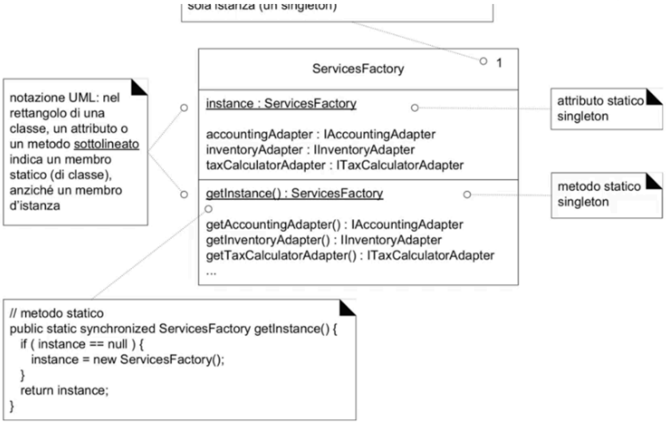
GOF

Saturday, 27 May 2023 17:18

- Sono dei design pattern
    - Ed essi sono una soluzione progettuale comune ad un problema progettazione ricorrente che è stata usata in passato per problemi
- Ne abbiamo 23 e sono strutturati da:
    - Creazionale
        - Singleton
            - ☐ Lavora su oggetti
            - ☐ Classe che può essere istanziata solamente 1 volta
            - ☐ Problema: è consentita esattamente 1 sola istanza di una classe. Gli altri oggetit hanno bisogno di accesso globale e singolo dell'oggetto
            - ☐ Soluzione: definisce metodo della classe statico che restituisce l'oggetto
              Se sottolineato vuol dire che è statico/singletone

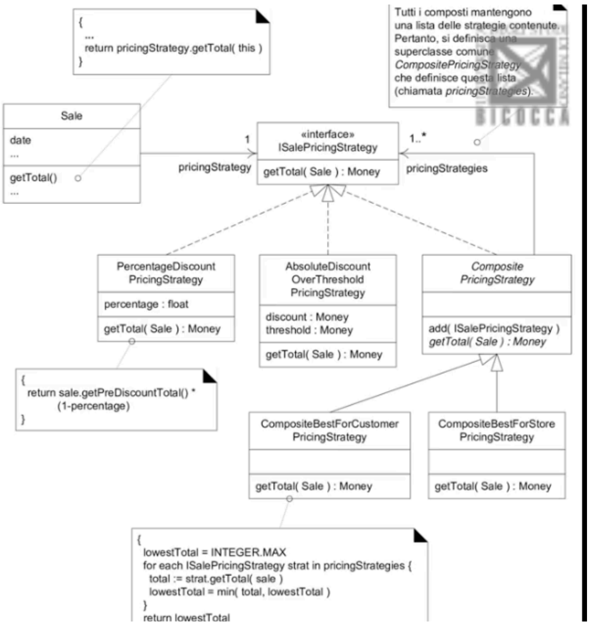


notazione UML: nel rettangolo di una classe, un attributo o un metodo sottolineato indica un membro statico (di classe), anziché un membro d'istanza

```
// metodo statico
public static synchronized ServicesFactory getInstance() {
  if ( instance == null ) {
    instance = new ServicesFactory();
  }
  return instance;
}
```

        - Factory
            - ☐ Lavora su classi
            - ☐ Crea delle istanze
            - ☐ Risolve i problemi: chi crea gli oggetti adattatori? Come stabiliamo quale classe adattatore creare?
            - ☐ Problema: chi deve essere responsabile della creazione di oggetti quando ci sono delle condizioni speciali, come una logica di creazione complessa quando si desidera separare la responsabilità di creazione per una coesione milgiore?
            - ☐ Soluzione: crea un oggetto factory che gestisce la creazione
            - ☐ Essa separa la responsabilità delle creazione, nasconde logica creazione e consente introduzione di gestione delle

