Design Patterns — A quick guide to Observer pattern.

medium.com/datadriveninvestor/design-patterns-a-quick-guide-to-observer-pattern-d0622145d6c2

February 8, 2019

Observer pattern is a very commonly used pattern. In fact, it is so common that is being standardized in many programming languages/libraries. In Java, it exists in `java.util.Observer` (deprecated in Java 9). In Python it as close as a `pip install pattern-observer`. In C++, we can sometimes use boost library, more precisely `#include <boost/signals2.hpp>`. However, it is widely used in industry as a custom-made solution. To be able to use it correctly and understand its complexity, we need to dive in and explore it.

> **Observer** pattern is classified among the behavioral design patterns. Behavioral design patterns are most specifically concerned with communication between classes/objects. [by Design Patterns explained simply]

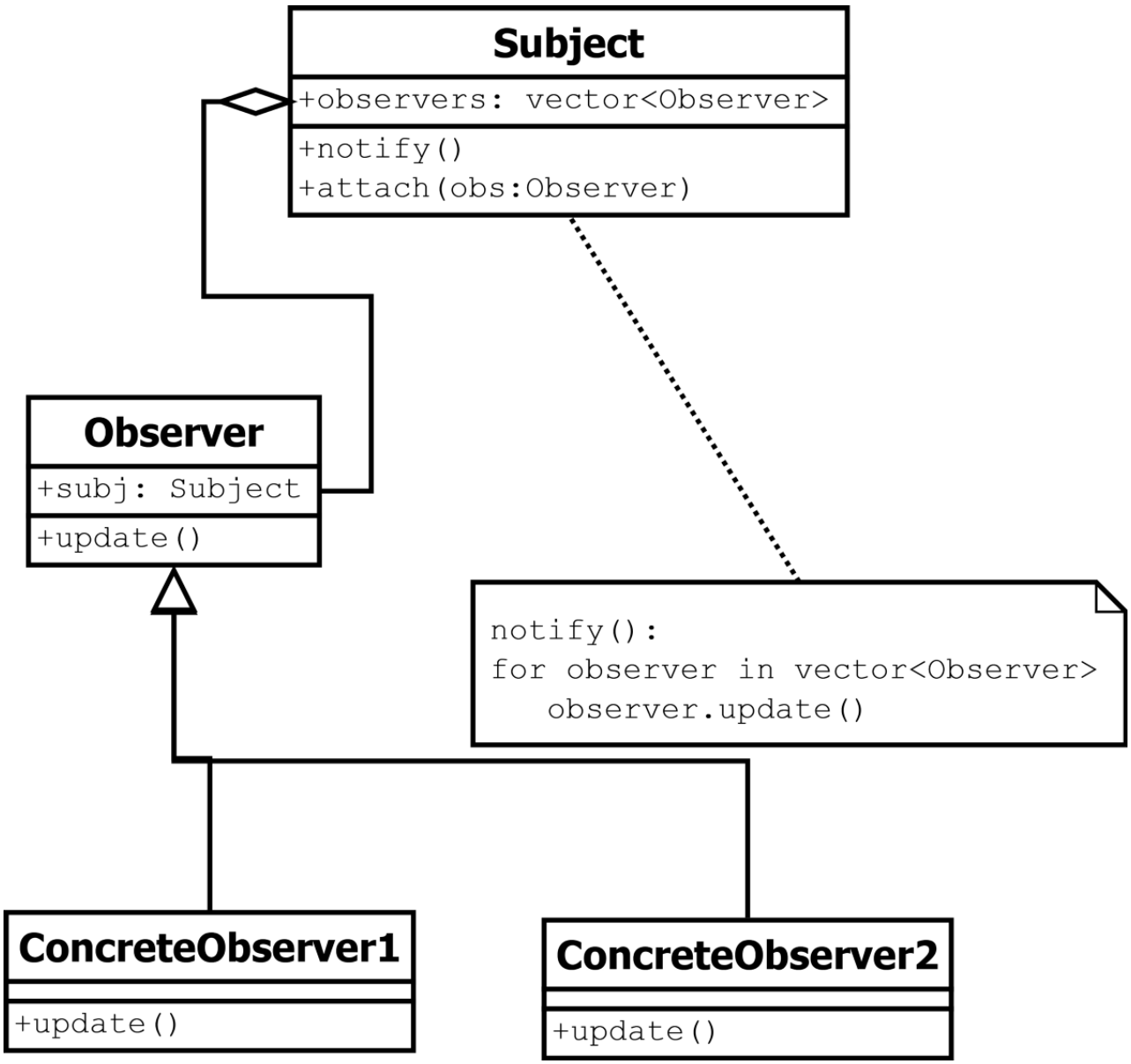
What is an **observer pattern**? Apart from a walking monitor that broadcasts analog television (as in the picture). The pattern's aim is to define a one-to-many relationship such that when one object changes **state**, the others are **notified** and **updated** automatically. More precisely, it wishes to be informed about events happening in the system. Lets put the pieces of the puzzle together in three steps.

