Observer (obserwator)

Cel:

Zdefiniowanie zależności między obiektami, aby zmiana stanu jednego z nich powodowała zmianę stanu innego.

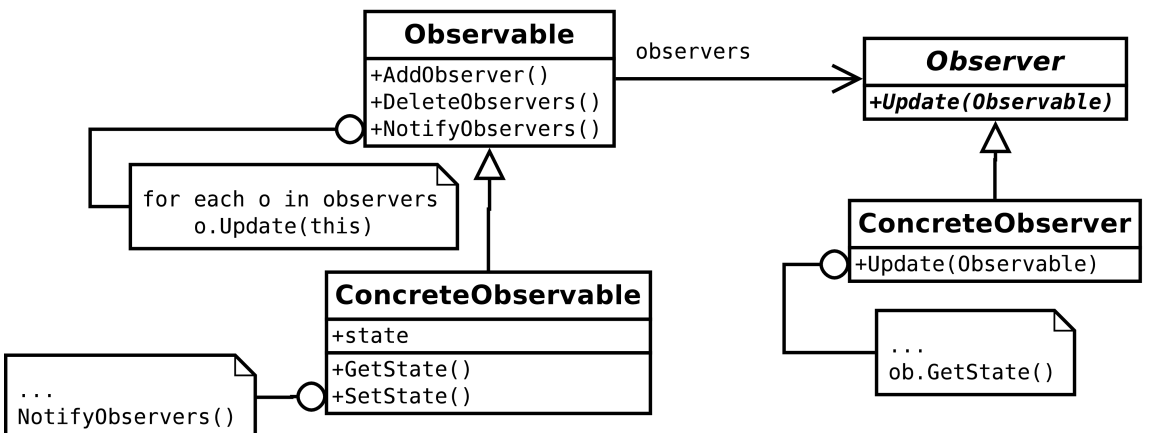
Przykład:

```
class Data {
     private double[] data;
     public void set(int idx, double value) { ... }
     public double get(int idx) { ... }
}
class Chart {
     public Chart(Data d) { ... }
     public void show() { ... }
}
// Main:
     Data data = new Data(...);
     new Chart(data).show();
     data.set(3, 2.5); // jak zaktualizować wykres?
```

Zastosowanie:

- Kiedy abstrakcja ma dwa zależne od siebie aspekty (a zawarcie ich w odrębnych klasach pozwoli rozbudowywać je je niezależnie).
- Gdy zmiana jednego obiektu powinna wyzwalać zmianę innych, lecz nie wiadomo ilu i jakich.
- Gdy obiekt powinien powiadamiać innego, niezależnie od tego kim on jest (czyli bez istnienia ścisłego związku między nimi).

Struktura:



Składniki:

- ***Observable*** (obiekt obserwowany) – wie o swych obserwatorach (gdyż może być obserwowany przez kilku), udostępnia interfejs dołączania czy usuwania obserwatorów
- ***Observer*** – deklaruje operację Update, powiadamiającą o zmianach w obiekcie
- ***ConcreteObservable*** – przechowuje stan interesujący obserwatora, wysyła mu powiadomienie, gdy się on zmieni
- ***ConcreteObserver*** – reaguje na zmianę stanu obiektu obserwowanego

Zależności:

Obiekt obserwowany powiadamia obserwatora, gdy zmieni się jego stan. Obserwator może pozyskać z niego informację potrzebną do zaktualizowania własnego stanu.

```
class Data extends Observable {
     ...
     public void set(int idx, double value) {
          ...
          notifyObservers();
     }
}
class Chart implements Observer {
     ...
     public Chart(Data d) {
          ...
          d.addObserver(this); // rejestracja obserwatora
     }
     public void update(Observable ob, Object arg) {
          ... // zmieniono dane – odświeżamy rysunek
     }
}
// Main:
     Data data = new Data(...);
     new Chart(data).show();
     data.set(3, 2.5); // Chart zostanie zaktualizowany
```