        risorse
    - Sostiene protected variation
- Strutturale
  - Adapter
    - Lavora su classi
    - Problema: gestire interfaccia incompatibile
      - Quando l'oggetto server offre servizi di interesse al client, ma l'oggetto client vuole fruire il servizio in una diversa modalità dall'oggetto server
      - L'adapter trasforma l'output del server in input del client
    - Soluzione: converti interfaccia di un componente in un'altra interfaccia attraverso un oggetto adattatore intermedio
    - Es: il sistema ha un sistema interno api, ed ora vogliamo renderli disponibili esterni. Per farlo raggruppiamo insieme di api interne in api esterne con l'adapter
    - Sostiene protected variation, attraverso inderection che applica poliformismo
  - Composite
    - Lavora su oggetti
    - Una strategy che però potrebbe anche essere conflittuale
    - Problema: trattare un gruppo o struttura composto di oggetti di un oggetto non composto (aka atomico)
      Quindi creare una strategy che usa delle strategy
    - Soluzione: definisci classi/oggetti composti e atomici in modo tale che implementano la stessa interfaccia
      Praticamente abbiamo una classe che ha tutte le strategy internamente



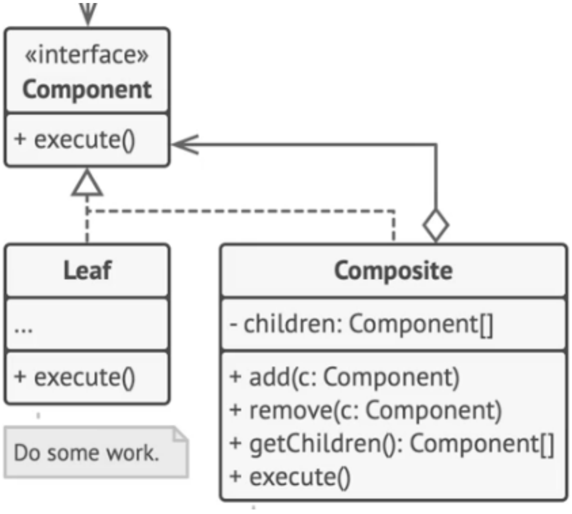
```
{
  ...
  return pricingStrategy.getTotal( this )
}
```

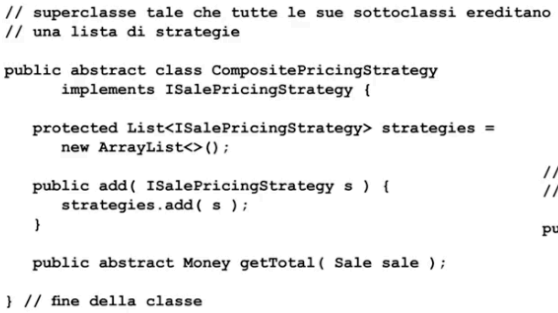
Tutti i composti mantengono una lista delle strategie contenute. Pertanto, si definisca una superclasse comune *CompositePricingStrategy* che definisce questa lista (chiamata *pricingStrategies*).

```
{
  return sale.getPreDiscountTotal() *
         (1-percentage)
}
```

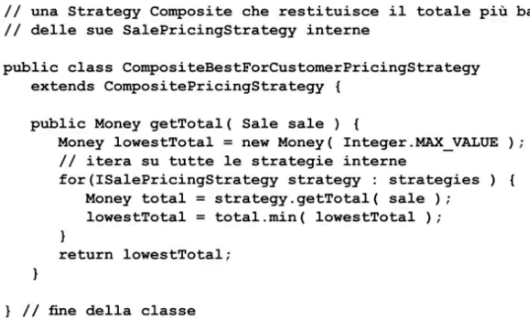
```
{
  lowestTotal = INTEGER.MAX
  for each ISalePricingStrategy strat in pricingStrategies {
    total := strat.getTotal( sale )
    lowestTotal = min( total, lowestTotal )
  }
  return lowestTotal
```

Possiamo notare la composite, e poi affianco tutte le strategie, la struttura dovrebbe essere la seguente:



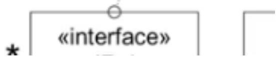


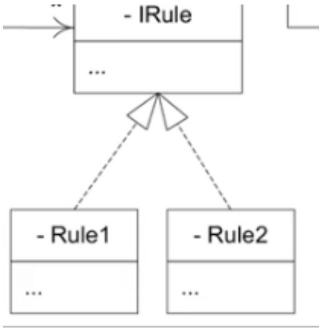
```
// superclasse tale che tutte le sue sottoclassi ereditano
// una lista di strategie

public abstract class CompositePricingStrategy
      implements ISalePricingStrategy {

   protected List<ISalePricingStrategy> strategies =
      new ArrayList<>();

   public add( ISalePricingStrategy s ) {
      strategies.add( s );
   }

   public abstract Money getTotal( Sale sale );

} // fine della classe
```



