Comparable interface

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

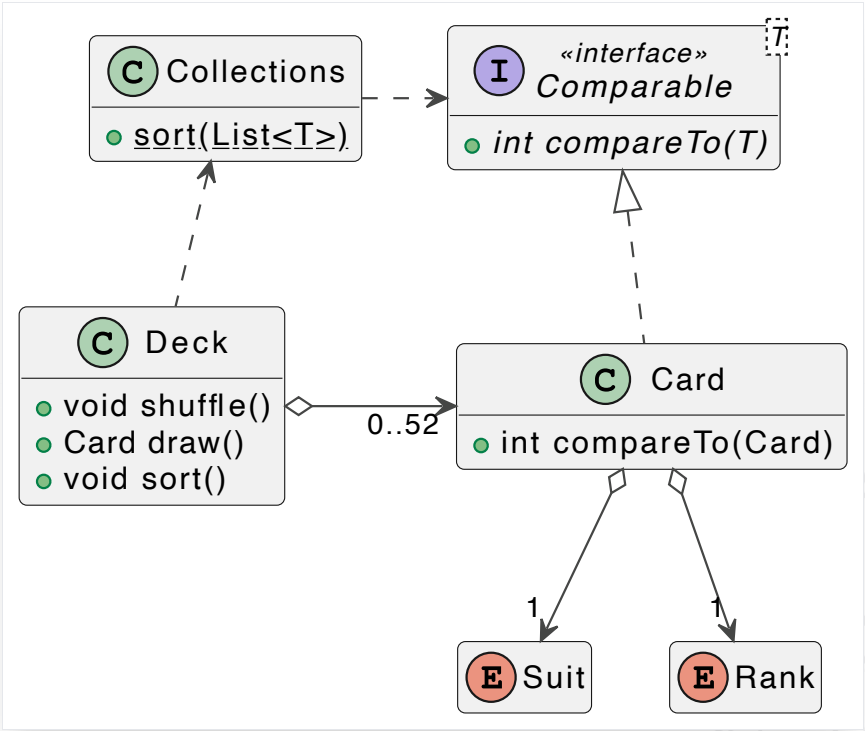
- Definisce un singolo metodo che restituisce un valore minore, uguale o maggiore a 0, a seconda del risultato del confronto.
- Questo metodo è quello che userà la `sort` di Collections.

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

Pattern

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

Antipattern

- La denuncia di una soluzione sbagliata (ma che sembrava ragionevole) ad un problema

Idiom

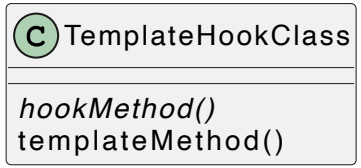
Come un pattern, ma non indipendente dal linguaggio

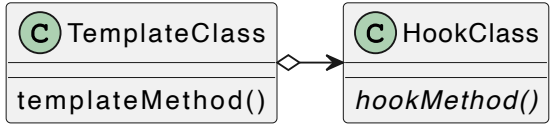
Meta patterns

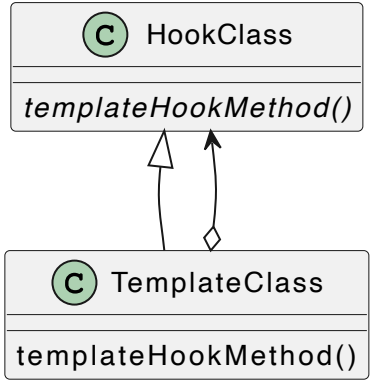
- Identifica due elementi base:
  - **HookMethod**: metodo astratto che determina il comportamento specifico nelle sottoclassi
    - È un *punto caldo* in cui si può intervenire per personalizzare, adattare lo schema
  - **TemplateMethod**: metodo che coordina generalmente più hook method
    - È l'*elemento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







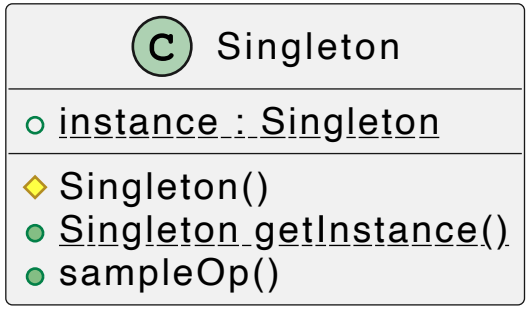
Gang of Four Patterns

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



- 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() {...}
}
```

NON THREAD SAFE

Singleton e thread safeness

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

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

SINGLETON Java Idiom

```java
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.

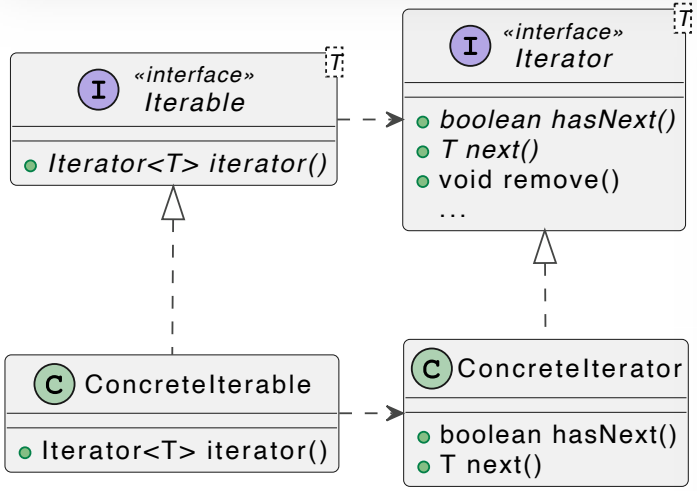
È perciò un "idioma" perché soluzione dipendente da uno specifico linguaggio e non architettura generale.

