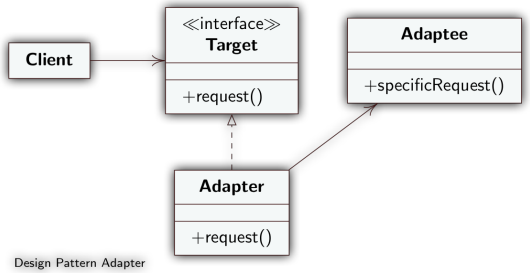
Adapter

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

Pattern "*strutturale*"

Class Adapter

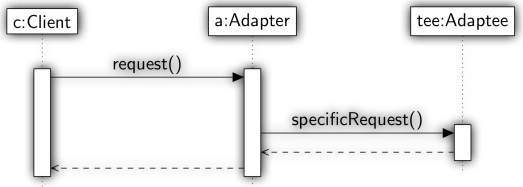
- Soluzione:
  - **Target** è l'interfaccia che il chiamante si aspetta
  - **Client** usa oggetti che sono conformi all'interfaccia Target
  - **Adaptee** è l'oggetto di libreria che necessita l'adattamento
  - **Adapter** converte, ovvero adatta, la chiamata che fa una classe client all'interfaccia della classe di libreria.



Design Pattern Adapter

Class Adapter

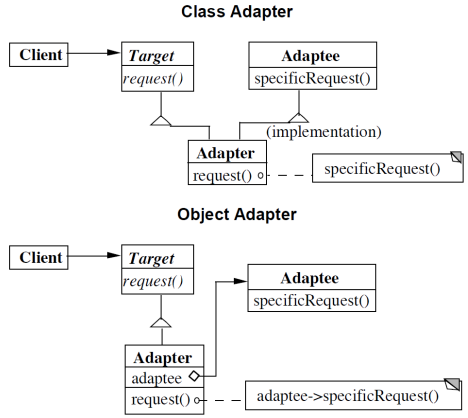
- Sequence diagram



- Non è possibile cambiare l'interfaccia dell'Adaptee, poiché non si ha il sorgente (comunque non conviene cambiarla)
- Non è possibile cambiare l'applicazione (il Client), e si può voler cambiare il metodo chiamato, senza renderlo noto al client

Object vs Class Adapter

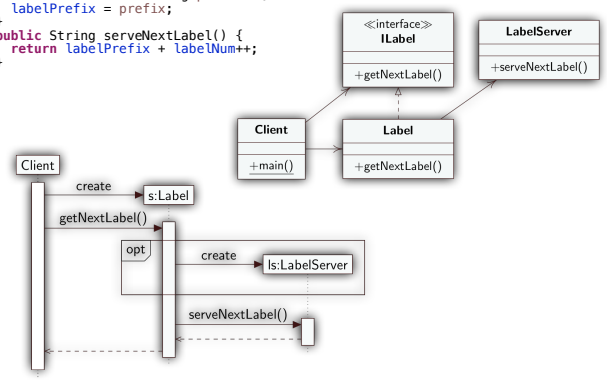
- Quando vuoi riusare molte classi esistenti è poco pratico adattare le interfacce sottoclassandole tutte
- Un ***object adapter*** può adeguare l'interfaccia di varie classi a quella del client senza sottoclassarle



Object Adapter example

```
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() {
    // lazy initialisation
    if (null == ls)
              ls = new LabelServer(p);
    return ls.serveNextLabel();
  }
}

public class Client {
  public static 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("Test1:Failed");
  }
}
```

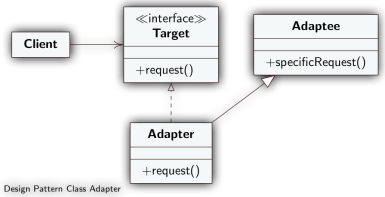
```
public class LabelServer { // Adaptee
  private int labelNum = 1;
  private String labelPrefix;
  public LabelServer(String prefix) {
    labelPrefix = prefix;
  }
  public String serveNextLabel() {
    return labelPrefix + labelNum++;
  }
}
```

