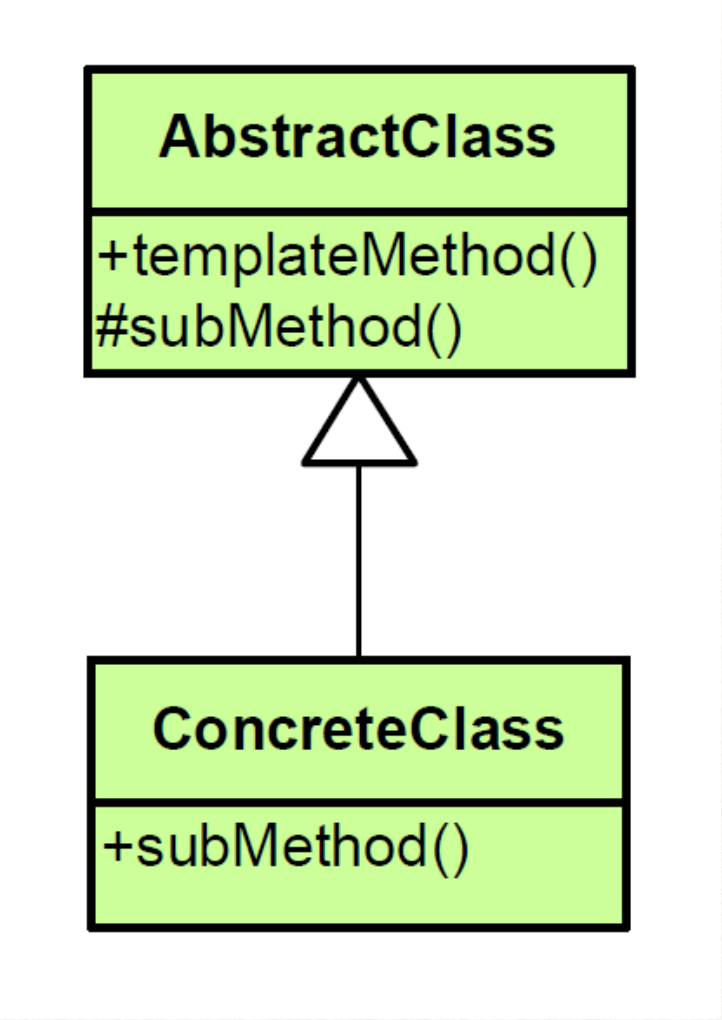
Template method

- Intento:
  - Permettere la codifica dell'algoritmo di una certa operazione, delegando l'implementazione di alcuni suoi passi alle sottoclassi
  - Attraverso le sottoclassi posso cambiare implementazione a certi passi di un algorimo senza però cambiarne la struttura complessiva

Template Method



- definisco la struttura (scheletro) del metodo nella classe origine ma lascio alcuni suoi passaggi astratti (o in versione default): per ri/definirli in varie versioni e/o combinazioni nelle sottoclassi

esempio senza pattern

```
class Account {
        String name;
        float balance;
        Account(String customerName, float InitialDeposit ) {
                name = customerName;
                balance= InitialDeposit;
        };
        public void Transaction(float amount){ balance += amount;}
}


class JuniorAccount  extends Account {
        public void Transaction(float amount) {// put code here}
}

class SavingsAccount extends Account {
        public void Transaction(float amount) {// put code here}
}
```

