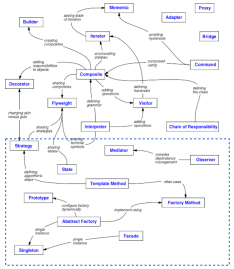
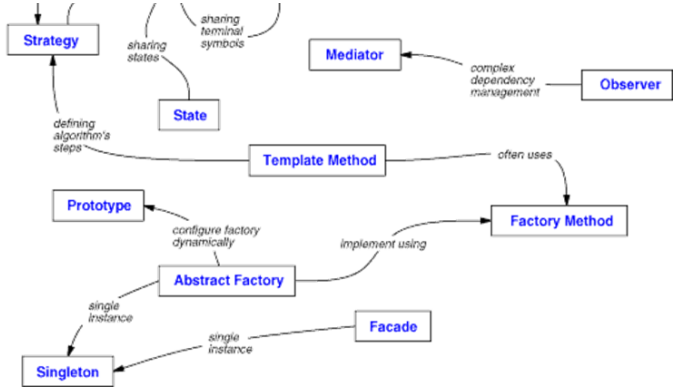
# Strategy
## Comportamentale

Prof. Andrea Calvagna



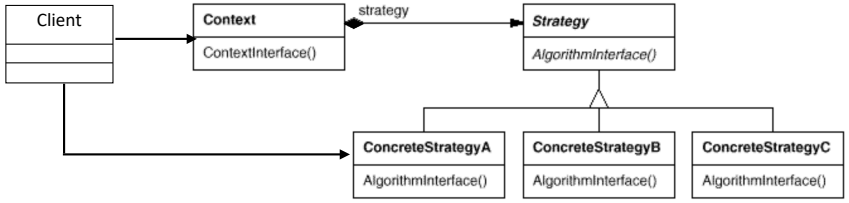
## Dove siamo…








## STRATEGY




Intento:

definire una famiglia di algoritmi correlati, incapsularli singolarmente e renderli intercambiabili

consentire la modifica di un algoritmo (strategia) indipendentemente dai contesti che lo usano

ContextInterface(){ …; strategy.AlgorithmInterface(); …}

Il contesto non sa quale strategia verrà applicata, viene scelta dal client

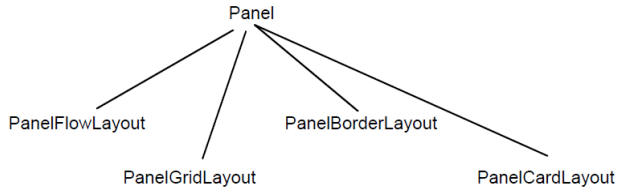


## Esempio: Java Layout Managers

```
import java.awt.*;
class FlowExample extends Frame {
        public FlowExample( int width, int height ) {
                setTitle( "Flow Example" );
                setSize( width, height );
                setLayout( new FlowLayout( FlowLayout.LEFT) );
                for ( int label = 1; label < 10; label++ )
                        add( new Button( String.valueOf( label ) ) );
                show();
        }

        public static void main( String args[] ) {
                new FlowExample( 175, 100 );
        }
}
```

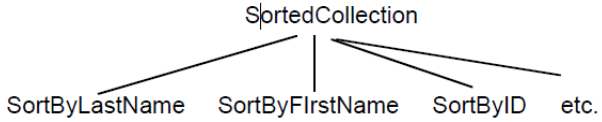
## Perché non derivare semplicemente sottoclassi?




- Ci sono circa 20 diversi layout
- Circa 40 sottoclassi di Component che possono usarle
  - Dovrei derivare circa 800 classi!



## Altro esempio: Java Comparable interface




- Si vuole supportare l'ordinamento delle collezioni
- Si può voler ordinare in qualsiasi modo arbitrariamente e con vari algoritmi
  - Avremmo una sottoclasse per ogni ordinamento di ogni tipo di Collection
  - Con i comparatori non derivo e posso combinare e variare a piacere i criteri con l'algoritmo di ordinamento



## In Java: Comparable interface

```
public class Player implements Comparable<Player> {
  private int ranking;
  private String name;
  private int age;
  // constructor, getters, setters
  @Override public int compareTo(Player otherPlayer) {
        return Integer.compare(getRanking(),otherPlayer.getRanking());
  }
  }
```



## Creiamo una collection di oggetti Player

```
public static void main(String[] args) {
    List<Player> footballTeam = new ArrayList<>();
    Player player1 = new Player(59, "John", 20);
    Player player2 = new Player(67, "Roger", 22);
    Player player3 = new Player(45, "Steven", 24);
    footballTeam.add(player1);
    footballTeam.add(player2);
    footballTeam.add(player3);
     Sort(…)… vedi esempio completo su vscode
    }
```

