Observer

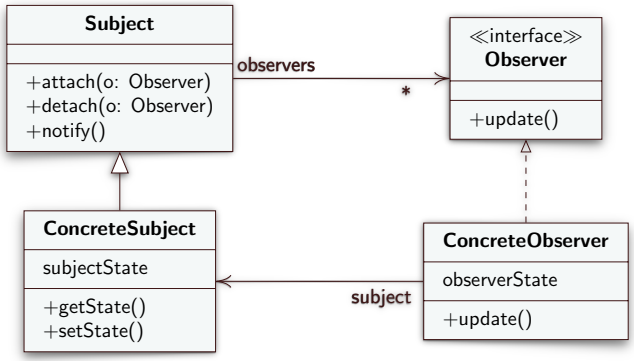
- **Intento**: Definire una dipendenza uno a molti fra oggetti, così che quando un oggetto cambia stato tutti i suoi oggetti dipendenti sono notificati e **aggiornati automaticamente**
- *Conosciuto anche come Publish-Subscribe*
- **Motivazioni**
  - Un sistema che è stato partizionato in un insieme di classi che cooperano deve mantenere la **consistenza** fra oggetti che hanno relazioni
  - Es. uno strumento di **presentazione** è separato dai **dati** dell'applicazione. Le classi che tengono i dati e quelle di presentazione sono separate e riusabili. Uno spreadsheet e un diagramma sono dipendenti dall'oggetto che contiene i dati e dovrebbero essere notificati dei cambiamenti dei dati

Observer

- Il design pattern Observer descrive come stabilire relazioni fra oggetti dipendenti
- Gli oggetti chiave sono **Subject** e **Observer**
- Un **Subject** può avere tanti **Observer** che dipendono da esso e gli Observer sono notificati quando lo stato del Subject cambia
- Applicabilità
  - Un'astrazione ha due aspetti (es. dati e presentazione), ciascuno dipendente dall'altro. Incapsulare questi aspetti in oggetti separati permette di riusarli indipendentemente
  - Un cambiamento su un oggetto richiede il cambiamento di altri, non si conosce quanti oggetti sarà necessario cambiare
  - Un oggetto deve notificare altri oggetti senza fare assunzioni su chi sono tali oggetti, quindi gli oggetti non devono essere strettamente accoppiati

Observer

- Soluzione
  - Diagramma delle classi

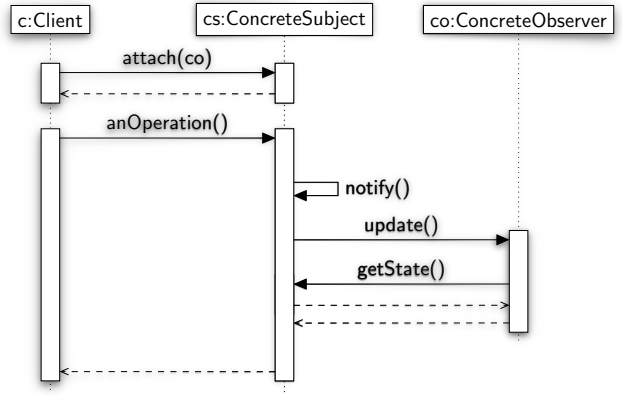


Observer

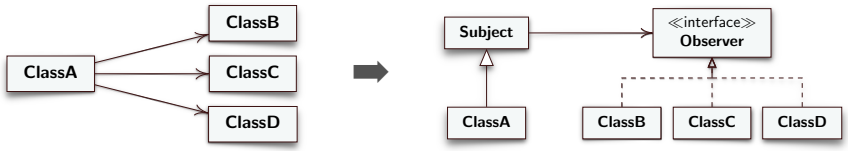
- Partecipanti
  - **Subject** conosce i suoi osservatori, un qualsiasi numero di oggetti Observer può osservare un Subject. Implementa le operazioni per aggiungere e togliere oggetti Observer e per notificarli
  - **Observer** definisce una interfaccia (operazione `update()`) comune a tutti gli oggetti che necessitano la notifica
  - **ConcreteSubject** tiene lo stato che interessa agli oggetti ConcreteObserver. Notifica i suoi osservatori quando il suo stato cambia. Eredita da Subject
  - **ConcreteObserver** tiene un riferimento all'oggetto ConcreteSubject, e tiene lo stato che deve rimanere consistente con quello del Subject. Implementa Observer per ricevere notifiche dei cambiamenti del Subject. Dopo la notifica può interrogare il subject per ottenere il nuovo dato

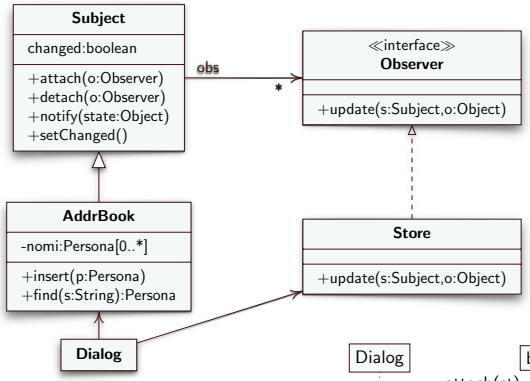
Observer

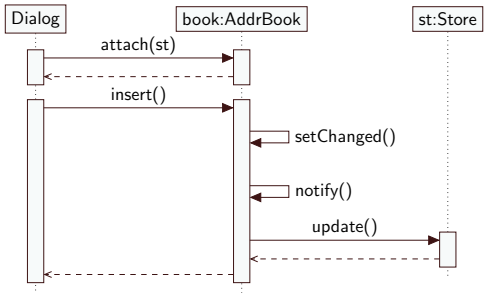
- Quando lo stato dell'oggetto ConcreteSubject cambia, l'oggetto ConcreteSubject chiama `notify()` che chiamerà `update()` sugli oggetti ConcreteObserver per aggiornarli



Prima e Dopo l'Uso di Observer








```java
public class Subject {
  private List<Observer> obs = new ArrayList<>();
  private boolean changed = false;
  public void notify(Object state) {
    if (!changed) return;
    for (Observer o : obs) o.update(this, state);
    changed = false;
  }
  public void setChanged() {
    changed = true;
  }
  public void attach(Observer o) {
    obs.add(o);
  }
  public void detach(Observer o) {
    obs.remove(o);
  }
}
```

```java
public class AddrBook extends Subject {
  private List<Persona> nomi = new ArrayList<>();

  public void insert(Persona p) {
    if (nomi.contains(p)) return;
    nomi.add(p);
    setChanged(); // la prossima notifica avverra'
    notify(nomi); // notifica i ConcreteObserver
  }
  public Persona find(String cognome) {
    for (Persona p : nomi)
       if (p.getCognome().equals(cognome)) return p;
    System.out.println("AddrBook.find: NOT found");
    return null;
  }
}
```



```java
public interface Observer {
  public void update(Subject s, Object o);
}
```

```java
public class Store implements Observer {
  @Override
  public void update(Subject s, Object o) {
    List<Persona> l = (List<Persona>) o;
    String nom;
    try (FileWriter f = new FileWriter("nomi.txt")) {
      for (Persona p : l) {
        nom = p.getNome() + "\t" + p.getCognome() +
              "\t" + p.getTelefono();
        f.write(nom + "\n");
      }
    } catch (IOException e) {  }
  }
}
```

```java
public class Dialog {
  private static final AddrBook book =
    new AddrBook();
  private static final Store st = new Store();
  private static final Persona p1 =
    new Persona("Oliver", "Stone", "012345", "NY");

  public static void main(String[] args) {
    book.attach(st);
    book.insert(p1);
  }
}
```

Observer

- Conseguenze
  - Il Subject conosce solo la classe Observer e non ha bisogno di conoscere le classi ConcreteObserver. ConcreteSubject e ConcreteObserver non sono accoppiati quindi più facili da riusare e modificare
  - La notifica inviata da un ConcreteSubject è mandata a tutti gli oggetti che si sono iscritti, il ConcreteSubject non sa quanti sono. Gli osservatori possono essere rimossi in qualunque momento. L'osservatore sceglie se gestire o ignorare la notifica
  - L'aggiornamento da parte del Subject può far avviare tante operazioni sugli Observer e altri oggetti per gli aggiornamenti. La notifica non dice agli Observer cosa è cambiato nel ConcreteSubject, è un evento che indica il completamento di un'operazione del ConcreteSubject

Avvertenze

- Con tanti Subject e pochi Observer, anziché tenere riferimenti a Observer nei Subject, si può usare una sola tabella associativa (in comune fra i Subject) per ridurre lo spazio impegnato
- Un Observer potrebbe osservare più oggetti, per sapere chi ha mandato la notifica, come parametro di `update()`, si manda il riferimento al Subject
- Subject chiama `notify()` dopo un cambiamento, oppure aspetta un certo numero di cambiamenti, in modo da evitare continue notifiche agli Observer
- Per mantenere la consistenza fra Observer e Subject, un cambiamento dell'Observer deve essere comunicato al Subject (con `setState()`)
- Quando si vuol eliminare un Subject, gli Observer dovrebbero essere avvisati per cancellare il loro riferimento al Subject
- Il Subject può passare ulteriori informazioni quando chiama `update()`, modello **push**; anziché aspettare che l'Observer legga lo stato, modello **pull**
- Gli Observer che si registrano possono specificare gli eventi di interesse, in modo che con l'`update()` si manda solo ciò che è cambiato

Observer In Java

- Il problema affrontato dal design pattern Observer è così comune che la sua soluzione è fornita nella libreria java.util, con i tipi `Observable` e `Observer`
- `Observable` svolge il ruolo di Subject quindi tiene traccia di tutti gli oggetti ConcreteObserver che vogliono essere informati di un cambiamento. `Observable` notifica il cambiamento di stato quando il metodo `notifyObservers()` viene chiamato
- La classe `Observable` ha una variabile (flag) che indica se lo stato è cambiato dalla precedente notifica, è impostato dal metodo `setChanged()`. La chiamata a `setChanged()` è da effettuare all'interno dei metodi della classe ConcreteSubject secondo la logica desiderata
- `Observer` è una interfaccia che ha solo il metodo `update()` e può avere un argomento che indica quale oggetto ha causato l'aggiornamento

Reactive Streams

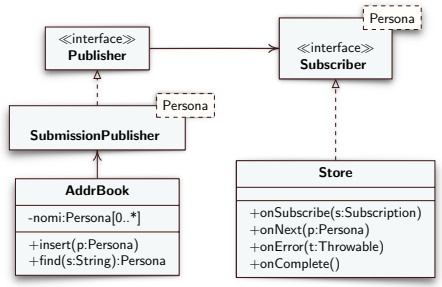
- Quando il ConcreteSubject chiama `notify()`, questo chiama `update()` che **esegue sul thread del chiamante**, costringendo il chiamante ad aspettare l'esecuzione di `update()` di ciascun ConcreteObserver
- Più recentemente, con i Reactive Streams, si è cercato di risolvere il problema del passaggio di un insieme di item da un **publisher** a un **subscriber** **senza bloccare il publisher** e **senza inondare il subscriber**
- Con la versione 9 di Java (settembre 2017), `Observable` e `Observer` sono disponibili per compatibilità con versioni precedenti, ma sconsigliati
- Java 9 fornisce nella libreria java.util.concurrent le interfacce `Publisher<T>`, `Subscriber<T>`, `Subscription`, e la classe `SubmissionPublisher`. Quest'ultima implementa un Publisher dedicando un thread per mandare ciascun messaggio ai Subscriber

Publisher Subscriber Java 9

- `Publisher` è l'astrazione per fornire item, definisce il metodo `subscribe()` che prende in input `Subscriber<T>`
- `SubmissionPublisher` implementa `Publisher` e invia item ai `Subscriber` **in maniera asincrona**. Quando sul SubmissionPublisher è chiamato il metodo `submit()`, esso esegue in un thread dedicato (asincrono) la chiamata al metodo `onNext()` dei Subscriber, e si blocca se il subscriber non può ricevere l'item
- `Subscriber` è usato per ricevere item, definisce i metodi `onComplete()`, `onError()`, `onNext()`, `onSubscribe()`
- `Subscription` definisce il collegamento fra `Publisher` e `Subscriber`; i metodi `cancel()` e `request(n)` permettono al `Subscriber` di fermare l'invio di messaggi e di richiedere l'invio dei prossimi n messaggi


```
public class AddrBook {
  private List<Persona> nomi = new ArrayList<>();
  private SubmissionPublisher<Persona> publ = new SubmissionPublisher<>();
  public void attach(Subscriber<Persona> s) {
    publ.subscribe(s);
  }
  public boolean insert(Persona p) {
    if (nomi.contains(p)) return false;
    nomi.add(p);
    publ.submit(p);
    return true;
  }
}

public class Store implements Subscriber<Persona> {
    private Subscription sub;
    @Override
    public void onSubscribe(Subscription s) {
        sub = s;
        sub.request(1);
    }
    @Override
    public void onNext(Persona p) {
        String nom = p.getNome() + "\t" + p.getCognome();
        System.out.println("Store onNext: "+nom);
        sub.request(1);
    }
    @Override
    public void onError(Throwable throwable) {
        System.out.println("In Store: errore");
    }
    @Override
    public void onComplete() {
        System.out.println("In Store: completato");
    }
}
```



Model View Controller (MVC)

- E' considerato un pattern architetturale per le applicazioni interattive, che individua tre componenti: **Model** per funzionalità principali e dati; **View** per mostrare i dati; **Controller** per prendere gli input dell'utente
- Motivazioni
  - Le interfacce utente possono cambiare, poiché funzionalità, dispositivi o piattaforme cambiano
  - Le stesse informazioni sono presentate in finestre differenti (per es. sotto forma di grafici diversi)
  - Le visualizzazioni devono subito adeguarsi alle manipolazioni sui dati
  - I cambiamenti all'interfaccia utente dovrebbero essere facili
  - Il supporto ai diversi modi di visualizzazione non dovrebbe avere a che fare con le funzionalità principali

Model View Controller (MVC)

- **Model** incapsula le funzionalità principali e i dati dell'applicazione. E' indipendente dalla rappresentazione degli output e dagli input. Registra View e Controller. Avvisa View e Controller registrati dei cambiamenti di dati
- **View** mostra i dati all'utente. Generalmente ci sono tante View, ogni View è associata a un Controller. View inizializza il proprio Controller, e mostra i dati che legge da Model
- **Controller** riceve gli input dell'utente (da mouse e tastiera) sotto forma di eventi. Traduce gli eventi in richieste di servizio per Model o avvisa View