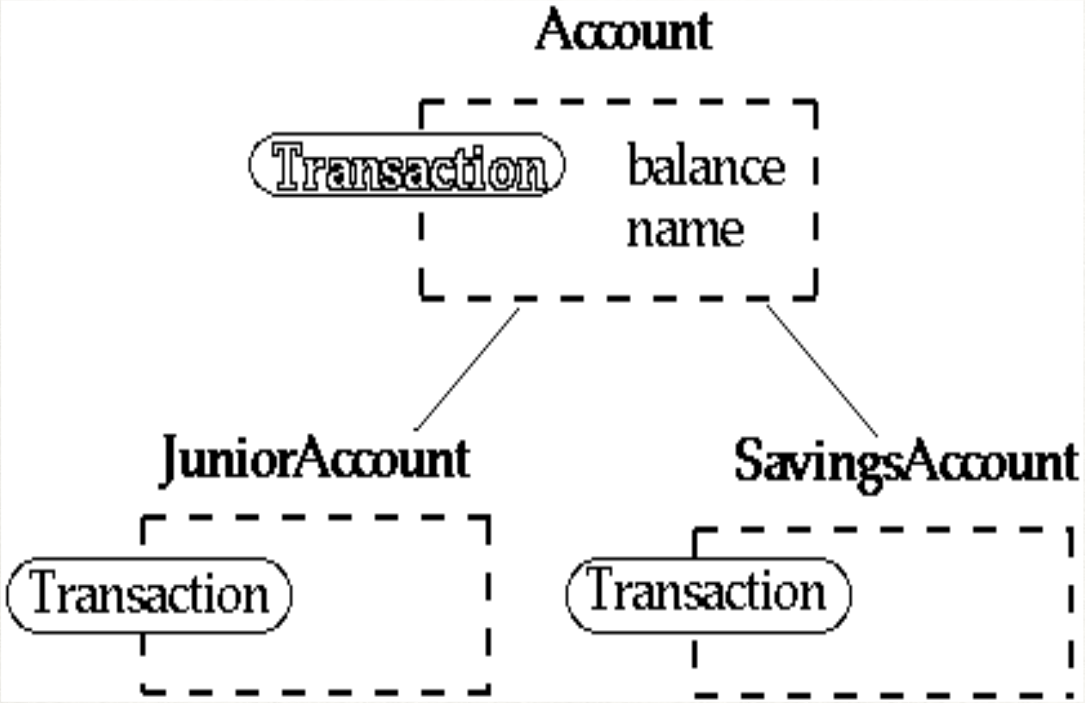
In una classe Client

```
Account createNewAccount(){
        // code to query customer and determine what type of
        // account to create
};


void main(…) {
        Account customer;
        customer = createNewAccount();
        customer->Transaction(amount);
}
```

l'implementazione di Transaction in Account è sostituita da quella della sottoclasse scelta: ho usato solo il classico polimorfismo

Notate l'uso di un factory method per incapsulare la scelta

Metodi «differiti»




```
abstract class Account {
        public abstract void Transaction();
}

class JuniorAccount  extends Account {
        public void Transaction() { //put code here}
}

class SavingsAccount  extends Account {
        public void Transaction() { //put code here}
}
```

Li dichiaro ma li lascio privi di implementazione: la possono dare le sottoclassi

