverA observerA(subject);
   ConcreteObserverB observerB(subject);

    subject.notifyObservers();
    cout<<"subject.unregisterObserver(ObserverA)"<<endl;
   subject. unregisterObserver( ObserverA);
   subject.notifyObservers();

 }
```

REAL TIME EXAMPLE

When you are creating cabin information application,In which multiple ground stations are responsible for collecting cabin information and you want to create system where multiple displays for can show real time updates. When ground station collects new data all registered displays are should be updated automatically with the latest information.

```
#include <iostream>
#include <vector>

class CabinStation;

class Observer
{
    public:
        virtual void update(CabinStation* m_cabin) = 0;
};

class Gen7Device : public Observer
{
   private:
     string m_name;
   public:
      Gen7Device(string name):m_name(name)
      { }
       void update(CabinStation* m_cabin) override;
 };
 class Gen8Device : public Observer
 {
    private:
        string m_name;
    public:
         Gen8Device(string name):m_name(name)
         { }
         void update(CabinStation* m_cabin) override;
  };
```

```
class CabinStation
{
    private:
      string name;
      int id;
      vector<Observer*>m_observers;
    public:
      CabinStation(int id):id(id),name("default opera tion")
      { }
      int getId()
      {
         return id;
      }
      string getName()
      {
         return name;
      }
       void attach(Observer* observer)
       {
         m_observers.push_back(observer);
       }
       void detach(Observer* observer)
       {
           for(auto it=m_observers.begin();it!=m_observers.end();++it)
           {
                 if(*it==observer)
                 {
                     m_observers.erase(it)
                     break;
                  }
            }
        }
       void setName(string newName)
       {
            name=newName;
            notifyObservers();
        }
        void notifyObservers()
        {
               for(Observer* obj:m_observers)
               {
                  obj→update(this);
               }
         }
};
```

```
void Gen7Device::update(CabinStation* m_cabin)
{
    cout<<"Gen7Device="<<m_cabin→getId()<<","<<m_cabin→getName()<<endl;
}
void Gen8Device::update( CabinStation* m_cabin)
{
    cout<<"Gen8Device="<<m_cabin→getId()<<","<<m_cabin→getName()<<endl;
}


int main()
{
    CabinStation m_cabin(123);

    Gen7Device m_gen7("Customer 1");
    Gen8Device m_gen8("Customer 2");

    m_cabin.attach(&m_gen7);
    m_cabin.attach(&m_gen8);

    m_cabin.setName("send Information");

    m_cabin.detach(&m_gen7);

    m_cabin.setName("Send Gen8 Infomation");

    return 0;
}
```

Adavantages: **Scalability:** you can easily add or remove observers without modifying the subject. This makes it a flexible solution for systems with dynamic requirements.

- Reusability: Observers can be reused in different contexts, provided they adhere to the observer interface or class