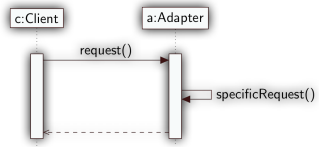


Soluzione Class Adapter

- La classe con il ruolo Adapter è sottoclasse di Adaptee

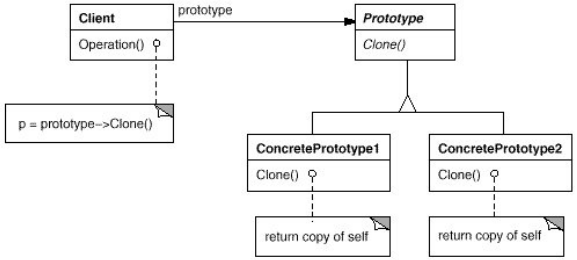
```
public class Label extends LabelServer
              implements ILabel { // Adapter

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

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





Prototype

- Al Client serve un oggetto di un tipo astratto T (o con un set di funzionalità F)
- Il Client stesso ne deve poter controllare l'istanziazione (come nel factory)
- Restando indipendente dall'implementazione dell'oggetto usato (polimorfismo)



- Novità: il metodo creazionale è direttamente dentro la classe che creo
  - Non è un factory method del Client, né di un Creator esterno

Intento

Creazione di istanze, non da una classe ma da una sua istanza base (il prototipo)

invece di : AbstractClass obj = new ConcreteSubClass();

avrò : AbstractClass obj = prototype->clone();

Nuove Istanze sono clonate dal client a partire da una istanza prototipale pre-esistente, di cui ho un riferimento

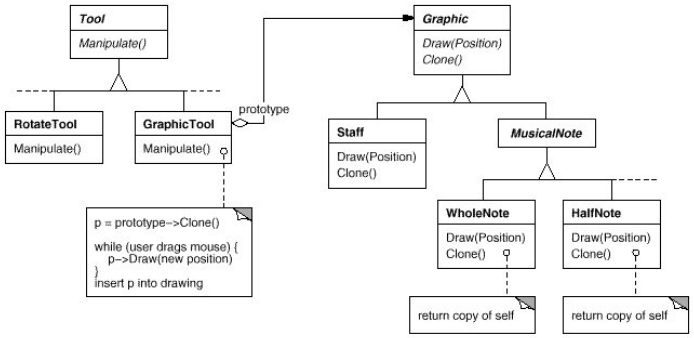
La clonazione è una operazione di duplicazione di un oggetto

La creazione dei prototipi è una operazione una-tantum, fatta altrove

Applicabilità

- Quando un sistema dovrebbe essere indipendente dal prodotto concreto che istanzia, e in più…
  - la classe concreta da usare **è nota solo a run-time:**
    - impossibile scrivere a priori metodi factory per incapsularne la creazione
  - oppure, la classe **ha solo in pochi stati possibili**, quindi ho poche versioni di istanze tutte uguali
    - conviene avere un prototipo da clonare per ogni versione dello stato invece di classi da istanziare e poi inizializzare.
  - oppure, se la creazione di nuove istanze di classe **è costosa**
    - es.: il costruttore fa varie query a DB. Copiare una istanza già pronta è più performante

Classi con poche istanze distinte



WholeNote, HalfNote, etc… cambiano solo nello stato (durata) e nel glifo

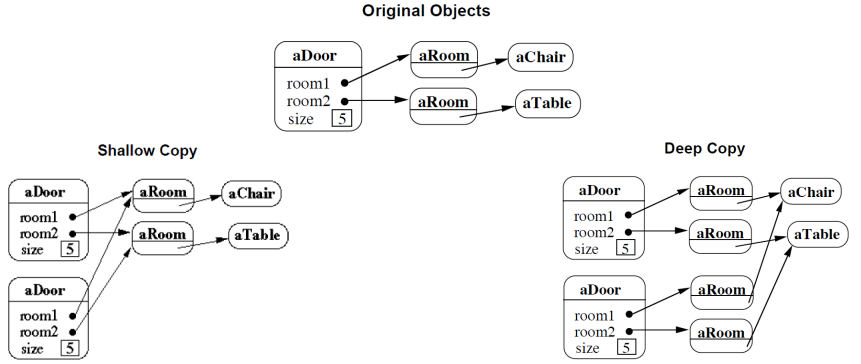
potrei avere la sola classe MusicalNote e alcune sue istanze configurate diversamente (prototipi) da clonare a volontà

Si riduce il numero di classi nel sistema

Prototipi in Java

- public interface Cloneable (dal JDK 1.0)
  - È vuota
  - Se una classe implementa Cloneable, invocare il metodo Object.clone() :
    - copia valore per valore tutte le variabili di istanza
    - altrimenti, exception CloneNotSupportedException
  - metodo posso fare override di Clone() per una deep copy

Shallow vs Deep copy



Deep copy example

```
class Door implements Cloneable {
      Room room1;
      Room room2;

      public void Initialize( Room a, Room b){
            room1 = a;    room2 = b;
      }

      public Object clone() throws CloneNotSupportedException {
            // method for deep copy
            Door cloned = super.clone();
            cloned.Initialize(room1.clone(), room2.clone());
            return cloned;

            // no need to implement this method for shallow copy
            //return super.clone();
      }
}
```


Conseguenze

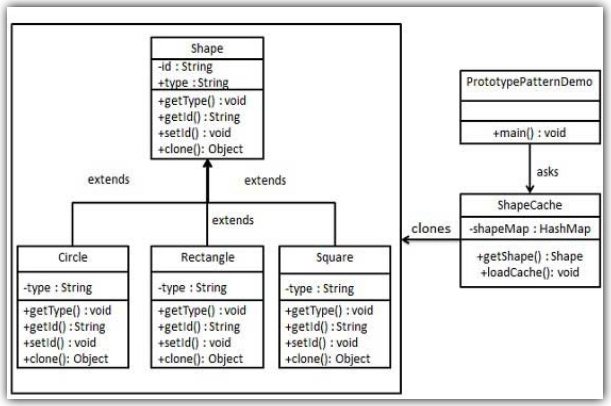
Pro:
- posso sostituire i prototipi facilmente (l'oggetto o la struttura cui fa capo) a run-time senza modifiche nel client
- tengono celata la struttura (eventualmente) complessa al loro interno
- non servono classi derivate, rispetto al factory method (classico)

Contro:
- devo implementare il metodo Clone()
- devo inizializzarli, se necessario
- a volte comodo un prototype manager

Prototype Manager

- In alcuni casi il numero di prototipi non è noto o fisso
- Uso un Catalogo (o registro) di prototipi clonabili
- E' una memoria associativa dove posso registrare (o cancellare) nuovi prototipi sotto un etichetta o un codice simbolico
- Restituisce il riferimento ad un prototipo associato ad una data chiave
- prototypeManager.get("ProtoXY123").clone()
- La chiave potrebbe essere il nome della classe cui appartiene

Esempio Prototype Manager



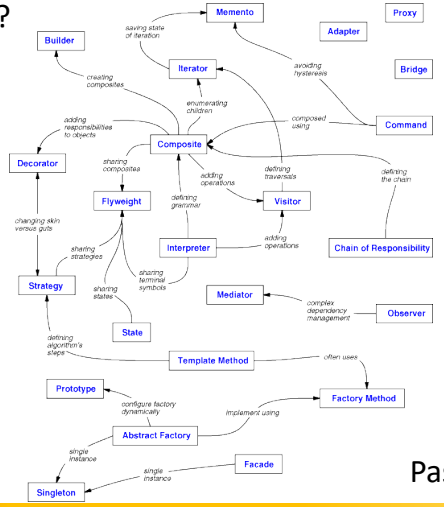
VS CODE SHAPE PROTOTYPE

Abstract Factory impl. con Prototipi

```
class WidgetFactory{ //non più abstract
        Window windowFactory;
        Menu menuFactory;
        Button buttonFactory;