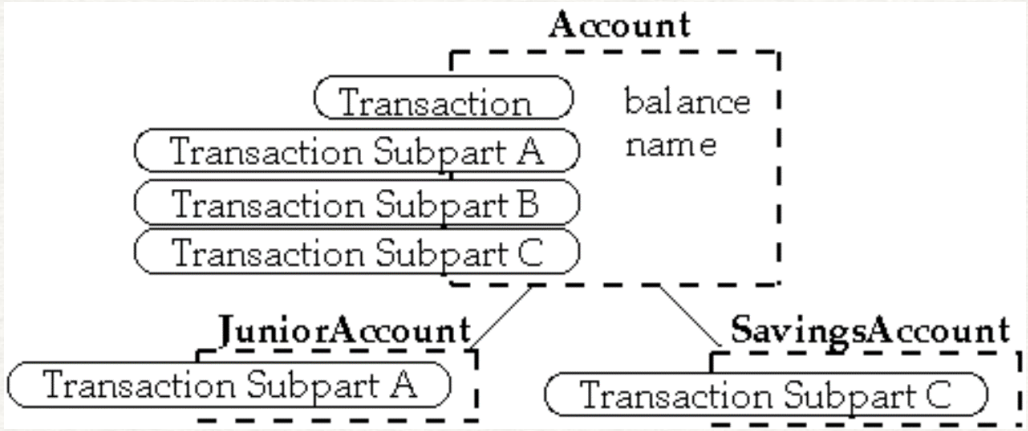
Oppure uso abstract

Oppure uso interfacce con *default methods (in java)*

Template method

```
class Account {
    public void TransactionSubpartA(){};
    public void TransactionSubpartB(){};
    public void TransactionSubpartC(){};

     Public void Transaction(float amount) {
            TransactionSubpartA();
            TransactionSubpartB();
            TransactionSubpartC();
            // EvenMoreCode;
}
```

```
class JuniorAccount extends Account {
            public void TransactionSubpartA() { //code};
}

class SavingsAccount extends Account {
            public void TransactionSubpartC(){//code};
}
```



…nel client….

```
Account customer;
customer = createNewAccount();
customer->Transaction(amount);
```

In base al tipo di Account selezionato sono in grado di cambiare in modo trasparente alcuni passi di un algoritmo mantenendone intatta la coerenza semantica complessiva

In Java (>8)

- **Default** methods nelle interfacce

```
public interface algoritmTemplate {

    void partA();
    void partB();
    void partC();

    default void algorithm(){
        partA();
        System.out.print("invariable part");
        partB();
        partC();
    }
}
```

```
class PartA implements prova {

    public void partA(){
        System.out.print("my partA\n");
    }

    public static void main (String[] args){
        new PartA().algorithm();
    }

 @Override
    public void partB() {
        …
    }
    @Override
    public void partC() {
        …
    }

}
```

Motivazione

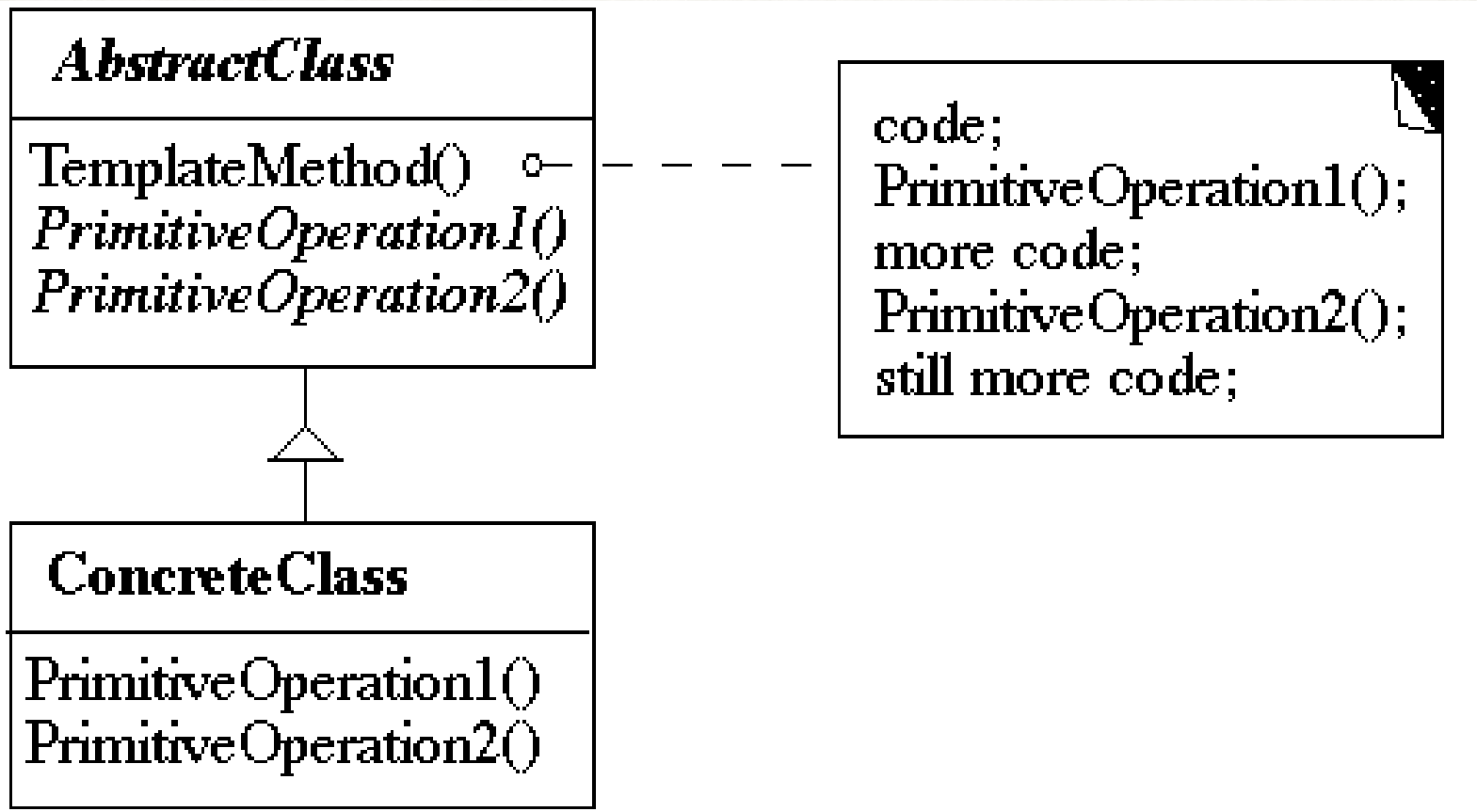
- Tipico degli Application framework con classi astratte con logica applicativa che opera su prodotti diversi, ad esempio su documenti con formati diversi:

```
abstract Class MyApplication {
…
        void OpenDocument (String  name ) {
                if (!CanNotOpenDocument (name)) return;
                Document doc = DoCreateDocument();

                if (doc) {
                        docs->AddDocument( doc);
                        AboutToOpenDocument( doc);
                        Doc->Open();
                        Doc->DoRead();
                }
        }
…

}
```

