



Design Pattern

Basic Tutorial

maggio 2005

Agenda



- Programmazione OO Avanzata:
 - Reflection
 - Introduzione ai Design Pattern
- I pattern principali
- I pattern dell'API di Java
- Bibliografia
- Conclusioni

Programmazione OO Avanzata



- La programmazione orientata agli oggetti è stata una formidabile rivoluzione nello sviluppo del software
- Incapsulamento, ereditarietà, polimorfismo sono concetti che conferiscono incredibile espressività ai linguaggi di programmazione e permettono di modellare efficacemente sistemi complessi e ambienti del mondo reale
- L'OOP fornisce gli elementi “atomici”, ma occorre andare oltre...

Introduzione ai Design Pattern



CRS4 - <http://www.crs4.it>

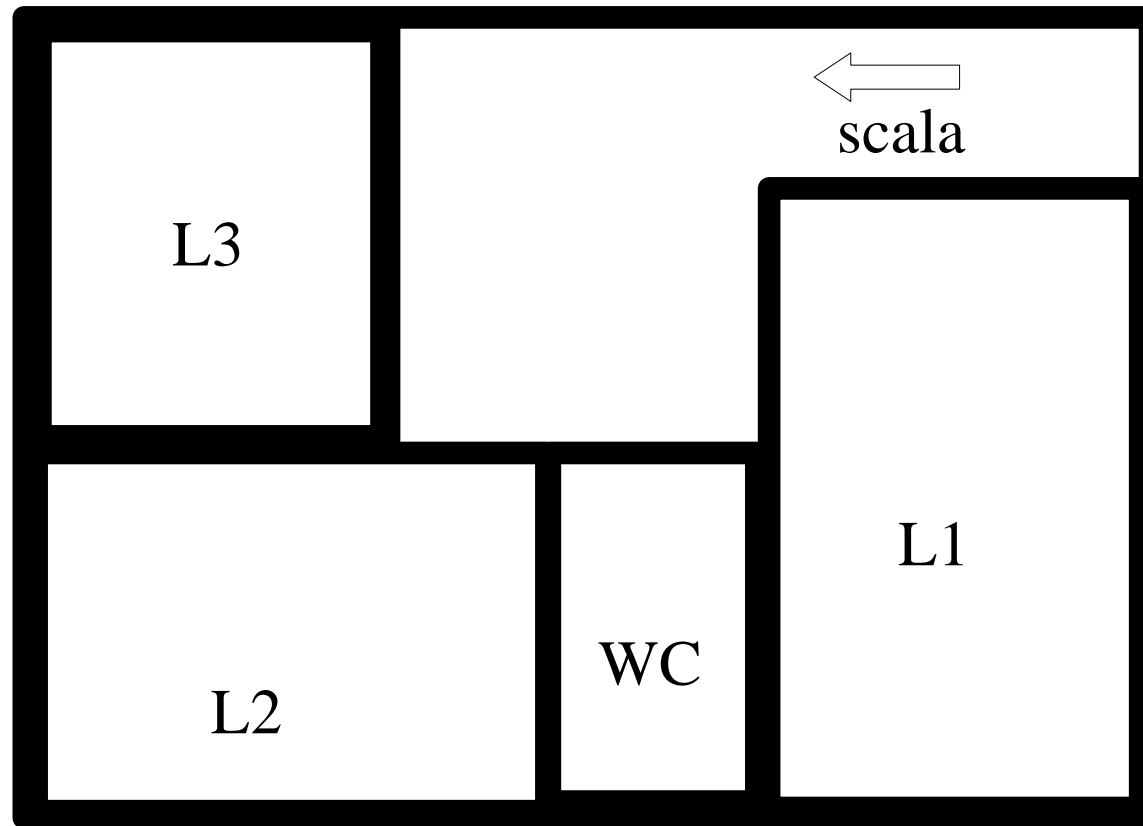


Vogliamo progettare il primo piano della nostra villetta disponendo 3 camere da letto e un bagno. Nessuna finestra deve aprirsi a nord. Nessuna camera da letto deve essere meno ampia di 9mq

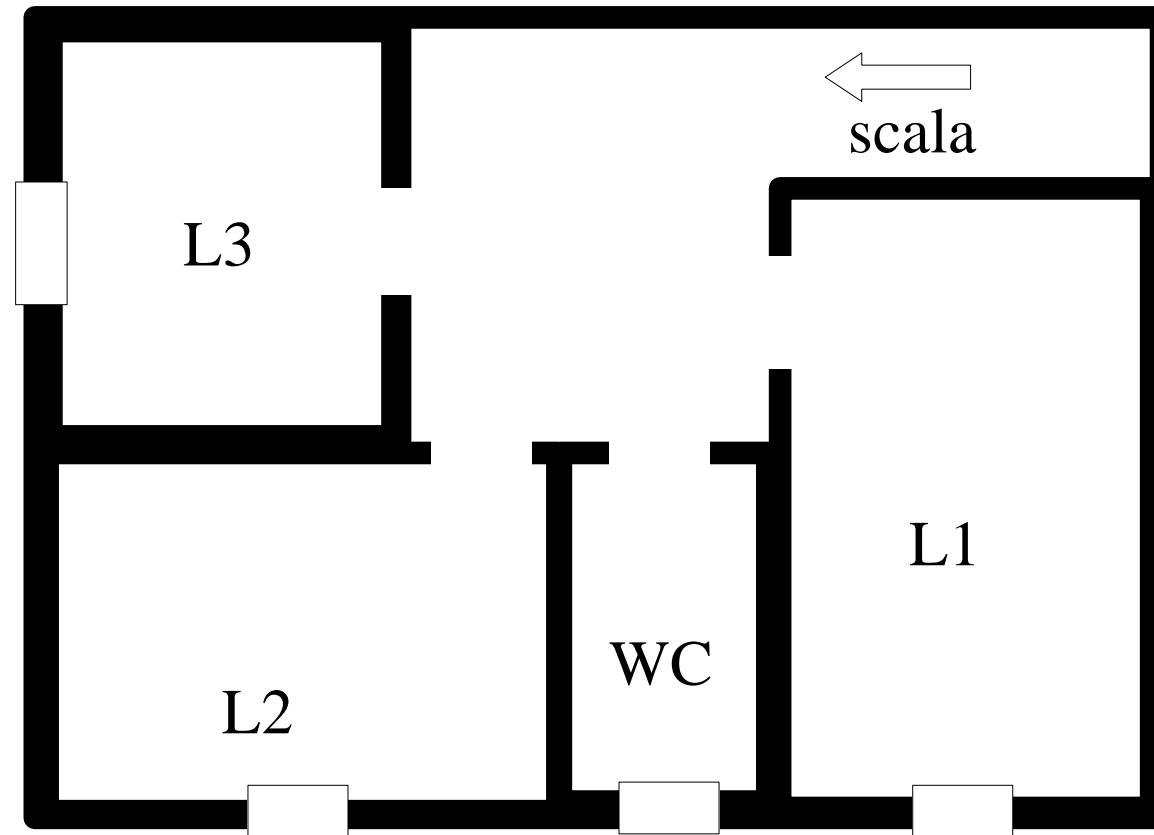
Introduzione ai Design Pattern



CRS4 - <http://www.crs4.it>



Introduzione ai Design Pattern



Introduzione ai Design Pattern



- Dato un problema, troviamo una soluzione
- Data una classe di problemi simili, troviamo una classe di soluzioni simili
- Una classe di soluzioni rappresenta una architettura generica detta “pattern”

Design Pattern



- Durante la modellazione dei sistemi software emergono delle **strutture ricorrenti**, degli schemi noti o comunque riconducibili a uno di essi.
- Nello sviluppo software è possibile:
 - Individuare la corrispondenza con un pattern e approcciarne una applicazione
 - Procedere con una prima progettazione non legata ad alcun pattern e, nel caso si evidenzino degli schemi noti, ricondurre parti del modello al design pattern

Tipi di Design Pattern



- Creational pattern
 - Descrivono le modalità per l'istanziamento di oggetti e famiglie di classi
- Structural pattern
 - Descrivono le modalità di organizzazione di classi e oggetti
- Behavioral pattern
 - Descrivono le modalità di comportamento a runtime degli oggetti

Creational pattern



- Abstract Factory
 - Crea istanze di diverse famiglie di classi
- Builder
 - Separa la costruzione e rappresentazione
- Factory Method
 - Crea istanze di diverse classi derivate
- Prototype
 - Fornisce una istanza generica di supporto
- Singleton
 - Definisce una classe con una sola istanza

Structural pattern



- Adapter
 - Connette classi con interfacce differenti
- Bridge
 - Separa l'interfaccia dalla implementazione
- Composite
 - Permette di creare una struttura ad albero di oggetti semplici e composti
- Decorator
 - Permette di aggiungere a runtime funzionalità ad oggetti esistenti

Structural pattern



- Façade
 - Permette di avere un'unica interfaccia verso un insieme di diversi sottosistemi separati
- Flyweight
 - Offre un sistema leggero di “object-sharing” dinamico
- Proxy
 - Permette di utilizzare un oggetto al posto di un altro

Behavioral pattern



- Chain of Responsibility
 - Permette di propagare le richieste in una catena di oggetti
- Command
 - Permette di incapsulare una richiesta in un oggetto che ne rappresenta il “comando”
- Interpreter
 - Permette di includere elementi di un linguaggio in una applicazione

Behavioral pattern



- Iterator
 - Permettere di accedere in maniera sequenziale gli elementi di una generica struttura dati
- Mediator
 - Semplifica la comunicazioni fra classi
- Memento
 - Permette di memorizzare e ripristinare lo stato
- Observer
 - Permette una gestione efficace della propagazione della variazione di un oggetto

Behavioral pattern



- State
 - Altera il comportamento di un oggetto in base al suo stato interno
- Strategy
 - Incapsula un algoritmo in una classe
- Template Method
 - Formalizza i passi di un algoritmo, delegando alle sottoclassi l'effettiva implementazione
- Visitor
 - Permette di definire funzioni dinamicamente

Uso dei DP della API Java



- I DP sono utilizzati diffusamente all'interno dell'API Java e costituiscono un valido esempio di uso sapiente di questa metodologia
- (Ri)conoscere l'adozione di un pattern nell'API standard aiuta a comprendere meglio l'architettura della libreria e scrivere codice *allineato* con il design degli sviluppatori Sun e JCP

Singleton



- Obiettivo: assicurare che, in ogni istante, esista **una sola istanza** di una certa classe all'interno dell'applicazione.
- Il Singleton è utile per identificare entità “uniche” all'interno del modello o per fornisce un accesso univoco a risorse di sistema (si pensi al sottosistema grafico o alla scheda di rete)
- E' compito della classe gestire l'accesso concorrente alle funzionalita' esportate

Singleton in Java



```
public class NetworkConnection {
    private static NetworkConnection instance;

    private NetworkConnection() {...}

    public static NetworkConnection getInstance() {
        if (instance == null) {
            instance = new NetworkConnection();
        }
        return instance;
    }
}
```

Singleton in Java



```
public class NetworkConnection {
    private static NetworkConnection instance;

    private NetworkConnection() {...}

    public static NetworkConnection getInstance() {
        if (instance == null) {
            instance = new NetworkConnection();
        }
        return instance;
    }
}
```

Il costruttore e' privato:
non e' possibile ottenere
istanze della classe all'interno
del codice di altre classi.

Singleton in Java



```
public class NetworkConnection {  
    private static NetworkConnection instance;  
  
    private NetworkConnection() {..  
  
    public static NetworkConnection  
        if (instance == null) {  
            instance = new NetworkConnection();  
        }  
        return instance;  
    }  
}
```

Un campo privato della classe conserva un riferimento all'unica istanza di NetworkConnection

Singleton in Java



```
public class NetworkConnection {  
    private static NetworkConnection  
  
    private NetworkConnection() {...
```

getInstance() restituisce la reference all'unica istanza ed eventualmente la crea, se non disponibile (ad esempio, alla prima invocazione)

```
public static NetworkConnection getInstance() {  
    if (instance == null) {  
        instance = new NetworkConnection();  
    }  
    return instance;  
}  
}
```

Il costruttore privato e' accessibile all'interno della classe!