Leggibile?

Normalmente direi di no, ma ormai si "conosce", quindi se dovete realizzare un SINGLETON in Java, è sicuramente l'approccio da usare.

ITERATOR pattern

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




```java
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

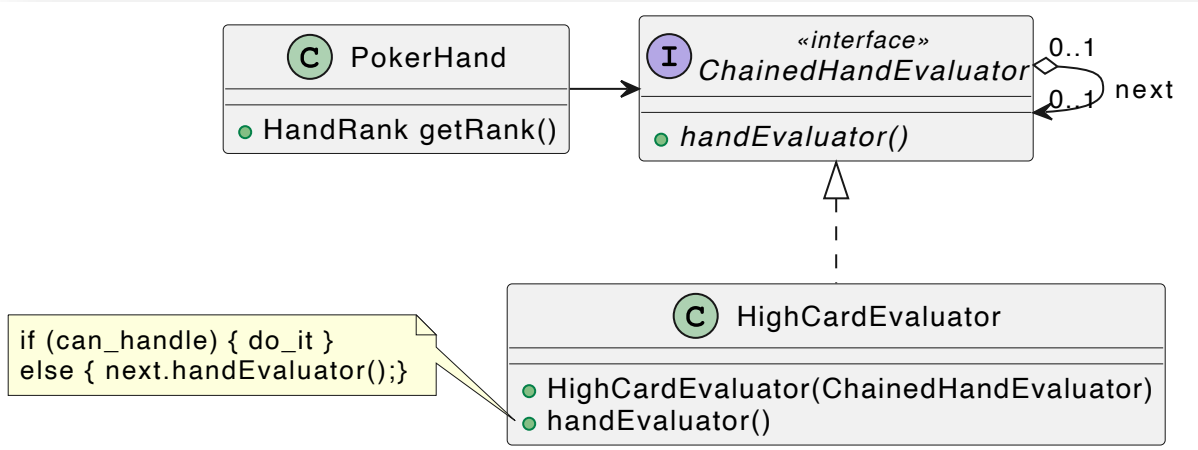
```
Iterator<Card> cardIterator = deck.getCards();
while (cardIterator.hasNext()) {
  Card card = cardIterator.next();
  System.out.println(card.getSuit());
}
```

- 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



Nullability

Ad una variabile che indica un *riferimento* a un oggetto (quindi in Java **sempre** a parte i *tipi primitivi*), allora possiamo assegnare il valore speciale `null`, per dire che … *non punta a nulla*.

```
Card card = null;  // oppure Card.rank = null;
```

Il problema si manifesta quando proviamo a *dereferenziare* la variabile e non puntando a nulla riceviamo un errore (una `NullPointerException`)

- un parametro può essere `null` o posso assumere che punti a un dato "valido"?

Il "problema" è che `null` viene usato per indicare diverse cose:

- errore
- stato temporaneamente inconsistente
- valore assente

Un codice chiaro non dovrebbe far uso di ***null*** o almeno limitarlo

Senza valori "assenti"

```java
public class Card {
  private Rank rank; // Should never be null
  private Suit suit; // Should never be null

  public Card(Rank rank, Suit suit) {
    this.rank = rank;
    this.suit = suit;
  }
}
```

Come possiamo "tradurre" in codice questi commenti?

```java
public Card(Rank rank, Suit suit) {
  if (rank != null && suit != null) {
    this.rank = rank;
    this.suit = suit;
  }
}
```

SBAGLIATO

```java
public Card(Rank rank, Suit suit) {
  if (rank == null || suit == null)
    throw new IllegalArgumentException();
  this.rank = rank;
  this.suit = suit;
}
```

```java
public Card(Rank rank, Suit suit) {
  assert rank != null && suit != null;
  this.rank = rank;
  this.suit = suit;
}
```

```java
final @NotNull private Rank rank;
final @NotNull private Suit suit;

public Card(@NotNull Rank rank, @NotNull Suit suit) {
  this.rank = rank;
  this.suit = suit;
}
```

Con valori "assenti"

Aggiungiamo la carta **Joker** che non ha nè *Suit* nè *Rank*

```
public class Card {
  private Rank rank;
  private Suit suit;
  private boolean isJoker;
  public boolean isJoker() { return isJoker; }
  public Rank getRank() { return rank; }
  public Suit getSuit() { return suit; }
}
```

Cosa ritorano i *getter* se è un Joker?

- null
  - è quello che stiamo sconsigliando
- un valore qualsiasi (tanto controlleremo prima se `isJoker() == true`)
  - confusione e probabili errori di uso
- aggiungo valore enumerativo: `NONE` … ma ci sarebbero 5 segni e 14 rank

NullObject pattern

Vogliamo creare un oggetto che corrisponda al concetto "nessun valore" o "valore neutro"

```
public interface CardSource {
  Card draw();
  boolean isEmpty();

  public static CardSource NULL = new CardSource() {
    public boolean isEmpty() { return true; }
    public Card draw() {
      assert !isEmpty();
      return null;
    }
  };
}
```

`CardSource.NULL` è un oggetto valido di un tipo anonimo che aderisce alla interfaccia `CardSource` ma che ha particolari implementazioni per i vari metodi

Serve ad evitare di dover trattare separatamente il caso `== null`, ma se proprio diventasse necessario si può sempre testare `== CardSource.NULL`