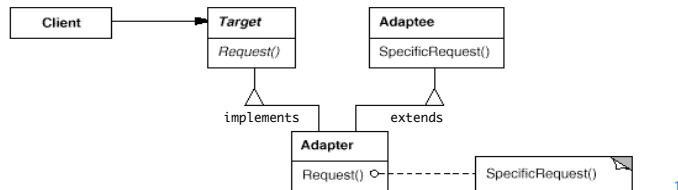


Design Pattern Adapter

- Intento
 - Converte l'interfaccia di una classe servizio in un'altra interfaccia che le classi client si aspettano. Permette alle classi di interagire, eliminando l'incompatibilità dell'interfaccia
- Soluzione (class Adapter)
 - Una classe Adapter converte, ovvero adatta, l'interfaccia che il client si aspetta (Target) all'interfaccia della classe che fornisce il servizio
 - Il client usa l'Adapter come se fosse l'oggetto servizio (detto Adaptee)
 - L'Adapter è derivato da Adaptee ed invoca il metodo corrispondente



Codice prima dell'Adapter

```
class LabelServer {
    private int nextLab = 1;
    private String labelPref;
    LabelServer(String prefix) {
        labelPref = prefix;
    }
    String serveNextLabel() {
        return labelPref+nextLab++;
    }
}
```

```
class Client {
    static void main(String args[]) {
        LabelServer s = new LabelServer("LA");

        String l = s.getNextLabel(); // incompatibilita'
    }
}
```

Ing. E. Tramontana - Design Pattern - 22-Nov-06 2

Codice con class Adapter

```
class LabelServer { // Adaptee
    private int nextLab = 1;
    private String labelPref;
    LabelServer(String prefix) {
        labelPref = prefix;
    }
    String serveNextLabel() {
        return labelPref+nextLab++;
    }
}
```

```
class Client {
    static void main(String args[]) {
        ILabel s = new Label("LA");
        String l = s.getNextLabel();
    }
}
```

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

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

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

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

Ing. E. Tramontana - Design Pattern - 22-Nov-06 3

Valutazioni

- Supponiamo di aver sviluppato vari client conformi a Target
- Per adeguare i client all'uso di Adapter dovremo
 - Cambiare ciascun client che chiama il servizio per far usare l'Adapter anziché il servizio
 - Implementare una classe Adapter ed una interfaccia Target per ogni servizio da adattare
 - Poiché il servizio può anche essere chiamato direttamente
 - Non possiamo sapere se tutte le chiamate passano da Adapter
 - Può essere un problema se l'Adapter aggiunge codice utile

Ing. E. Tramontana - Design Pattern - 22-Nov-06 4

AspectJ Adapter

- Si vorrebbe usare un l'Adapter senza dover cambiare i client
- Un aspetto può essere implementato con tale finalità

```
aspect LabelAdapter { // AspectJ Adapter

    String LabelServer.getNextLabel() {
        return serveNextLabel();
    }
}
```

```
class Client {
    static void main(String args[]) {
        LabelServer s = new LabelServer("LA");

        String l = s.getNextLabel();
    }
}
```

Ing. E. Tramontana - Design Pattern - 22-Nov-06 5

AspectJ Adapter vers. 2

- Per avere un warning nel caso in cui il metodo del servizio venisse invocato direttamente

```
public aspect LabelAdapter { // AspectJ Adapter

    String LabelServer.getNextLabel() {
        return serveNextLabel();
    }

    declare warning :
        call(* LabelServer.serveNextLabel(..)) &&
        !within(LabelAdapter2) :
            "\nLa chiamata a serveNextLabel() dovrebbe essere
            sostituita con la chiamata a getNextLabel()";
}
```

Ing. E. Tramontana - Design Pattern - 22-Nov-06 6

Valutazioni

- La soluzione AspectJ Adapter
 - Non obbliga a modifiche del client
 - È localizzata solo nell'aspetto
 - Eventuali sottoclassi di Adapter non necessitano di un loro Adapter
 - Il metodo definito dall'aspetto è utilizzabile dalle sottoclassi
 - Eventuali metodi specifici possono essere aggiunti dallo stesso aspetto
 - Si può impedire l'uso di metodi preesistenti generando un errore o un warning a compile time

Ing. E. Tramontana - Design Pattern - 22-Nov-06 7

Static Crosscutting

- Gli aspetti possono modificare la struttura di classi, interfacce o altri aspetti
- Ci sono 4 tipi di static crosscutting
 - Inserimento attributi e metodi
 - Modifica gerarchia di classi
 - Dichiarazione di errori e warning a compile-time
 - Exception softening

Ing. E. Tramontana - Design Pattern - 22-Nov-06 8

Inserimento attributi e metodi

- Un aspetto può aver bisogno di introdurre attributi su una classe per memorizzare valori calcolati

```
public aspect Introduce {  
    // inserisce l'attributo minimumBalance di tipo float su Account  
    private float Account.minimumBalance;  
  
    public float Account.getAvailableBalance() {  
        return getBalance() - minimumBalance;  
    }  
}
```

Modifica gerarchia di classi

- È possibile dichiarare una superclasse o una interfaccia per una classe

```
declare parents : [childType] implements [interface]  
declare parents : [childType] extends [class]
```

Dichiarazione errori e warning

- È possibile far segnalare al compilatore un errore o un warning quando un join point soddisfa un pointcut specificato

```
declare error : <pointcut> : <message>;  
declare warning : <pointcut> : <message>;
```

- I pointcut usati non possono selezionare un contesto dinamico
 - Quindi `this`, `target`, `args`, `cflow` e `cflowbelow`, `if` non possono essere usati