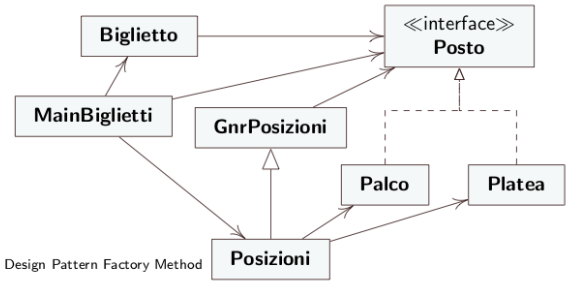
30-03-2023

Esempio di Factory Method




```
// Codice Java che implementa il design pattern Factory Method

// Posto e' un Product
public interface Posto { // è tipo una matrice che avrà la coppia (lettera,riga) oppure
(numero)
        public String getPosizione();
        public int getCosto();
        public String getSettore();
}
```

```
// Palco e' un ConcreteProduct
import java.util.Random;

public class Palco implements Posto {
        private final int numero;

        public Palco() {
                numero = new Random().nextInt(20) + 1;
        }

        @Override
        public int getCosto() {
                if (numero > 10) return 50;
                return 40;
        }

        @Override
        public String getPosizione() {
                return Integer.toString(numero);
        }

        @Override
        public String getSettore() {
                if (numero == 20) return "Centrale";
                if (numero > 10) return "Verde";
                return "Blu";
        }
}
```

```java
// Platea e' un ConcreteProduct
import java.util.Random;

public class Platea implements Posto {
	private final String[] nomi = { "A", "B", "C", "D", "E", "F" };
	private final int numero;
	private final int riga;

	public Platea() { //indica un posto ben preciso.
		numero = new Random().nextInt(10) + 1;
		riga = new Random().nextInt(5) + 1;
	}

	@Override
	public int getCosto() { //fasce di costo diverso a seconda della posizione
		if (numero > 5 && rigaMax()) return 100;
		if (numero > 5 && rigaMin()) return 80;
		return 60;
	}

	@Override
	public String getPosizione() {
		return nomi[riga].concat(Integer.toString(numero));
	}

	@Override
	public String getSettore() {
		if (riga == 0) return "Riservato";
		return "Normale";
	}

	private boolean rigaMax() {
		return (riga >= 1 && riga <= 4);
	}

	private boolean rigaMin() {
		return (riga == 0 || riga == 5);
	}
}
```

```java
// GnrPosizioni e' un Creator
import java.util.Set;
import java.util.TreeSet;
public abstract class GnrPosizioni { // versione 1.1
	private static final int MAXPOSTI = 100;
	private final Set< String > pst = new TreeSet< >(); // set di posti -> Il controllo è più veloce rispetto a una List. L'ordine non è importante e NON CI SONO DUPLICATI

	// metodo che rimanda alla sottoclasse l'istanziazione della classe
	public Posto prendiNumero(int x) {
		if (pst.size() == MAXPOSTI) return null;
		// il chiamante dovrebbe controllare se null
		Posto p;
		do {                                // fino a quando non trova un posto libero
			p = getPosto(x); // chiama metodo della sottoclasse
```

```
                } while (pst.contains(p.getPosizione()));
                pst.add(p.getPosizione());
                return p;
        }

        public void printPostiOccupati() {
                for (String s : pst)
                        System.out.print(s + " ");
        }

        // il metodo factory e' dichiarato ma non implementato
        public abstract Posto getPosto(int tipo);
}
```

```
// Posizioni e' un ConcreteCreator con un metodo factory
public class Posizioni extends GnrPosizioni {
        // metodo factory che ritorna una istanza
        @Override
        public Posto getPosto(int tipo) {
                // crea l'istanza di un ConcreteProduct
                if (1 == tipo) return new Palco();
                return new Platea();
        }
}
```

```
// Biglietto e' un client del Product Posto
public class Biglietto {
        private String nome;
        private final Posto pos;

        public Biglietto(Posto p) {
                pos = p;
        }

        public void intesta(String s) {
                nome = s;
        }

        public String getDettagli() {
                return nome.concat(" ").concat(pos.getPosizione());
        }

        public String getNome() {
                return nome;
        }

        public int getCosto() {
                return pos.getCosto();
        }
}
```

```
// Classe con il main che usa il ConcreteCreator
public class MainBiglietti {
        private static Posizioni cp = new Posizioni();
```

```
        public static void main(String[] args) {
                Posto pos = cp.prendiNumero(0);
                Biglietto b = new Biglietto(pos);
                b.intesta("Mario");
                System.out.println("Costo " + b.getCosto());

                new Biglietto(cp.prendiNumero(0));
                new Biglietto(cp.prendiNumero(0));
                cp.printPostiOccupati();
        }
}
```

`@Override` è detta **ANNOTAZIONE**. Devono stare in posti ben precisi: subito prima della dichiarazione/implementazione di un *metodo/classe/passaggio di parametri nei metodi*. Sono:

- predefinite da JAVA
- create dall'utente

`@Override` è **predefinita**. Serve per evitare errori in fase di pre-compilazione.
A run-time le annotazioni potrebbero essere state eliminate dal compilatore. Servono solo come informazioni aggiuntive ma non servono a run-time. Servono per fare in modo che il compilatore faccia **un controllo in più** o evita di segnalare errori dove effettivamente non ce ne sono. Assicura che effettivamente sto facendo un override e sto definendo un metodo della superclasse.

> *Vuol dire che il metodo deve essere stato dichiarato nell'interfaccia che è super-tipo della classe che si sta implementando*

Design pattern Adapter (object adapter e class adapter)

Questo è un design patter **STRUTTURALE**. I precedenti visti erano **CREAZIONALI**.

Intento

Converte l'interfaccia di una classe in un altra interfaccia che si aspetta il client. Il client si aspetta una certa interfaccia (un certo nome di metodo con un certo tipo di parametro in ingresso). Quindi c'è **INCOMPATIBILITA'** fra la richiesta del client e l'effettiva implementazione.

Problema

Si è progettato un sistema software facendo riferimento a una certa interfaccia da usare.

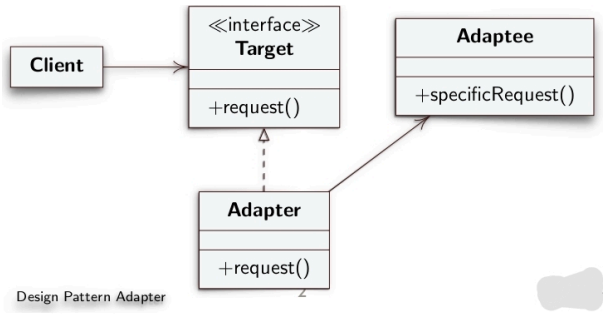
> Si è implementato, per esempio, un metodo getCosto() ma effettivamente serve un getCosto(`parametro`).

- La parte che deve fornire il risultato(il chiamante) **NON può essere modificata**.
- Cambiare l'interfaccia in modo da adattarsi ai requisiti potrebbe creare problemi se tali metodi sono usati anche altrove.
    - **Se modifico un metodo**, questo stesso deve venire modificato in modo da attenersi all'attività che deve svolgere.

Esempio

- *Alcune volte una classe di una libreria non può essere usata poiché incompatibile con l'interfaccia che si aspetta l'applicazione. Ovvero nome metodo, parametri, tipo parametri di chiamate all'interno dell'applicazione non sono corrispondenti a quelli offerti da una classe di libreria.*
- *Inoltre non è possibile cambiare l'interfaccia della libreria, poiché non si ha il sorgente (comunque non conviene cambiarla*

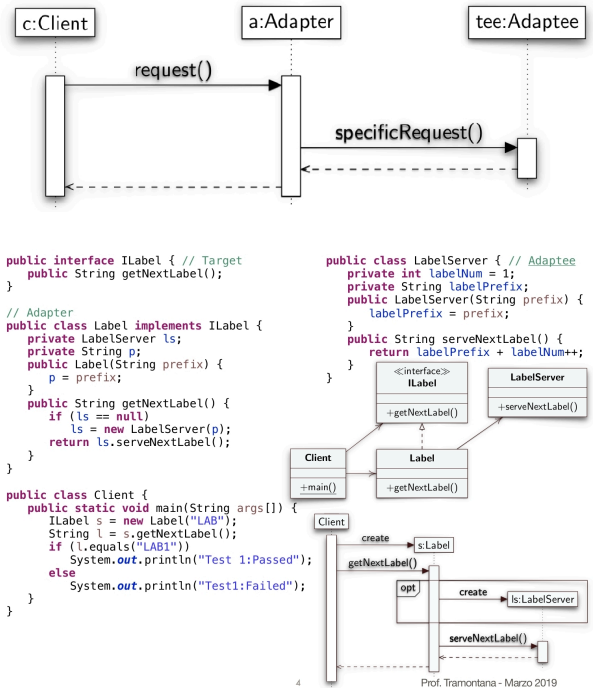
Soluzione



Crea una nuova classe Adapter che farà da adattatore fra invocazioni del client e server.

- I metodi saranno **conformi al client** (*chiamante*).
- La classe che mette a disposizione i metodi saranno conosciuti all'interno della classe Adapter. Quindi da Adapter si chiamano i metodi effettivi.
- Il client si interfaccia con **Target** che è la classe che si aspetta
- Adaptee sarà la classe di libreria che mette a disposizione il servizio che risulta **INCOMPATIBILE** con il Client.

(ESAME): Pronunciare bene il nome delle classi per evitare fraintendimenti. -> *Adapter* e Adaptee (*Adaptìi*).



