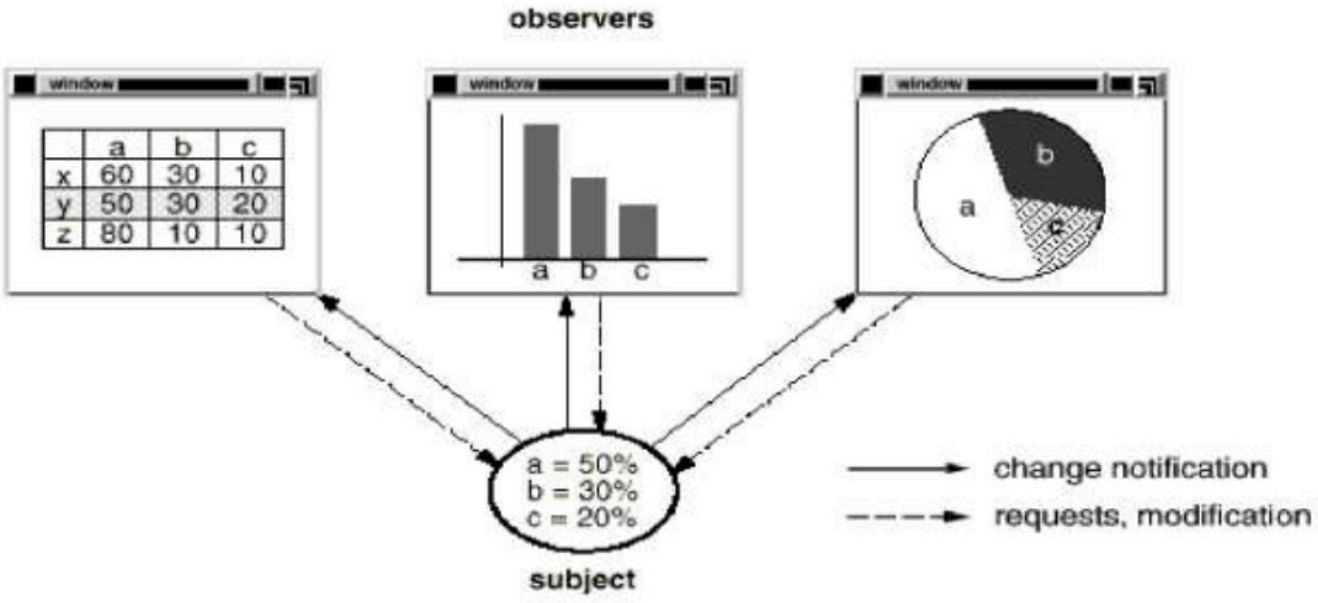
Sablonul Observer (Observer Design pattern)

Defineste o relatie de dependenta one-to-many intre obiecte astfel incat in momentul in care obiectul schimba starea toate obiectele dependente sunt notificate automat.



Observer - cod C++

```
class Observer {
public:
	virtual void update()=0;
};

class InterestedObj: public Observer {
public:
	void update() {
		std::cout << "Notified" << std::endl;
	}
};

void notify(Observer* obs) {
	obs->update();
}
class Observable {
public:
	void addObserver(Observer *obs) {
		observers.push_back(obs);
	}
	void doStuff() {
		//some stuff
		notifyObservers();
	}
private:
	std::vector<Observer*> observers;
	void notifyObservers() {
		for_each(observers.begin(), observers.end(), notify);
	}
};

int main() {
	Observable someObject;
	Observer* obs1 = new InterestedObj();
	Observer* obs2 = new InterestedObj();
	someObject.addObserver(obs1);
	someObject.addObserver(obs2);
	someObject.doStuff();
	return 0;
}
```