Step 1 — Keywords

Defining **keywords** is the secret recipe in this series of quick-guides. This method helped me truly understand the design patterns, hardcode them in my mind and comprehend the differences among other design patterns.

1. **Subject:** It is considered as the keeper of information, of data or of business logic.
2. **Register/Attach**: Observers register themselves to the subject because they want to be notified when there is a change.
3. **Event:** Events act as a trigger in the subject such that all the observers are notified.
4. **Notify:** Depending on the implementation, the subject may "push" information to the observers, or, the observers may "pull" if they need information from the subject.
5. **Update:** Observers update their state independently from other observers however their state might change depending on the triggered event.

Step 2 — Diagram



Lets split this design into different classes to simplify this a little bit.

- The **ConcreteObservers** are classes that contain information specific to the current instance. The update function is called by the subject's `notify()` operation. The observers update independently based on their current state.
- The **Observer** is the parent class of the concrete observers. It contains a subject instance. When an observer is initialized, it registers/attaches itself to the subject.
- The **Subject** class has a list or a collection of observers. When an event is triggered it calls the `notify()` operation which loops through all the observers by calling their update function.

Step 3 — Code by Example

I would suggest to copy the code class by class from my **git repository "Andreas Poyias"** or the snippets below (in the order provided) and paste it in any of the available online C++ editors like c++shell, jdoodle , onlineGDB and run it to **observe the output**. Then read the comments or description below. Take your time, reading it thoroughly (that means one minute, not less and not more).

Example: Consider a football game. Many supporters are watching the game. We split the supporters into two categories by age, young and old. When their team scores a goal the supporters react differently according to their age and their excitement level.
Now, let's talk with terms used for the observer pattern:

- The game is the **subject** and the supporters are the **observers**.
- All observers are **attached/registered** to the subject and they are **notified** when their football team scores (the **trigger-event** is if their team scores).
- The observers **update** their behavior depending on the notification received.

Subject

For this class, we need access to a list of observers. When the observers are about to register, they call the `attach(this)` function to add themselves to the available list (`this` is an observer's instance). When an event is triggered we `notify()` all of the observers to independently update their state. In this example, the trigger is if the observer's football team scored.

```
#include <iostream>
#include <vector>
using namespace std;class Subject {
   vector < class Observer * > observers;
   bool scored;                  // trigger, event

public:
   // register observers
void attach(Observer *obs) {
      observers.push_back(obs);
   }

       // This is the EVENT
  // set the if scored and notify ALL observers
  void setScored(bool Score) {
    scored = Score;
    notify();
  }bool getScored() {
    return scored;
  }   // notify implementaion is further down
  // so that the script compiles and runs
  void notify();
};
```

Observer

This class is dependent on the subject it is registered with. When concrete observers get initialized they attach themselves to the `Subject` . In this example, the **state** of each observer is his `excitementLevel` about the game.

```
class Observer
{
   Subject *subj;
   int excitementLevel;              // state  public:
   Observer(Subject *mod, int excLevel)
   {
      subj = mod;
      excitementLevel = excLevel;
      // Observers register/attach themselves with the Subject
      subj->attach(this);
   }    virtual void update() = 0;  protected:
   Subject *getSubject() {
     return subj;
   }    void setExcitementLevel(int excLevel) {
     excitementLevel = excLevel;
   }    int getExcitementLevel() {
     return excitementLevel;
   }
};
```