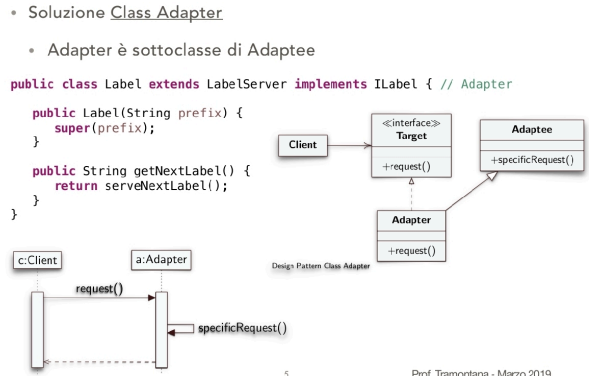
> Si crea **un Adapter per ogni classe che devo adattare**.
> *Se ho **2 librerie** da adattare allora ho 2 **Adapter diversi**.*

- `opt` sta a significare che, quando si deve istanziare `ls` non è detto che essa venga fatta perchè c'è un'istruzione condizionale e quindi potrebbe non venire fatta.

- Questo è un **Object Adapter** perchè si ha **l'invocazione fra oggetti**

Variante Class Adapter

- Adapter è sottoclasse di Adaptee.
- Adapter implementa l'interfaccia Target (*come prima*)
- Si possono liberamente invocare i metodi della superclasse e quindi non serve più un riferimento ad `Adaptee` (un oggetto `Adaptee`)
- Quando devo invocare un metodo di Adaptee, non serve più l'istanza ma invoco il metodo sulla **stessa istanza**



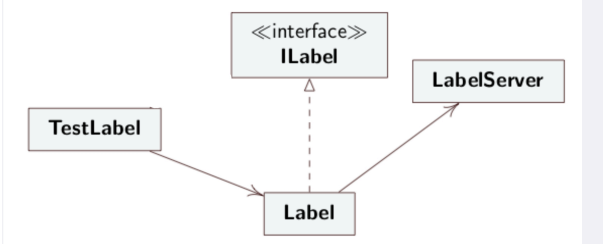
Differenze Object Adapter e Class Adapter

- Dipendono dai requisiti, ovviamente, e dalle scelte di progettazione
- si crea un'unica istanza di Adapter che **contiene tutto**. Se la superclasse richiede molta memoria allora si inizializza anche Adaptee allo stesso momento e quindi si spende quel tempo di inizializzazione (*a prescindere*)
- In Object Adapter, invece, **rimando l'inizializzazione fino a poco prima dell'invocazione del metodo**. (quindi il più tardi possibile, in modo da non far perdere tempo al Client)
  - Adaptee potrebbe non essere mai inizializzato (*in base all'uso del software*) e quindi si **risparmia tempo**. Questo *modo di fare* è detto **LAZY INITIALISATION** (cioè **inizializzazione in ritardo, pigra**)
- **Ci sono MOMENTI DIVERSI di inizializzazione**

> In Java non è concessa l'ereditarietà multipla per semplificare la scrittura del codice e per evitare incompatibilità fra metodi ereditati da una classe `A` o `B` (**aventi la stessa *signature***)
> **Si può, invece, ereditare da una classe A e da un'interfaccia B**.

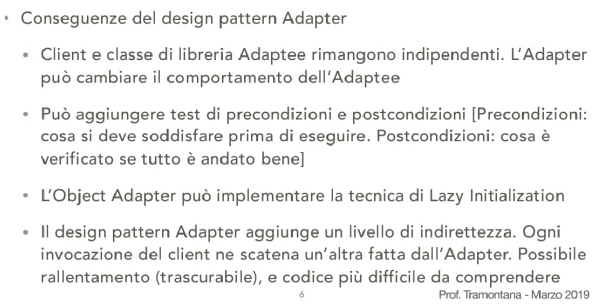
Variante Adapter a due vie

Fornisce sia l'interfaccia di Target e sia l'intefaccia di Adaptee. Se un client si riferisce a due interfacce. Usando il object adapter uso questa versione e al suo interno istanzio Adaptee (*LabelServer*)



- Il ***Client*** deve invocare i **metodi dell'intefaccia *Target* e i metodi dell'interfaccia *Adaptee***
- *Client* si interfaccia con *Adapter* essendo più "ricca" di informazioni rispetto a *Target*

Conseguenze



- **PRECONDIZIONE**: deve verificarsi prima dell'invocazione di un metodo per consentire al metodo di essere **BEN ESEGUITO**.
  - Se queste precondizioni possono essere scritte, **evito** di fare chiamate di metodo che molto **probabilmente incontra errori di esecuzione** o non ritorna il dato voluto.
  - Le precondizioni devono essere individuate durante la progettazione. Vengono messe nel Adapter per non farlo fare al Client.
  - Se si vuole semplificare l'Adaptee, tali controlli si fanno sempre nell'Adapter
- **POSTCONDIZIONI**: si devono verificare **se la condizione è andata a BUON FINE**.
  - *Per esempio, **un valore di ritorno** non deve essere mai nullo oppure deve essere di un certo tipo.*

L'Adapter introduce un livello di dipendenza: ogni volta che viene usato, esso chiama un metodo da Adaptee. Non si passa da chiamante->chiamato ma si aggiunge qualcosa **in mezzo**. In questo caso, a livello di leggibilità di codice, essa viene meno perchè magari Adapter non è nemmeno documentato perchè fa solo da "*tramite*".