Observer - cod C++

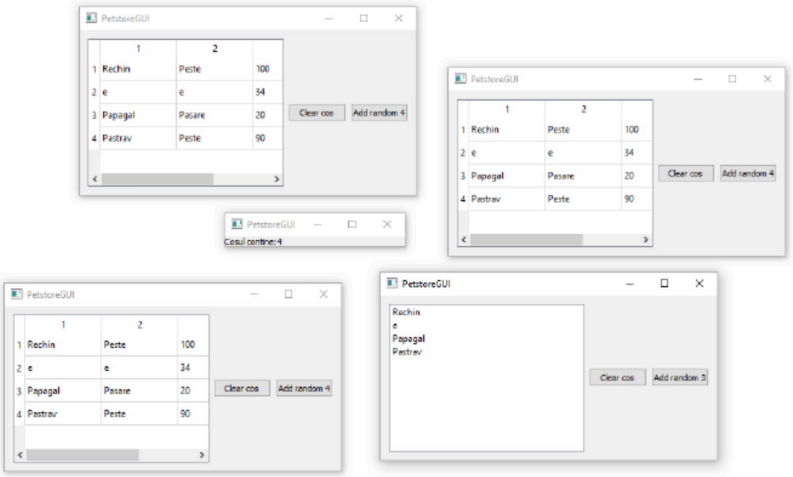
```
/* Update method needs to be implemented by observers
  Alternative names: Listener */
class Observer {
public:
      /* Invoked when Observable change
      Alternative names:properyChanged      */
      virtual void update() = 0;
};
/* Derive from this class if you want to provide notifications
  Alternative names: Subject, ChangeNotifier */
class Observable {
private:
      /*Non owning pointers to observer objects*/
      std::vector<Observer*> observers;
public:
      /* Observers use this method to register for notification
        Alternative names: attach, register, subscribe, addListener */
      void addObserver(Observer *obs) {
            observers.push_back(obs);
      }
      /* Observers use this to cancel the registration
      !!!Before an observer is destroyed need to cancel the registration
     Alternative names: detach, unregister, removeListener */
      void removeObserver(Observer *obs) {
        observers.erase(std::remove(begin(observers), end(observers),obs),
                        observers.end());
      }
protected:
      /* Invoked by the observable object
         in order to notify interested observer */
      void notify() {
            for (auto obs : observers) {
                  obs->update();
            }
      }
};
```

```
class InterestedObj : public Observer {
public:
  void update() override{
    std::cout << "Notified\n";
  }
};
```

```
class ConcreteSubject: public Observable {
public:
      void doStuff() {
            //...
            notify();//inherited
      }
};
```

Exemplu : cos de cumparaturi

Problema : sa avem multiple ferestre (de diferite tipuri) care prezinta acelasi cos cu animale. Fiecare fereastra trebuie actualizata in momentul in care se schimba continutul cosului.



Clasa Cos extinde **Observable** - contine lista de Pet din cos, notifica orice modificare in lista.
Clasele CosTableGUI, CosListGUI, CosLabelGUI - extind Observer si se inscriu pentru notificare (addObserver)

Folosind sablonul Observer obtinem :

- Clasa Cos nu este dependenta de clasele GUI
- Clasele GUI nu depind una de cealalta
- Se pot adauga cu usurinta noi clase GUI

Sablonul Observer - variante

Pull vs Push

`class Observer {`
`public:`
`  virtual void update() = 0;`
 	
`};`
	`class Observer {`
	`public:`
	`virtual void update(int changedState)=0;`
	`};`

Pull: cel care este notificat trebuie sa obțină datele. In general obiectul interesat are referința la subiect (așa obține informațiile dorite)	Push: cel care este notificat primește informații despre ce s-a schimbat. Obiectul interesat nu are nevoie de referință la subiect. Exista si varianta in care se primeste subiectul ca parametru la update

Notificari diferite in functie de ce s-a schimbat in obiectul subiect

`class Observer {`
`public:`
` virtual void itemAdded()=0;`
` virtual void itemRemoved()=0;`
` virtual void itemUpdated()=0;`
`};`
	`class Observer {`
	`public:`
	` virtual void itemAdded(Item item)=0;`
	` virtual void itemRemoved(Item item)=0;`
	` virtual void itemUpdated(Item item)=0;`
	`};`

Obiectul interesat trebuie sa implementeze toate metodele pure. Poate reacționa diferit in funcție de ce schimbări au apărut	

Observer Varianta Qt - semnale, sloturi/lambda, QObject::connect, emit

addObserver -> `QObject::connect` **removeObserver** -> `QObject::disconnect`
Elimina nevoia de a implementa o anume interfața (extinde Observer, Observable) Funcționează atât cu varianta Pull cat si Push (daca un semnal trimite si valori acestea se vor primi ca si parametru la slot sau lambda care s-a connectat la semnal Sursa semnalului si slotul (codul care se executa când apare semnalul) sunt total independente Se bazează pe generare de cod C++ (moc compiler)