

Design pattern Adapter (object adapter e class adapter)

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

Intento *Converte un'interfaccia in modo da adattarla al tipo che si aspetta il client.*

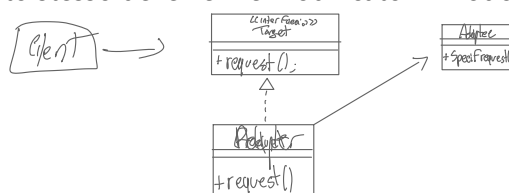
Converte l'interfaccia di una classe in un'altra interfaccia che si aspetta il client. Il client si aspetta una certa interfaccia (un certo nome di metodo con un certo tipo di parametro in ingresso). Quindi c'è **INCOMPATIBILITÀ** fra la richiesta del client e l'effettiva implementazione.

Problema *(Abbiamo un'interfaccia che non va bene al client, NON POSSIAMO MODIFICARE METODI)
IL CHIAMANTE NON PUÒ ESSERE MODIFICATO.*

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)`.

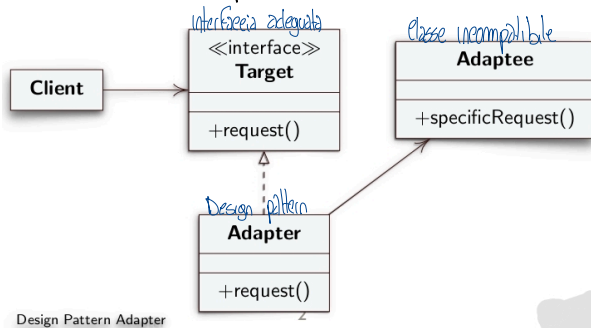
- La parte che deve fornire il risultato (il chiamante) **NON può essere modificata.**
- Cambiare l'interfaccia in modo da adattarsi ai requisiti potrebbe creare problemi se tali metodi sono usati anche altrove.
 - **Se modifico un metodo, questo stesso deve venire modificato in modo da attenersi all'attività 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 classe adapter che faccia da adattatore (Metodi conformi al CLIENT) La classe che mette a disposizione i metodi saranno conosciuti da adapter

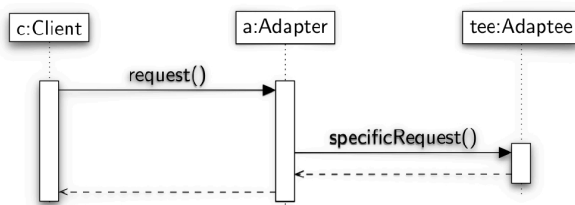


Si deve CREARE un adapter per ogni classe da adattare.

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 (Adaptii).

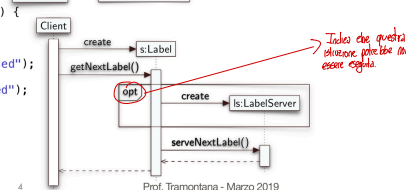
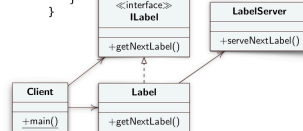


```
public interface ILabel { // Target
    public String getNextLabel();
}

// Adapter
public class Label implements ILabel {
    private LabelServer ls;
    private String p;
    public Label(String prefix) {
        p = prefix;
    }
    public String getNextLabel() {
        if (ls == null)
            ls = new LabelServer(p);
        return ls.getNextLabel();
    }
}

public class Client {
    public static void main(String args[]) {
        ILabel s = new Label("LAB");
        String l = s.getNextLabel();
        if (!l.equals("LAB"))
            System.out.println("Test 1:Passed");
        else
            System.out.println("Test1:Failed");
    }
}
```

```
public class LabelServer { // Adaptee
    private int labelNum = 1;
    private String labelPrefix;
    public LabelServer(String prefix) {
        labelPrefix = prefix;
    }
    public String getNextLabel() {
        return labelPrefix + labelNum++;
    }
}
```



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.

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**

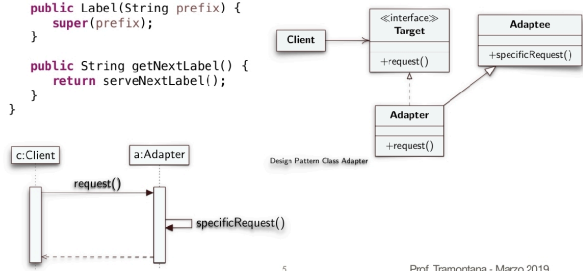
- Soluzione Class Adapter

- Adapter è sottoclasse di Adaptee

```
public class Label extends LabelServer implements ILabel { // Adapter
```

```
public Label(String prefix) {
    super(prefix);
}

public String getNextLabel() {
    return serveNextLabel();
}
}
```



Object Adapter: Ritarda l'inizializzazione fino a poco prima dell'uso del metodo per evitare che il client perda tempo

Adapter: Si crea subito la classe Adapter (si spende tempo nell'inizializzazione)

Differenze Object Adapter e Class Adapter

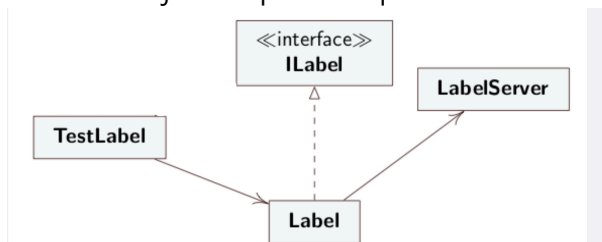
- Dipendono dai requisiti, ovviamente, e dalle scelte di progettazione
- **si crea un'unica istanza di Adapter che contiene tutto.** Se la superclasse richiede molta memoria allora si inizializza anche Adapter allo stesso momento e quindi si spende quel tempo di inizializzazione (*a prescindere*)
- In **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)
 - Adapter 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 Utile per far comunicare due classi incompatibili

Fornisce sia l'interfaccia di Target e sia l'interfaccia 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'interfaccia Target** e i **metodi dell'interfaccia Adaptee**
- **Client** si interfaccia con **Adapter** essendo più "ricca" di informazioni rispetto a **Target**

Conseguenze. #1 Client e Classe Adapter sono indipendenti. Adapter può Cambiare il COMPORTAMENTO di ADAPTEE.

• Conseguenze del design pattern Adapter

- Client e classe di libreria Adaptee rimangono indipendenti. L'Adapter può cambiare il comportamento dell'Adaptee
- Può aggiungere test di precondizioni e postcondizioni [Precondizioni: cosa si deve soddisfare prima di eseguire. Postcondizioni: cosa è verificato se tutto è andato bene]
- L'Object Adapter può implementare la tecnica di Lazy Initialization
- Il design pattern Adapter aggiunge un livello di indirettezza. Ogni invocazione del client ne scatena un'altra fatta dall'Adapter. Possibile rallentamento (trascurabile), e codice più difficile da comprendere

Prof. Tramontana - Marzo 2019

#2 Può Aggiungere test di **PRECONDIZIONI** = Devono essere effettuate durante la FASE di PROGETTAZIONE
I controlli li inseriamo nella classe Adapter.

POSTCONDIZIONI = Verifichiamo se la CONDIZIONE è ANDATA A BUON FINE

#3 Codice più difficile da comprendere perché Adapter poco DOCUMENTATO essendo che fa da tramite.
(Poco usato (classiche) dovuto che ad ogni azione di noi classe Adapter deve ADATTARSI)

- **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".