This is the `Subject::notify()` declaration and as we mentioned before its job is to notify all observers to update their state.

```
void Subject::notify() {
  for (int i = 0; i < observers.size(); i++)
    observers[i]->update();
}
```

Concrete Observers

The concrete observers inherit from the Observer class and they all must have the update function. In this example, the concrete observers are distinguished between young and old supporters. If their excitement level gets too high the older supporters have the risk of heart attacks and the younger ones have the risk of drink and drive. Their state updates independently as we will prove in the main function further below.

```
class Old_ConcreteObserver: public Observer
{
  public:
   // Calls parent constructor to register with subject
   Old_ConcreteObserver(Subject *mod, int div)
     :Observer(mod, div){}     // For older people, if the excitement level
   // is over 150 they run risk of heart attack
   void update()
   {
     bool scored = getSubject()->getScored();
     setExcitementLevel(getExcitementLevel() + 1);        if (scored && getExcitementLevel()
> 150)
     {
      cout << "Old Observer's team scored!!"
          << " His excitement level is "
          << getExcitementLevel()
          << " watch out of heart attacks!" << endl;
     }else{
      cout << "Team didn't score. Yeeeih nothing to worry about"
          << endl;
     }
   } // end update()
};class Young_ConcreteObserver: public Observer
{
  public:
   // Calls parent constructor to register with subject
   Young_ConcreteObserver(Subject *mod, int div)
    : Observer(mod, div){}     // For older people, if the excitement level
   // is over 100 they run risk of heart attack
   void update()
   {
     bool scored = getSubject()->getScored();
     setExcitementLevel(getExcitementLevel() + 1);        if (scored && getExcitementLevel()
> 100)
     {
      cout << "Young Observer's team scored!!"
          << " His excitement level is "
          << getExcitementLevel()
          << " dont't drink and drive!!" << endl;
     }else{
      cout << "Team didn't score. Yeeh nothing to worry about"
          << endl;
    }
   } // end update()
};
```

Main function

The concrete observers register themselves to the `Subject` instance. Their state is the excitement level which is the second parameter. When the event is triggered " `subj.setScored(true)` ", then `Subject::notify()` is called to update the registered observers. In the scenario below, we have three observers, the `youngObs1` is

overexcited and runs the risk of drink and drive, the `oldObs1` is also overexcited a runs a different risk (of heart attack). Finally, `youngObs2` who is also young as the first one has nothing to worry about as he is not overexcited.

> It is important to notice that the three observers updated independently based on their state (excitement level) and their type (young or old).

```
int main() {
  Subject subj;
  Young_ConcreteObserver youngObs1(&subj, 100);
  Old_ConcreteObserver oldObs1(&subj, 150);
  Young_ConcreteObserver youngObs2(&subj, 52);
  subj.setScored(true);
}// Output
// Young Observer's team scored!! His excitement level is 101
// dont't drink and drive!! // Old Observer's team scored!! His excitement level is 151 watch
// out of heart attacks! Team didn't score. // Yeeh nothing to worry about
```

There are a few **benefits** for the use of Observer pattern and a few points to be noted when this pattern is to be approached[Learning Python Design Patterns].

- The Observer pattern provides a design where the Subject and Observer are **loosely coupled**. The subject does not need to know about the ConcreteObserver class. Any new Observer can be added at any point in time. There is no need to modify the Subject when a new Observer is added. Observers and subjects are not tied up and are independent of each other, therefore, changes in the Subject or Observer will not affect each other.
- There is no option for composition, as the Observer interface can be instantiated.
- If the Observer is misused, it can easily add complexity and lead to performance issues.
- Notifications can be unreliable and may result in race conditions or inconsistency.

The next blog will be a quick guide to the **Bridge** design pattern. It is a structural design pattern which is used quite a lot in the industry. Don’t forget to like/clap my blog-post and follow my account. This is to give me the satisfaction that I helped some fellow developers and push me to keep on writing. If there is a specific design pattern that you would like to learn about then let me know so I can provide it for you in the future.

Other quick-guides on design patterns: