



UNIVERSITÀ DEGLI STUDI
DI MILANO

Ingegneria del software

Progettazione

Estraiamo le interfacce

```
public static List<Card> drawCards(Deck deck, int number) {  
    List<Card> result = new ArrayList<>();  
    for (int i = 0; i < number && !deck.isEmpty(); i++) {  
        result.add(deck.draw());  
    }  
    return result;  
}
```

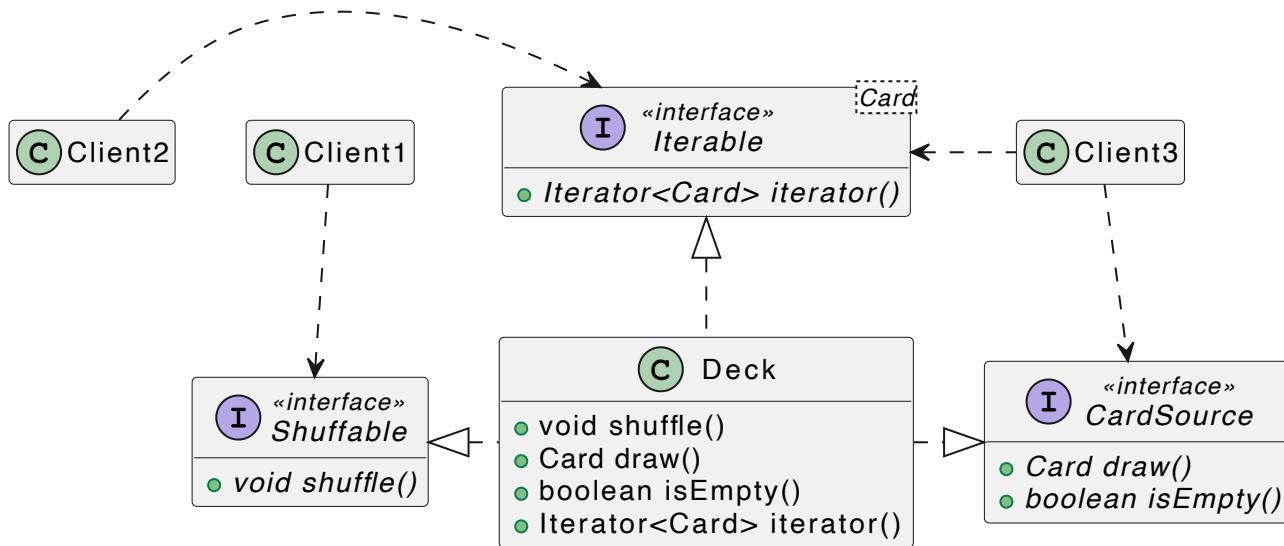
Questo metodo funziona solo per gli oggetti di tipo Deck

In realtà cosa gli interessa?

solamente che l'oggetto abbia i due metodi

- `isEmpty()` che controlla se ci sono ancora carte
- `draw()` che restituisce la prossima carta (e la toglie dal mazzo)

Interface segregation



Interfaccia e Polimorfismo

```
1  public interface CardSource {  
2      /**  
3      * @return The next available card.  
4      * @pre !isEmpty()  
5      */  
6      Card draw();  
7  
8      /**  
9      * @return True if there is no card in the source.  
10     */  
11     boolean isEmpty();  
12 }  
13  
14 public class Deck implements CardSource { . . . }  
15  
16 public static List<Card> drawCards(CardSource deck, int number) {  
17     List<Card> result = new ArrayList<>();  
18     for (int i = 0; i < number && !deck.isEmpty(); i++) {  
19         result.add(deck.draw());  
20     }  
21     return result;  
22 }
```



Polimorfismo e *loose coupling*

- La capacità di un identificatore di variabile/parametro di accettare oggetti in realtà di forme diverse, a patto che siano dei suoi sottotipi

```
Deck deck = new Deck();  
  
CardSource source = deck;  
  
List<Card> cards;  
cards = drawCards(deck, 5);
```

Collegamento dinamico e *extensibility*

```
public static List<Card> drawCards(CardSource cardSource, int number) {  
    List<Card> result = new ArrayList<>();  
    for (int i = 0; i < number && !cardSource.isEmpty(); i++) {  
        result.add(cardSource.draw());  
    }  
    return result;  
}
```

- Il compilatore non sa a tempo di compilazione quale metodo draw() dovrà chiamare
- Rimanda la decisione al momento effettivo della esecuzione

Permette di chiamare codice *non ancora scritto*

Collegato quindi a estendibilità, a *Open Close Principle*



Esempio nella libreria standard di Java

```
public class Deck {  
    private List<Card> cards = new ArrayList<>();  
  
    public void shuffle() { Collections.shuffle(cards); }  
}
```

- cards è un oggetto di tipo ArrayList<Card>
- che implementa interfaccia List<Card>
- che è passabile come parametro a shuffle di Collections

```
public static void shuffle(List<?> list)
```

- shuffle accetta List di qualunque elemento

Infatti per *disordinare* una lista di elementi non dobbiamo mica guardarli...

DOMANDA: per riordinare un mazzo di carte, ci sarà una funzione equivalente da sfruttare?

```
public void sort() { Collections.sort(cards); }
```



Comparable interface

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

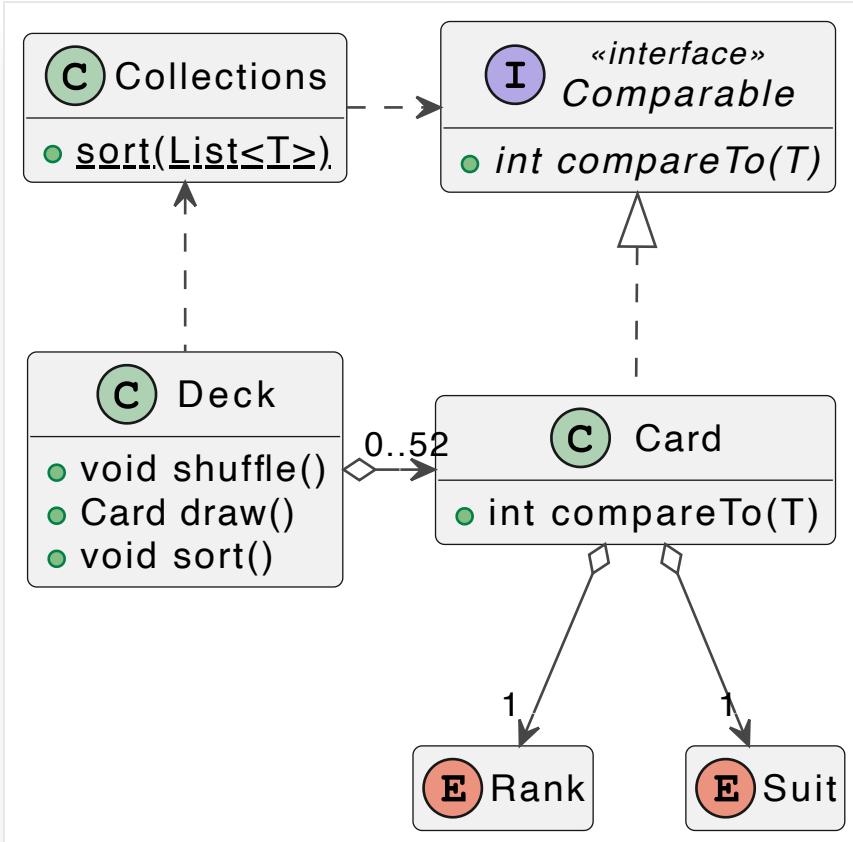
- Definisce un singolo metodo che restituisce un valore minore, uguale o maggiore a 0, a seconda del risultato del confronto.
- Questo metodo è quello che userà la `sort` di Collections.

Quindi la interfaccia di `Collections.sort` è:

```
public static <T extends Comparable<? super T>> void sort(List<T> list);
```

Cioè `sort` ha bisogno che gli elementi della lista da ordinare implementino Comparable, e quindi siano confrontabili tra di loro.

Esempio di diagramma delle classi UML



- classi e interfacce
 - attributi
 - metodi
- relazioni
 - generalizzazione e implementazione
 - associazione, aggregazione e composizione
 - dipendenze

Patterns

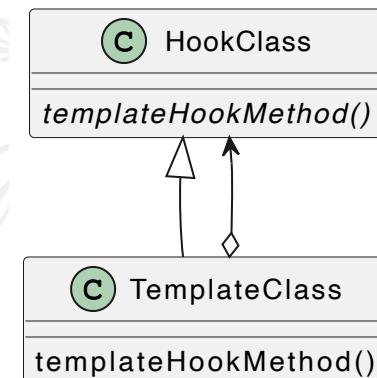
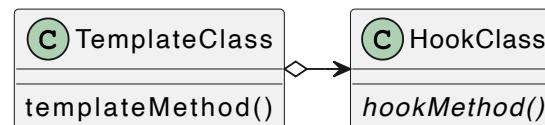
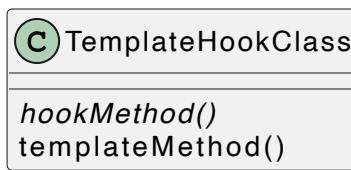
- Soluzioni a problemi ricorrenti
- strumento concettuale
 - che cattura la soluzione per una famiglia di problemi
 - che esprime architetture vincenti

Meta patterns

- Identifica due elementi base:
 - **HookMethod:** metodo astratto che determina il comportamento specifico nelle sottoclassi
 - È un *punto caldo* in cui si può intervenire per personalizzare, adattare lo schema
 - **TemplateMethod:** metodo che coordina generalmente più hook method
 - È l'*elemento freddo*, l'elemento di invariabilità del pattern

Come si relazionano *hook* e *template*

- **Unification:** template e hook sono nella stessa classe del framework
- **Connection:** hook e template sono in classi separate indicate rispettivamente come hook class e template class tra di loro collegate da una associazione
- **Recursive connection:** hook e template sono in classi tra di loro collegate anche tramite relazione di generalizzazione



Gang of Four Patterns

Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides

Definiscono 23 pattern e li classificano in tre categorie

- Creazionali
 - creazione degli oggetti
- Comportamentali
 - interazione tra gli oggetti
- Strutturali
 - Composizione di classi e oggetti



SINGLETON pattern



Singleton

- `instance : Singleton`
- ◆ `Singleton()`
- `Singleton.getInstance()`
- `sampleOp()`

- Si vuole avere un oggetto e non una classe.
- In un linguaggio che fornisce solo classi
- Si vuole rendere la classe responsabile del fatto che non può esistere più di una istanza

```
public class Singleton {  
    protected Singleton() {}  
    private static Singleton instance = null;  
    public static Singleton getInstance(){  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
    public void sampleOp() {...}  
}
```

SINGLETON Java Idiom

```
public enum MySingleton {  
    INSTANCE;  
    public void sampleOp() {...}  
}  
  
MySingleton.INSTANCE.sampleOp();
```

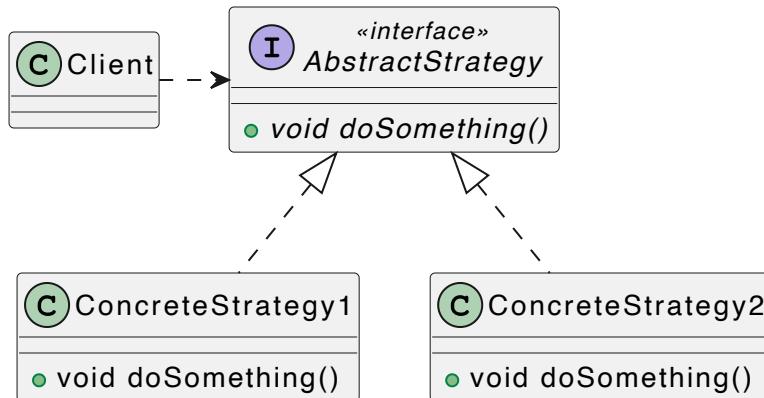
Sfrutta il fatto che in Java i campi degli enumerativi sono realizzati tramite degli oggetti costanti creati al momento del loro primo uso.

È perciò un "idioma" perché soluzione dipendente da uno specifico linguaggio e non architettura generale.

Chiaramente se dovete realizzare un **SINGLETON** in Java, è sicuramente l'approccio da usare.

Delegation/Strategy

Definisce una famiglia di algoritmi, e li rende (tramite encapsulation) tra di loro intercambiabili

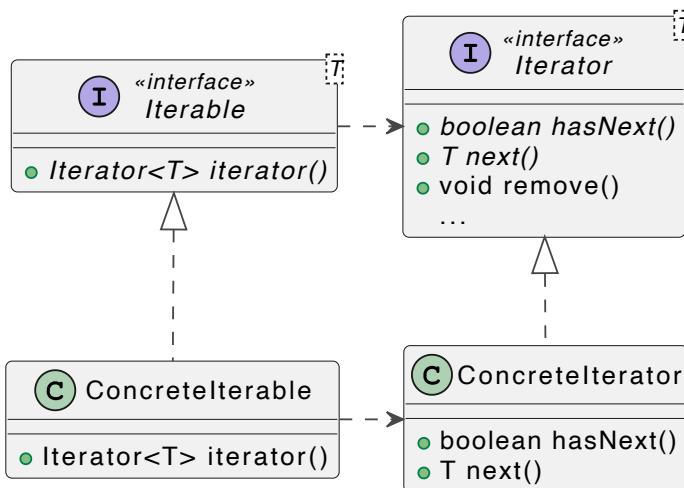


Abbiamo visto `sort` di `Collections` che richiedeva un parametro `Comparable`, una altra possibilità è il metodo `sort` con un secondo parametro `Comparator`

```
static <T> void sort(List<T> list, Comparator<? super T> c)
```

ITERATOR pattern

Fornisce un modo di accedere agli elementi di un oggetto aggregatore in maniera sequenziale senza esporre la rappresentazione interna



```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

Iteratori

- se `getCards()` ritorna un `Iterator<Card>` possiamo scrivere

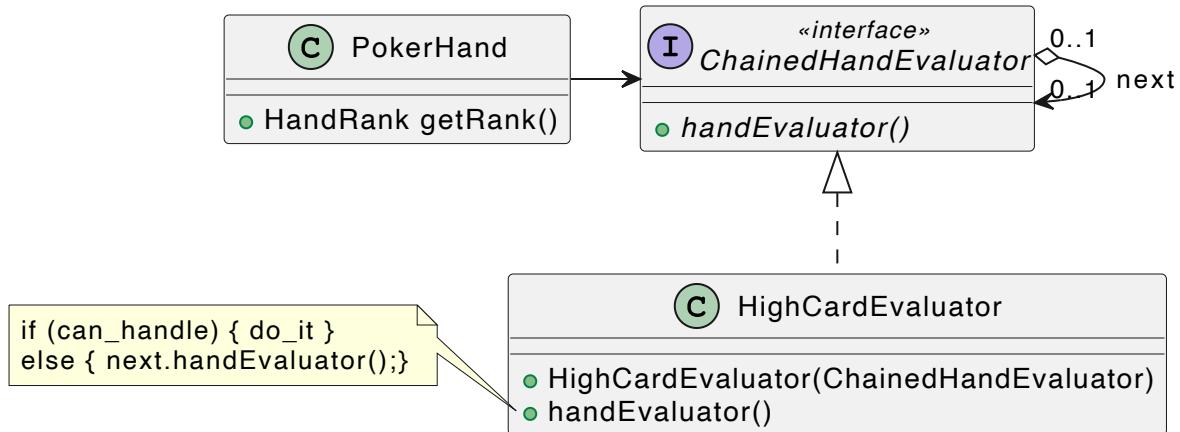
```
Iterator<Card> cardIterator = deck.getCards();
while (cardIterator.hasNext()) {
    Card card = cardIterator.next();
    System.out.println(card.getSuit());
}
```

- ma se invece di `getCards()` lo chiamiamo `iterator()` in modo da aderire alla interfaccia `Iterable` allora possiamo usare il costrutto *for esteso*

```
for (Card card : deck)
    System.out.println(card.getSuit());
```

CHAIN OF RESPONSABILITY pattern

Permette di definire una catena di potenziali gestori di una richiesta di cui non sappiamo a priori chi sarà in grado di gestirla effettivamente



```

public class ForthInterpreter implements Interpreter {
    private final Deque<Integer> stack = new ArrayDeque<>();
    private final Map<String, Consumer<ForthInterpreter>> op = new HashMap<>();

    void addOperator(String operator, Consumer<ForthInterpreter> lambda) { op.put(operator, lambda); }

    public ForthInterpreter() {
        op.put("+", (forth) -> forth.push(forth.pop() + forth.pop()));
        op.put("-", (forth) -> forth.push(-forth.pop() + forth.pop()));
    }
}

```

PARTE DEL CODICE PUNTO DI ARRIVO DI QUANTO CODIFICATO IN CLASSE

```

void push(Integer num) { stack.push(num); }
Integer pop() { return stack.pop(); }
Integer peek() { return stack.peek(); }

```

Vista (limitata alle classi del package)
di un sottoinsieme delle operazioni
disponibili sulla deque per permettere
di definire nuovi operatori del Forth

```

void parseAndExecute(String program) {
    Scanner sc = new Scanner(program);
    while (sc.hasNext()) {
        String item = sc.next();
        try {
            if (op.containsKey(item))
                op.get(item).accept(this);
            else
                stack.push(Integer.valueOf(item));
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Token error '" + item);
        } catch (NoSuchElementException e) {
            throw new IllegalArgumentException("Stack Underflow");
        }
    }
}

```

Vista (limitata alle classi del package)
della funzione per eseguire possibili macro



Template method della classe ForthInterpreter con chiamata all'**Hook Method** che permette la definizione di operazioni di preprocessing del programma.

Definizione della classe

ExtendedForthInterpreter con:

- aggiunta di nuovi operatori dentro al costruttore
- implementazione dell'**Hook Method** per il parsing delle definizioni delle funzioni macro all'inizio del programma

```
@Override  
public void input(String program) {  
    stack.clear();  
    program = registerMacros(program);  
    parseAndExecute(program);  
}  
  
public class ExtendedForthInterpreter extends ForthInterpreter {  
    String registerMacros(String program) {  
        Scanner sc = new Scanner(program);  
        sc.useDelimiter(";;");  
        while (sc.hasNext()) {  
            String element = sc.next().trim();  
            if (element.startsWith(":")) {  
                var el = element.split(" ", 3);  
                addOperator(el[1], forth -> forth.parseAndExecute(el[2]));  
            } else  
                return element;  
        }  
        return program;  
    }  
  
    public ExtendedForthInterpreter() {  
        addOperator("*", (forth) -> forth.push(forth.pop() * forth.pop()));  
        addOperator("dup", (forth) -> forth.push(forth.peek()));  
    }  
}
```