Design Pattern Strutturali

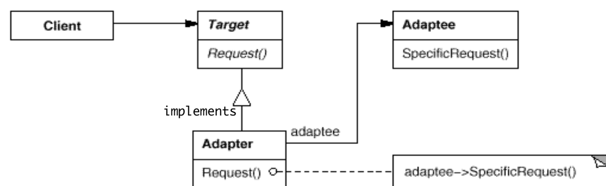
- Se agiscono sulle classi (ad es. il *Class Adapter*)
  - Usano l'ereditarietà per comporre interfacce o classi
- Se agiscono sugli oggetti
  - Descrivono modi per comporre oggetti
  - La composizione di oggetti può variare a run-time
- Permettono di diminuire le dipendenze tra classi o tra algoritmi

Adapter

- Intento
  - Converte l'interfaccia di una classe in un'altra interfaccia che i client si aspettano
  - Adapter permette ad alcune classi di interagire, eliminando il problema di interfacce incompatibili
- Motivazione
  - Certe volte una classe di una libreria non può essere usata poiché incompatibile con l'interfaccia richiesta dall'applicazione
    - Nome metodo, parametri, tipo parametri non corrispondenti
  - Non è possibile cambiare l'interfaccia della libreria
    - Poiché non si ha il sorgente (comunque non conviene cambiarla)
  - Non è possibile cambiare l'applicazione
    - Si può voler cambiare quale metodo invocare, senza renderlo noto al client

Adapter

- Soluzione (Object Adapter)
  - Creare una classe `Adapter` che converte, ovvero adatta, l'interfaccia che il client si aspetta (`Target`) all'interfaccia della classe di libreria
  - Il client usa l'`Adapter` come se fosse l'oggetto di libreria
  - L'`Adapter` possiede il riferimento all'oggetto di libreria (detto `Adaptee`) e sa come invocarlo



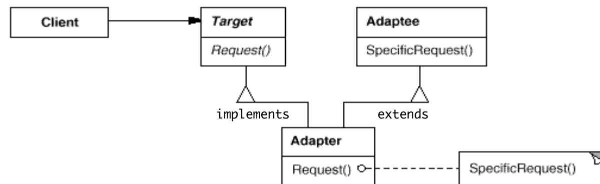
Object Adapter

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

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

```
// Adapter
public class Label implements ILabel {
  private LabelServer theService;
  public Label(String prefix) {
     theService = new LabelServer(prefix);
  }
  public String getNextLabel() {
     return theService.serveNextLabel();
  }
}

public class Client {
  static public void main(String args[]) {
    ILabel s = new Label("LAB");
    String l = s.getNextLabel();
    if (l.equals("LAB1"))
       System.out.println("Test 1: Passed");
    else System.out.println("Test 1: Failed");
  }
}
```

Class Adapter

- Altra soluzione (Class Adapter)
  - Uso dell'ereditarietà multipla




```
// Adapter
public class Label extends LabelServer implements ILabel {
  public Label(String prefix) {
    super(prefix);
  }
  public String getNextLabel() {
    return serveNextLabel();
  }
}
```


Adapter

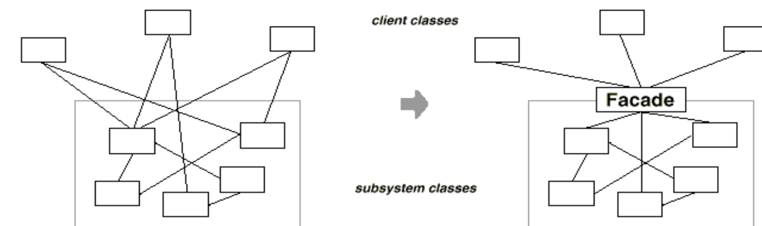
- Variante
  - Adapter a due vie
    - La classe Adapter include l'interfaccia di Adaptee
    - La versione Class Adapter fornisce l'Adapter a due vie
- Conseguenze
  - Client e classe di libreria rimangono indipendenti
  - L'Adapter può cambiare il comportamento dell'Adaptee
    - Aggiungendo codice, per es. test di precondizioni e postcondizioni
    - Permette di implementare la tecnica di Lazy Initialization
  - L'Adapter aggiunge un livello di indirettezza
    - Ogni invocazione del client ne scatena un'altra fatta dall'Adapter
    - Possibile overhead e codice più difficile da comprendere

Façade

- Intento
  - Fornire una interfaccia unificata ad un set di interfacce in un sottosistema (consistente di un gruppo di classi)
  - Definire una interfaccia di alto livello (semplificata) che rende il sottosistema più facile da usare
- Problema
  - Spesso si hanno tante classi che svolgono funzioni correlate e l'insieme delle interfacce può essere complessa
  - Può essere difficile capire qual è l'interfaccia essenziale ai client per l'insieme di classi
  - Si vogliono ridurre le comunicazioni e le dipendenze dirette tra i client ed il sottosistema

Façade

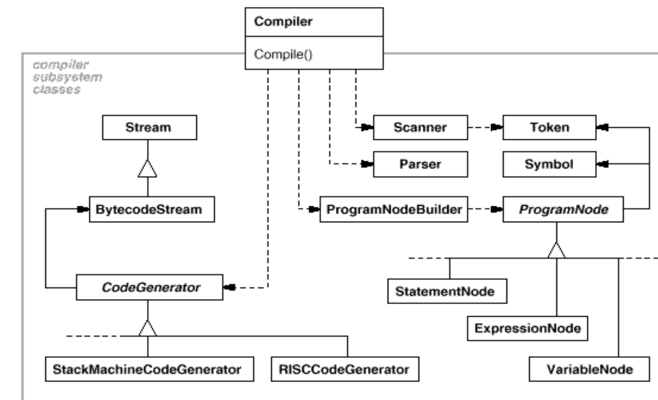
- Soluzione
  - Un modo per ridurre la complessità è introdurre un oggetto Facade che fornisce un'unica interfaccia semplificata e nasconde gli oggetti del sottosistema
  - Il client interagisce solo con l'oggetto Facade
  - Il Facade invoca i metodi degli oggetti che nasconde



Façade

- Conseguenze
  - Nasconde ai client l'implementazione del sottosistema
  - Promuove accoppiamento debole tra sottosistema e client
  - Riduce dipendenze di compilazione in sistemi grandi
  - Non previene l'uso di client più complessi, quando occorre, che accedono oggetti del sottosistema
- Implementazione
  - Per rendere gli oggetti del sottosistema non accessibili al client le corrispondenti classi possono essere annidate dentro la classe Facade

Façade



Façade

```
public class English {
  ...
  private String text = " ";
  private String[] EngDict = {"Alright",
            "Hello", "Understood", "Yes"};
  public int getPos(String s) { ... }
  public boolean test(String s) { ... }
  public void add(String s) {
    text = text + " " + s; }
  public String getText() {return text;}
  public void printText() {
    System.out.println(text); }
}

public class TestFacade {
  static public void main(String args[]) {
    Translator t = new Translator();
    t.addEnglish("Hello");
    t.multiPrinting();
  }
}
```

```
// Facade
public class Translator {
  private English eng = English.getInstance();
  private Italian it = Italian.getInstance();
  public void addEnglish(String s) {
    String s2 = null;
    if (eng.test(s)) {
      eng.add(s);
      s2 = it.intoItalian(s);
      it.add(s2);
    }
  }
  public void multiPrinting() {
    System.out.print("Italiano:  ");
    it.printText();
    System.out.print("English:   ");
    eng.printText();
  }
}
```