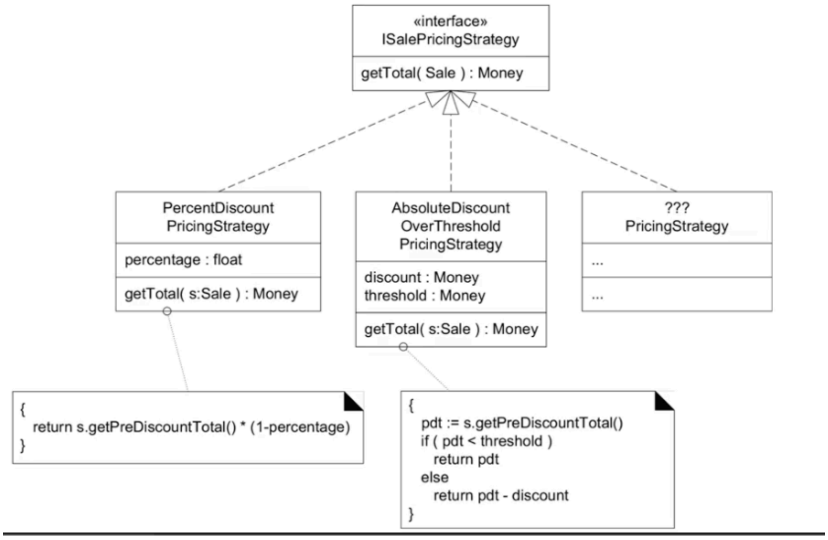
```
// una Strategy Composite che restituisce il totale più ba
// delle sue SalePricingStrategy interne

public class CompositeBestForCustomerPricingStrategy
   extends CompositePricingStrategy {

   public Money getTotal( Sale sale ) {
      Money lowestTotal = new Money( Integer.MAX_VALUE );
      // itera su tutte le strategie interne
      for(ISalePricingStrategy strategy : strategies ) {
         Money total = strategy.getTotal( sale );
         lowestTotal = total.min( lowestTotal );
      }
      return lowestTotal;
   }

} // fine della classe
```

  - ☐ Si basa sul polifrmosmo e protected variations
- ■ Facade
  - ☐ Lavora su oggetti
  - ☐ Problema: è richiesta una interfaccia comune e unificata per un insieme disparato di implementazioni/interfacce, come per definire un sottoinsieme. Può verificarsi un accoppiamento indesiderato a molti oggetti nel sottoinsieme, oppure implementazione del sottoinsieme può cambiare. Che cosa fare?
    Deve nascondere il sottosistema
  - ☐ Soluzione: definisci un punto di contatto singolo con il sottoinsieme ovvero un oggetto facade che copre il sottosistema. Questo oggetto facade presenta un'interfaccia singola e unificata ed è responsabile della collaborazione con i componenti del sottosistema.





- Comportamentale
    - Observer
        - Lavora su oggetti
        - Problema: diversi tipi di oggetti sono interessati al cambio di stato di un oggetto publisher. I subscriber vogliono reagire al cambiamento di evento e nel mentre il publisher vuole un basso accoppiamento
        - Soluzione: si crea una interfaccia subscriber (listener) che ascolta ciò che fà il publisher. I subscriber si iscrivono all'observer e lui li avviserà al cambiamento
    - Strategy
        - Lavora su oggetti
        - Creazione di un algoritmo con tanti algoritmi che potrebbero cambiare nel tempo
        - Problema: come progettare per gestire un insieme di algoritmi/politiche correlate? Come progettare per consentire modificare questi algo/poli?
        - Soluzione: definisci ciascun algo/poli/strate in una classe separata, con un'interfaccia comune
        - Si basa su poliformismo e protected variations



```
{
  return s.getPreDiscountTotal() * (1-percentage)
}
```

```
{
  pdt := s.getPreDiscountTotal()
  if ( pdt < threshold )
    return pdt
  else
    return pdt - discount
}
```

- I pattern GRASP potrebbero essere implementati in questi pattern