Applicabilità

- Applica questo pattern….
  - Per implementare le parti invarianti di una algoritmo una sola volta per tutte:
    - Le sottoclassi implementano le parti variabili
  - Per fattorizzare in una sola classe dei comportamenti comuni a varie sottoclassi ed eliminare la loro duplicazione
  - Per avere controllo di cosa sia modificabile in una classe attraverso la derivazione:
    - metodi template uniche parti estendibili dalle sottoclassi

Struttura



- **Abstract class**: definisce lo scheletro di un algoritmo e le operazioni primitive astratte che lo implementano
- **Concrete class:** implementa le operazioni primitive, dettagliando così in modo personalizzato l'algoritmo principale

Conseguenze

- È il più usato dei 23 pattern dei GoF
- Fondamentale nelle librerie di classi
- Inverte la struttura di controllo:
  - È la classe padre che si ritrova ad eseguire metodi della sottoclasse
  - Esempio in java: paint method in AWT/SWING

Java paint (AWT)

public void paint(Graphics g) definite in *java.awt.Component*

Il metodo Java paint è una operazione primitiva che io implemento ma chiamata da un metodo della classe padre, mai direttamente da me ***(callback function)***

```
class HelloApplication extends Frame {
    public void paint( Graphics display ){
        int startX = 30;
        int startY = 40;
        display.drawString( "Hello World", startX, startY );
    }
}
```

Java paint (swing)

- javax.swing.JComponent implementa il metodo paint dividendolo in tre metodi separati, invocati nell'ordine seguente:
  - **protected void paintComponent(Graphics g)**
  - protected void paintBorder(Graphics g)
  - protected void paintChildren(Graphics g)

- Metto il mio codice in paintComponent() e lo invoca paint():

```
public void paintComponent(Graphics g) {
    g.drawString("This is my custom Panel!",10,20);
    redSquare.paintSquare(g);
}
```

Template method

- Il metodo template invoca:
  - Operazioni concrete
  - Operazioni primitive (astratte)
  - Metodi Factory
  - Metodi «gancio» (hook operations)
    - Hanno implementazione di default estendibile dalle sottoclassi
    - È importante individuare cosa DEVE, cosa Può e cosa NON DEVE essere ridefinibile nell'algoritmo : uso

implementazione

- Le operazioni primitive le dichiaro ***protected*** in modo da consentirne l'uso solo alle sottoclassi
- I metodi template li dichiaro ***final*** per impedirne l'override (non virtuali in C++)
- Minimizzo le operazioni primitive
- Uso convenzioni di nome per indicare i metodi primitivi
    - Ad esempio nel framew. MacApp era usato il prefisso «Do»

Implementazione

- Primo passo: scrivo tutto il codice in un unico grande metodo che diverrà il template
- Lo divido in passi successivi usando ad es. i commenti
- Incapsulo ogni passo in un metodo separato
- Riscrivo il template invocando i metodi estratti
- Ripeto i passi dal primo su ognuno dei metodi estratti finchè:
  - Tutti i passi in ogni metodo non hanno la stessa granularità
  - Tutte le parti costanti sono fattorizzate nei loro propri metodi

Metodi costanti

- Non hanno decisioni: tornano sempre lo stesso valore
- Comuni nell'applicazione del template method alla *lazy inizialization*
- *Supponiamo di avere:*

```
public class Foo {
    Bar field;
    public final Bar getField() {
        if (field == null) field = new Bar( 10);
        return field;
    }
}
```

Se una sottoclasse volesse cambiare il valore di default del campo?

```
public final Bar getField({
        if (field == null) field = defaultField();
        return field;
}

protected Bar defaultField() {
        return new Bar(10);
}
```

Ora può farlo semplicemente con l'override di defaultField()

Lo stesso posso fare con I costruttori:

```
public Foo() {
        field := defaultField();
}
```

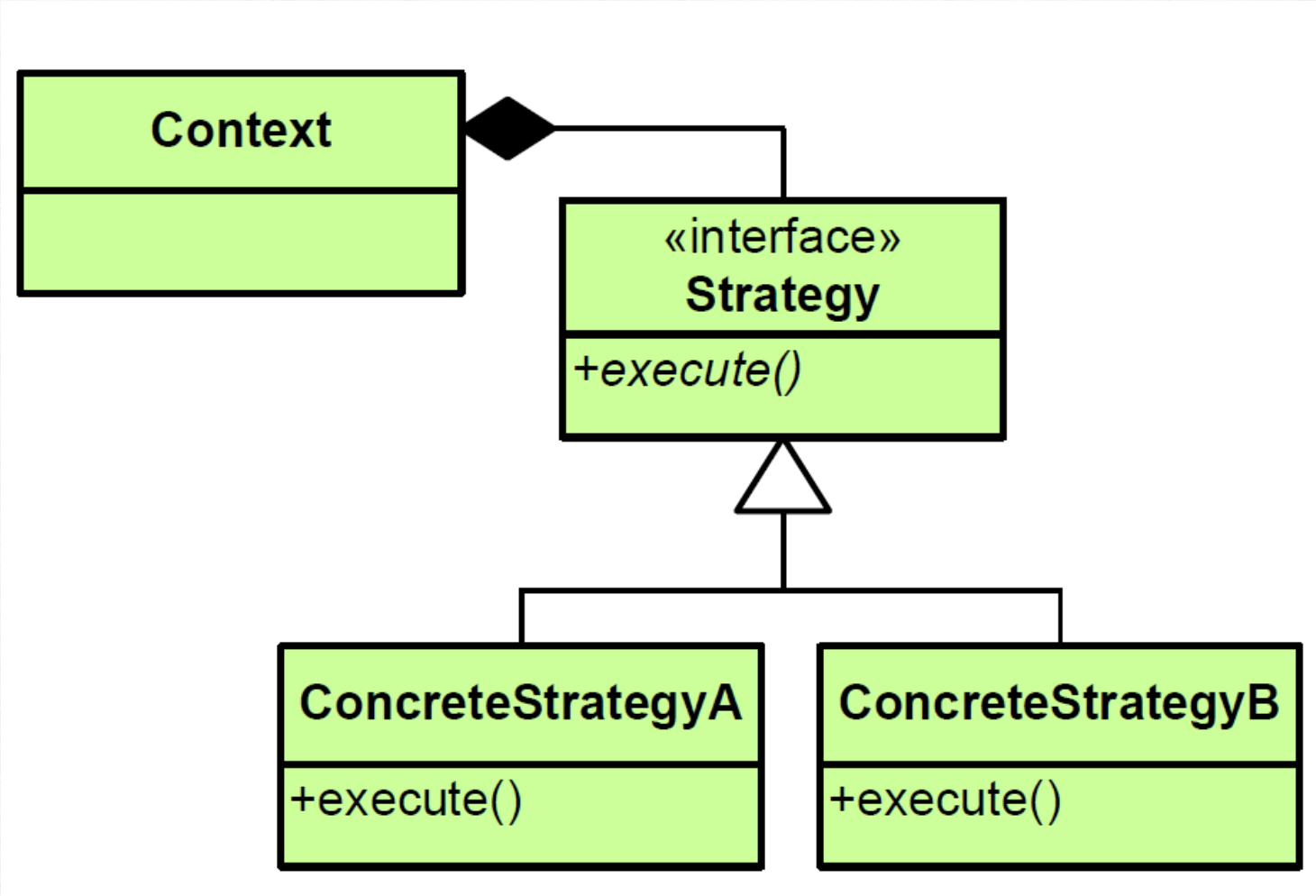
La sottoclasse ora può cambiare il valore di default per quel campo

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



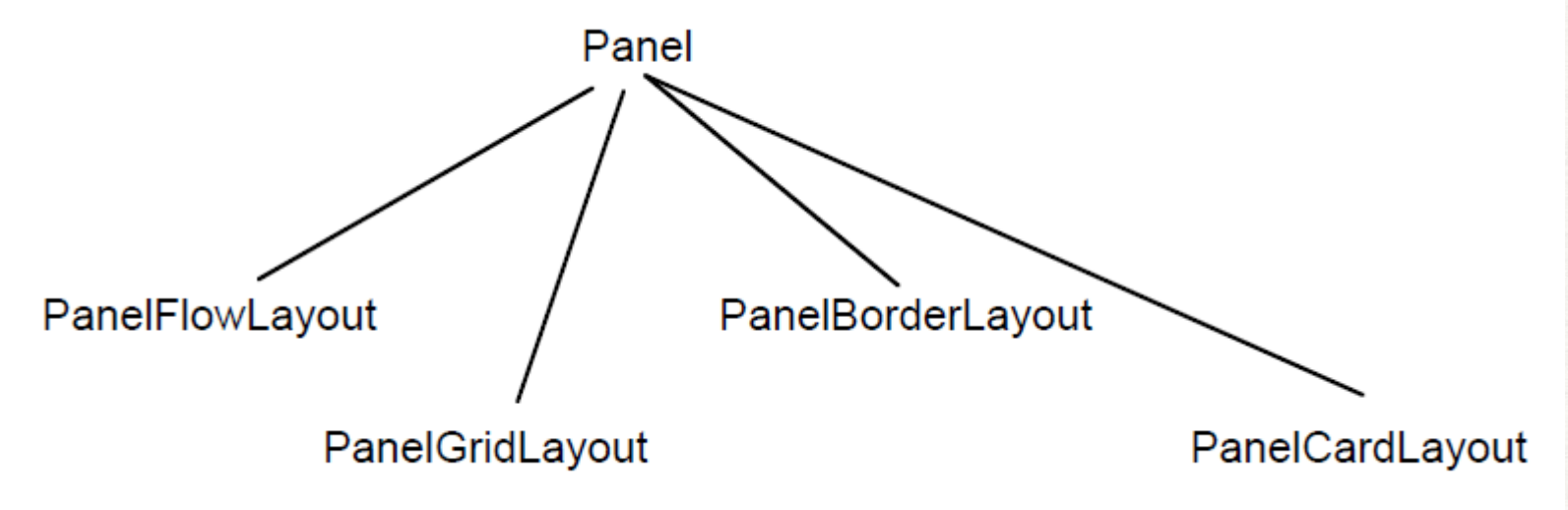
motivazione/esempi

- Java Layout managers per oggetti grafici
    - Posso scegliere con che stile disporre gli oggetti visuali in una finestra grafica, tra più stili disponibili
    - Nel contesto in cui lo applico, la scelta di un layout piuttosto che un altro non ha alcun impatto in termini di codice
- Java Comparators
    - Le interfacce Comparator e Comparable ci consentono di definire criteri di confronto e quindi ordinamento per tipi definiti dall'utente
    - Sono indipendenti dagli algoritmi che effettivamente implementano l'ordinamento e intercambiabili

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 sottoclassi?