Konsekwencje:

Pozwala zmieniać obserwatorów i obiekty obserwowane niezależnie od siebie, a także:

1. Luźny związek obserwatora z obiektem obserwowanym – który wie tylko, że ma jakichś obserwatorów – dzięki temu mogą oni należeć do różnych warstw abstrakcji systemu: np. obiekt niskiego poziomu może powiadamiać obiekty wysokiego poziomu.
2. Obiekt obserwowany nie musi wiedzieć ilu ma obserwatorów, ani kto jest odbiorcą jego powiadomienia, dzięki temu można ich swobodnie dodawać i usuwać.
3. Obserwatorzy nie wiedzą o sobie nawzajem, zatem powodując zmiany w kolejnych obiektach mogą wyzwolić kolejne powiadomienia obserwatorów. Można temu zaradzić rozbudowując protokół powiadamiania – dodając informację o tym co się zmieniło i w jaki sposób.

Implementacja:

1. Przeważnie obiekt obserwowany przechowuje referencje na swoich obserwatorów:

```
class Observable {
     private List<Observer> obs;
     public void addObserver(Observer o) {
          if(!obs.contains(o))
               obs.add(o);
     }
     public void deleteObserver(Observer o) {
          obs.remove(o);
     }
     public void notifyObservers(Object arg) {
          for(Observer o : obs)
               o.update(this, arg);
     }
}
```

2. Inne rozwiązanie, to użycie tablicy asocjacyjnej (obiekt – obserwator).
3. Z powiadomieniem wysyłana jest informacja o źródle powiadomienia oraz parametry dodatkowe, które mogą być wykorzystane przez obserwatora.

4. Klasa Observable z Javy udostępnia metodę śledzenia zmian w obiekcie. Jeśli obiekt nie został zmodyfikowany, powiadomienie nie jest wysyłane do obserwatorów.

```
class Observable {
     private boolean changed = false;
     public void setChanged() {
          changed = true;
     }
     public void clearChanged() {
          changed = false;
     }
     public boolean hasChanged() {
          return changed;
     }
     public void notifyObservers(Object arg) {
          if(!changed)
               return;
          clearChanged();
          for(Observer o : obs)
               o.update(this, arg);
     }
}
```

```
class Data extends Observable {
     ...
     public void set(int idx, double value) {
          ...
          setChanged();
          notifyObservers();
     }
}
```

5. Kto powinien wywoływać metodę notifyObservers?
   - operacje zmieniające stan, po zmianie stanu obiektu – dzięki temu klient nie musi o tym pamiętać; może okazać się mało efektywne, gdy dokonujemy serii zmian
   - klient, po całej serii zmian, unikając niepotrzebnych powiadomień; jest to podatne na błędy (klienta)
6. Usunięci obserwatorzy (np. zamknięte okno wykresu) powinni informować o tym obiekt obserwowany, aby uniknąć niepotrzebnych powiadomień

```
class Chart implements Observer {
     ...
     public void close() {
          data.deleteObserver(this);
     }
}
```

7. Obiekt obserwowany wysyła (metoda ***update()***) dodatkowe informacje:
   - **model push** – są to szczegółowe informacje dotyczące dokonanej zmiany (zakłada, że obiekt obserwowany wie, czego obserwator potrzebuje)
   - **model pull** – jedynie powiadomienie, obserwator sam musi pytać o szczegóły (zakłada, że obiekt obserwowany nie wie zbyt wiele o obserwatorze)
8. Można podnieść efektywność, rejestrując obserwatorów oddzielnie dla różnych zdarzeń – czyli obserwujących tylko pewien aspekt danego obiektu.

Powiązania:

- Jeśli zależności między obiektem obserwowanym a obserwatorami są złożone, można je zamknąć w obiekcie ChangeManagera – będzie mapował przedmiot obserwacji do obserwatora, a także definiował pewną (bardziej wydajną?) strategię powiadomień. Jest to instancja Mediatora i zapewne również Singleton.