# Java Comparator vs Comparable

...VSCODE

- **L'interfaccia *Comparator* definisce un metodo *compare(arg1, arg2)*** con due argomenti che rappresentano gli oggetti confrontati e funziona in modo simile alla *Comparable.compareTo()*.

```
public class PlayerRankingComparator implements Comparator<Player>{
        @Override
        public int compare(Player firstPlayer, Player secondPlayer){
            return Integer.compare(firstPlayer.getRanking(),secondPlayer.getRanking());
        }
}
```

- Comparable deve implementarla un oggetto del tipo Player
- Comparator può implementarla un oggetto a parte, con funzione comparatore
  - Posso definire tutti i criteri di comparazione possibili, oltre quello *naturale,* per Player



# Lambda comparators (Java v.8)

- posso definire ***Comparators*** usando **Lambda expressions**

```
Comparator byRanking =
                (Player player1, Player player2) —>
                 Integer.compare( player1.getRanking(), player2.getRanking() ) ;
```

Espressioni Lambda: funzioni *anonime (senza nome)*
indico simbolicamente gli argomenti ed un'espressione che gli abbina un risultato
Ispirate al Lambda calcolo



# Comparator.Comparing() method

- Il metodo **statico *Comparator.comparing()*** prende come parametro un metodo che restituisce la proprietà da usare per il confronto, e ritorna una istanza corrispondente di ***Comparator*** :

```
Comparator<Player> byRank = Comparator.comparing(Player::getRanking);

Comparator<Player> byAge = Comparator.comparing(Player::getAge);
```



# Applicabilità

- Uso lo strategy pattern quando…
  - Mi servono versioni diverse di un algoritmo
  - Una classe definisce i suoi comportamenti attraverso metodi strutturati essenzialmente a switch multiplo
  - Ho molte classi correlate che differiscono solo nel modo in cui attuano il loro scopo
  - Un algoritmo deve manipolare dati, in modalità scelte dai suoi client, senza che questi possano accedervi
    - Come limitare l'accesso alla sola classe strategy? uso di *inner* class

# Inner class

- Classi definite come membri di altre classi

```
class MyOuter {
    class MyInner {
      …
    }
    …
}
```

- In quanto membri di un classe, possono essere private (unico caso)
- Ed hanno privilegio di accesso agli altri membri privati della classe outer
- Sono istanziabili solo nel contesto della classe outer

- possono essere anonime
- Vediamo un esempio di inner class passata anonima su VSCODE



# Implementazione

…esempi su VSCODE

- Definisco le interfacce delle strategie e del contesto
  - Definisco come interagiscono
    - Il contesto può passare i dati alla strategia, come parametro
    - Oppure la strategia ha accesso privilegiato al contesto
- posso implementare le strategie come parametri template dei contesti
  - Se sono in grado di indicarla a compile-time e non voglio cambiarla a run-time : es.: SortedList<ShellSort> studentRecords



# Conseguenze

- Avrò una famiglia di algoritmi a disposizione
- Situazione alternativa al sottoclassare il contesto di utilizzo
- I client devono essere a conoscenza delle strategie disponibili
- Elimino istruzioni condizionali multiple sostituendole con ***strategy.do()***
  - *Invece che:*

```
switch ( flag ) {
        case A: doA(); break;
        case B: doB(); break;
        case C: doC(); break;
}
```

- Contesto e strategia sono oggetti distinti che si messaggiano, invece che uno stesso oggetto: c'è un overhead per questa comunicazione.
- Notare come nei fatti uso la delega su un metodo di un oggetto di servizio, con uno scopo diverso ma similmente che in altri pattern già studiati



# Template vs Strategy

- Entrambi pattern comportamentali: mirano all'alterazione del comportamento
- Con il Template definisco ad alto livello un algoritmo, lasciando uno o più suoi passi aperti a varie possibili implementazioni (con l'override)
  - Creo varie sottoclassi (ADT), coesistenti: una "famiglia" di varianti dell'algoritmo base (**class scope**)
  - La variante è un ADT indicato nel contesto d'uso a compile-time: devo essere in fase di design per usarlo
  - la modifica del comportamento non compromette la funzionalità di alto livello
- Con lo Strategy creo una famiglia di oggetti intercambiabili che incapsulano algoritmi correlati (**object scope**)
  - Posso applicarli a tempo di esecuzione ad un contesto di esecuzione (un altro algoritmo), alterandone l'effetto ma non la funzionalità di alto livello
  - La variante da usare posso indicarla a run-time: è applicabile anche a contesti già compilati
- Entrambi devono essere trasparenti (non visibili) al contesto**: tutte le varianti fanno la stessa cosa (in modo diverso)**
- Entrambi supportano l'uso di **una sola variante per volta**: scelta la variante da applicare, quella è.