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

Java Comparator

```
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());
  }
  }
```

Comparable **è l'interfaccia che definisce la strategia per confrontare un oggetto con altri oggetti dello stesso tipo. Quello che viene detto comunemente il "natural ordering" per quel tipo**

Creiamo alcuni oggetti

```
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);
   System.out.println("Before Sorting : " + footballTeam);
   Collections.sort(footballTeam);
   System.out.println("After Sorting : " + footballTeam);
}
```

Comparable

- **L'ordinamento è deciso dal valore di ritorno del metodo *compareTo()* .** La funzione *Integer.compare(x, y)* ritorna -1 se *x* è minore di *y*, 0 se sono uguali e 1 altrimenti.
- Con questo metodo posso definire il criterio e quindi l'effetto dell'algoritmo di ordinamento.
- l'algoritmo di ordinamento è un template method
- Solo una volta e per tutte, perchè l'ho fatto dentro la classe che definisce I mio tipo
- Se avessi voluto poter disporre di vari criteri tra cui scegliere liberamente in base al contesto? Strategy pattern…

Java Comparator vs Comparable

- **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());
      }
}
```

Il comparatore è una classe a sé rispetto alla classe che compara, non come prima

Ne definiamo un altro

Posso definire liberamente tutti i comparatori possibili/utili

```
public class PlayerAgeComparator implements
Comparator<Player> {

   @Override
   public int compare(Player firstPlayer, Player secondPlayer)
   {

      return Integer.compare(firstPlayer.getAge(),
    secondPlayer.getAge());
   }

   }
```

In questo modo non solo abbiamo ridefinito il criterio di ordinamento rispetto a quello “naturale” ma ne possiamo anche usare due o più sulla stessa struttura di oggetti

```
PlayerRankingComparator playerComparator = new PlayerRankingComparator();
Collections.sort(footballTeam, playerComparator);
```

```
PlayerAgeComparator playerComparator = new PlayerAgeComparator();
Collections.sort(footballTeam, playerComparator);
```

Da Java 8

- Nuovi modi di definire *Comparators* usando Lambda expressions, e il metodo static *comparing()*

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

Comparing()

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

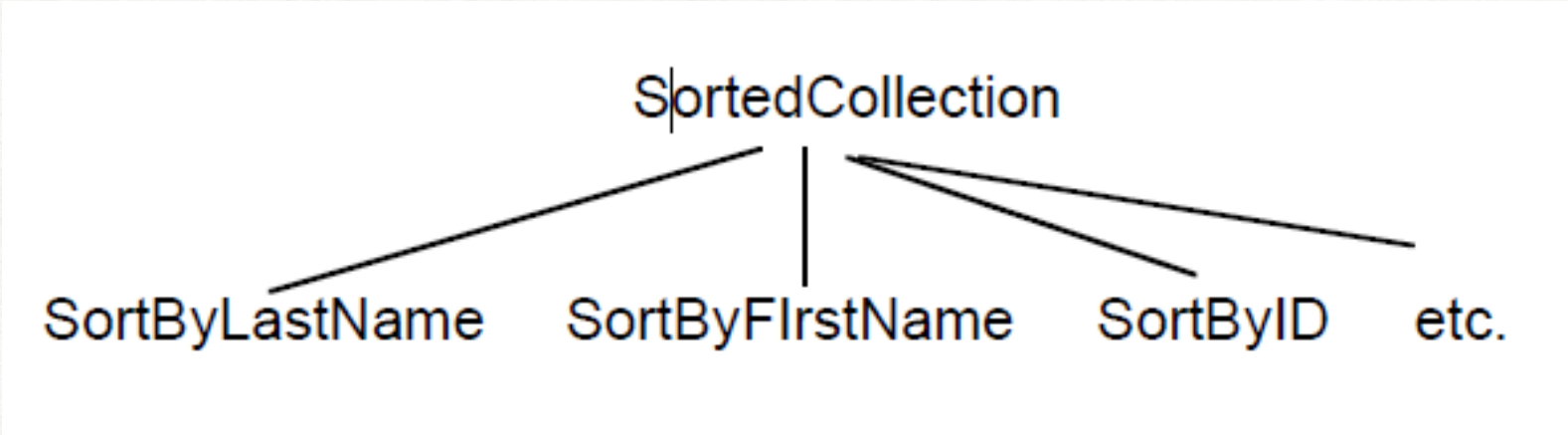
```
Comparator<Player> byRank = Comparator.comparing(Player::getRanking);