        public WidgetFactory( Window windowPrototype, Menu menuPrototype, Button buttonPrototype){
                this.windowFactory= windowPrototype;
                this.menuFactory= menuPrototype;
                this.buttonFactory= buttonPrototype;
        }
        public Window createWindow() { return windowFactory.clone() }
        public Menu createMenu(){ return MenuFactory.clone() }
        public Button createButton(){ return ButtonFactory.clone() }
    etc.
}
```

Non serve sottoclassare WidgetFactory ma inizializzarla con le giuste istanze di prototipi, da clonare invece che istanziare. Non è solo una dependency injection

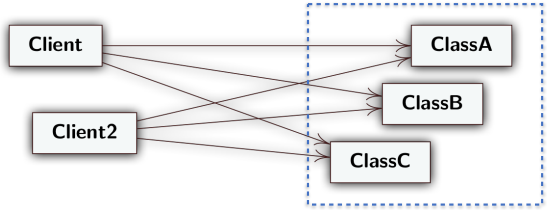
A CHE PUNTO SIAMO?



Passiamo al Facade

Design pattern Facade

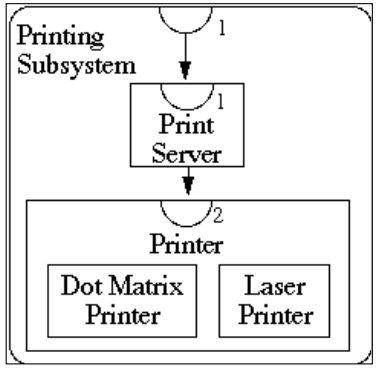
- **Intento**: Fornire un'interfaccia unificata al posto di un insieme di interfacce in un sottosistema (consistente di un insieme di classi). Definire un'interfaccia di alto livello (semplificata) che rende il sottosistema più facile da usare



FACADE

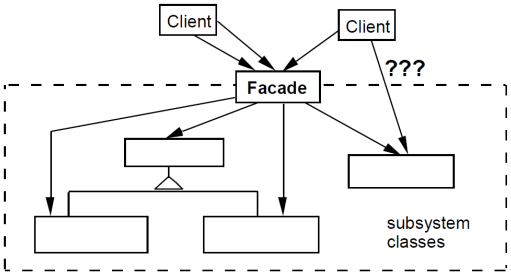
Si applica ad un Sottosistema:

Gruppi di classi, ed eventualmente altri sottosistemi, che collaborano tra loro per supportare la realizzazione di un set di compiti specifici



Design pattern Facade

- Soluzione
  - **Facade** fornisce un'unica interfaccia semplificata ai client e nasconde gli oggetti del sottosistema, questo riduce la complessità dell'interfaccia e quindi delle chiamate. **Facade** invoca i metodi degli oggetti che nasconde
  - **Client** interagisce solo con l'oggetto Facade



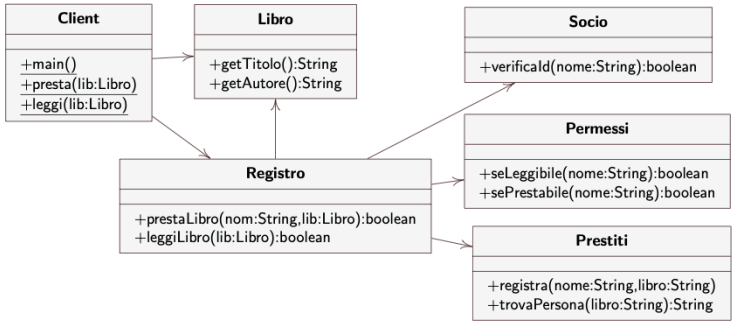
Conseguenze

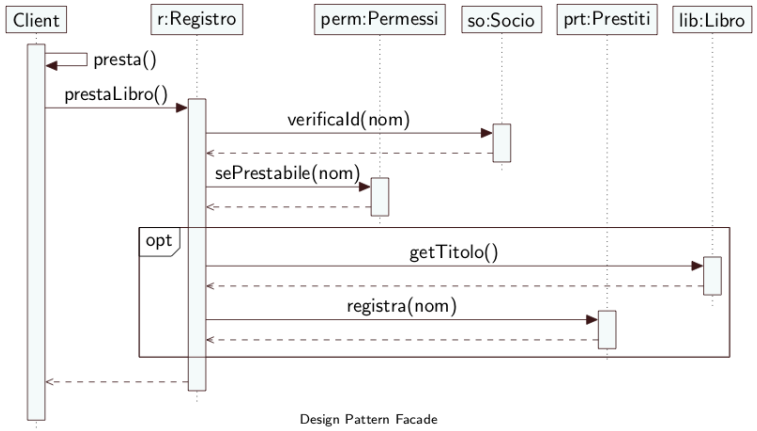
- Nasconde ai client l'implementazione del sottosistema
- Riduce l'accoppiamento tra sottosistemi
  - Riduce le dipendenze di compilazione in sistemi grandi.
  - Se si cambia una classe del sottosistema, si può ricompilare la parte di sottosistema fino al facade, quindi non i vari client
- Se era direttamente il client **a istanziare gli oggetti, ora ci deve pensare il facade**
- **Non previene l'accesso diretto, quando occorre, agli oggetti del sottosistema**

Implementazione

- **Non deve aggiungere funzionalità non presenti già nel sottosistema**
  - può rielaborare e comporre funzionalità esistenti, solo per semplificare l'uso e ridurre il numero di metodi dell'interfaccia rispetto al totale delle singole classi
- Per rendere gli oggetti del sottosistema non accessibili all'esterno:
  - le classi possono essere annidate dentro il Façade (nested/inner class)
  - posso usare modificatori di accesso che limitino al package
  - In Java: Default class Access Modifier per le classi del sottosistema
  - Solo per Façade *public* class e/o membri

Esempio: Libreria





Design Pattern Facade