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

Comparator vs Comparable

- A volte non ci è possible modificare il codice sorgente delle classi I cui oggetti vogliamo ordinare rendendo *Comparable* impossible da usare
- Adottando il pattern strategy attraverso i *Comparators* ci ha permesso di evitare di aggiungere codice nelle classi del dominio applicativo
- Abbiamo potuto definire strategie di ordinamento multiple, cosa impossibile con la semplice interfaccia *Comparable*

- Vediamo l’applicazione completa…

Perché non usare la derivazione?



- 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 ed algoritmi di ordinamento

Applicabilità

- Uso lo strategy pattern quando
  - Mi servono versioni diverse di un algoritmo
  - Un algoritmo deve manipolare dati ma i client di questo algoritmo non dovrebbero accedervi
  - 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

Conseguenze

- Mi ritrovo con una famiglia di algoritmi
- Situazione alternativa al sottoclassare il contesto di utilizzo
- Elimino istruzioni condizionali multiple sostituendole con ***strategy.do()***
  - ***Invece che:***

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

- I client devono conoscere le strategie disponibili
- Contesto e strategia sono oggetti distinti che si messaggiano, invece che una stessa classe: overhead di comunicazione.

Implementazione

- Definisco le interfacce delle strategie e del contesto
  - Definisco come interagiscono
  - Il contesto può passare dati alla strategia come parametro
  - Oppure La strategia deve ricevere un puntatore al contesto per prenderli da sé
  - Possibile l'uso di ***inner*** class
- 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**

Inner class

- Classi definite come membri di altre classi

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

In quanto membri di un classe possono essere ***private***

Ed hanno accesso agli altri membri privati della classe outer

Sono istanziabili solo nel contesto della classe outer

Oppure possono essere ***anonime***

Posso usarle per limitare alle sole strategie l'accesso ai dati privati del contesto

esempio

```
interface Strategy {
        String appliedStrategy(int x);
}

public class Context{
        // dati privati da non rivelare se non alla strategia
        private int data = 10;

        public void executeTask(Strategy s) {// strategia ricevuta come parametro
                System.out.println(s.appliedStrategy(data) + ", has been used to execute requested task");
        }

        public static void main(String args[]) {
                Context c = new Context();

                //passo come inner class una istanza anonima sottoclasse di Strategy
                c.executeTask( new Strategy()  {
                                                        public String appliedStrategy(int x) {
                                                            return x+" squared";
                                                        }
                                                }
                );
        }
}
```

Solo le sottoclassi di strategy passate al contesto accedono al dato privato.

Proviamolo al pc

Template vs strategy

- Template per fissare lo scheletro di un algoritmo i cui vari passi sono implementabili in vari modi, risultando in una famiglia di algoritmi customizzati, coesistenti (class scope)
- Strategy per disporre di una famiglia di algoritmi correlati applicabili in modo intercambiabile, ma uno per volta, ad uno stesso contesto (un altro algoritmo) o a contesti diversi, customizzandone l'esecuzione (object scope)
- Entrambi sono pattern comportamentali: riguardano problematiche algoritmiche