Singleton: esercizio



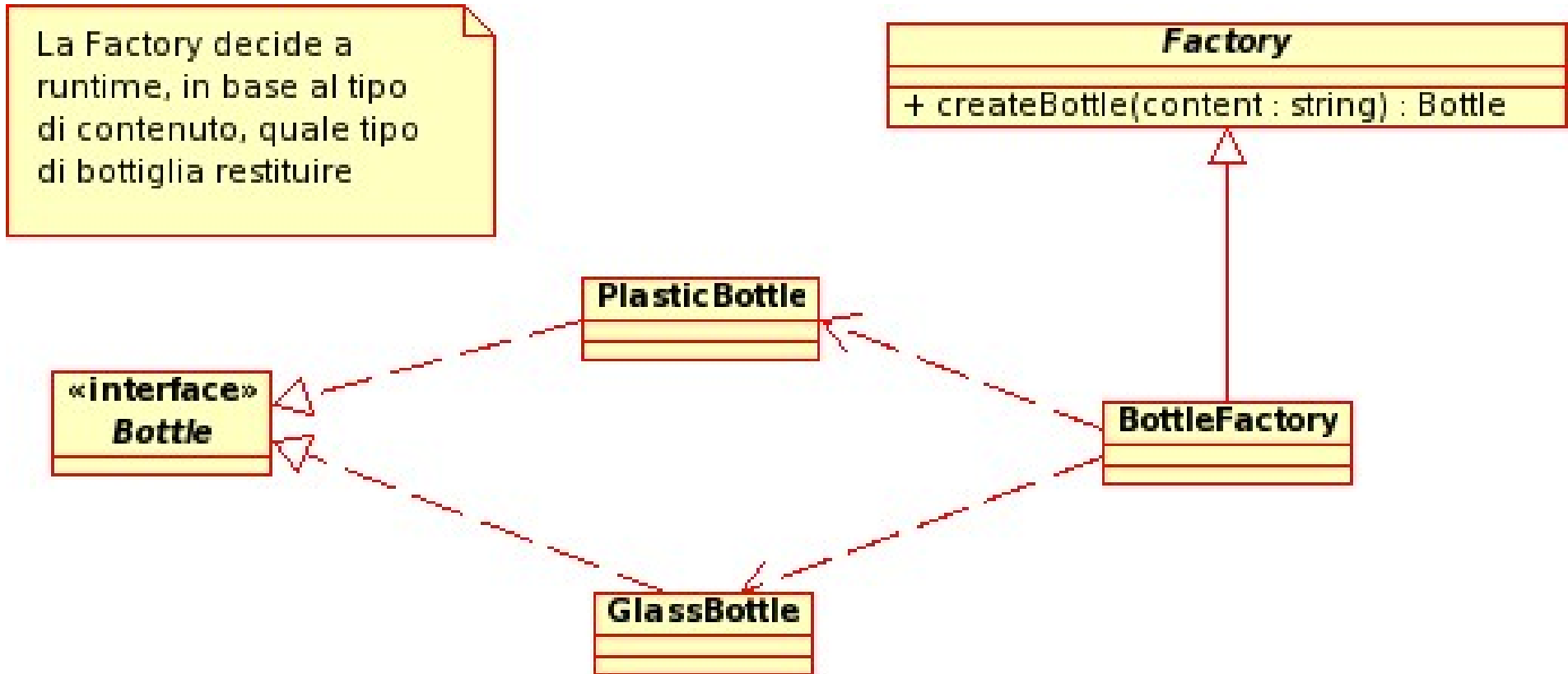
- Si vuole implementare il Multiton:
 - è una variante del Singleton, in cui si vuole limitare il numero totale di istanze di una classe ($1 < n_istanze < N_MAX$)
 - è un pattern utile per amministrare un numero finito di risorse fisiche o gestire il *load balancing* tra diversi elementi di codice attivo

Factory Method



- Obiettivo: poter decidere a runtime quale classe istanziare a partire da una gerarchia
- E' un pattern utilizzato per modellare sistemi reali o per aumentare la flessibilità dell'architettura:
 - Il tipo di oggetto istanziato non deve essere esplicitato nel codice del client (niente new MyClass(...))
 - Le regole di funzionamento della Factory possono essere variate dinamicamente (Abstract Factory)

Factory Method



Factory Method in Java



```
public interface Bottle {...}
```

```
public class GlassBottle implements  
    Bottle {...}
```

```
public class PlasticBottle implements  
    Bottle {...}
```

Factory Method in Java



```
public class BottleFactory {  
  
    public static Bottle getBottle(String content) {  
        if (content.equals("wine")) {  
            return new GlassBottle();  
        }  
        return new PlasticBottle();  
    }  
}
```

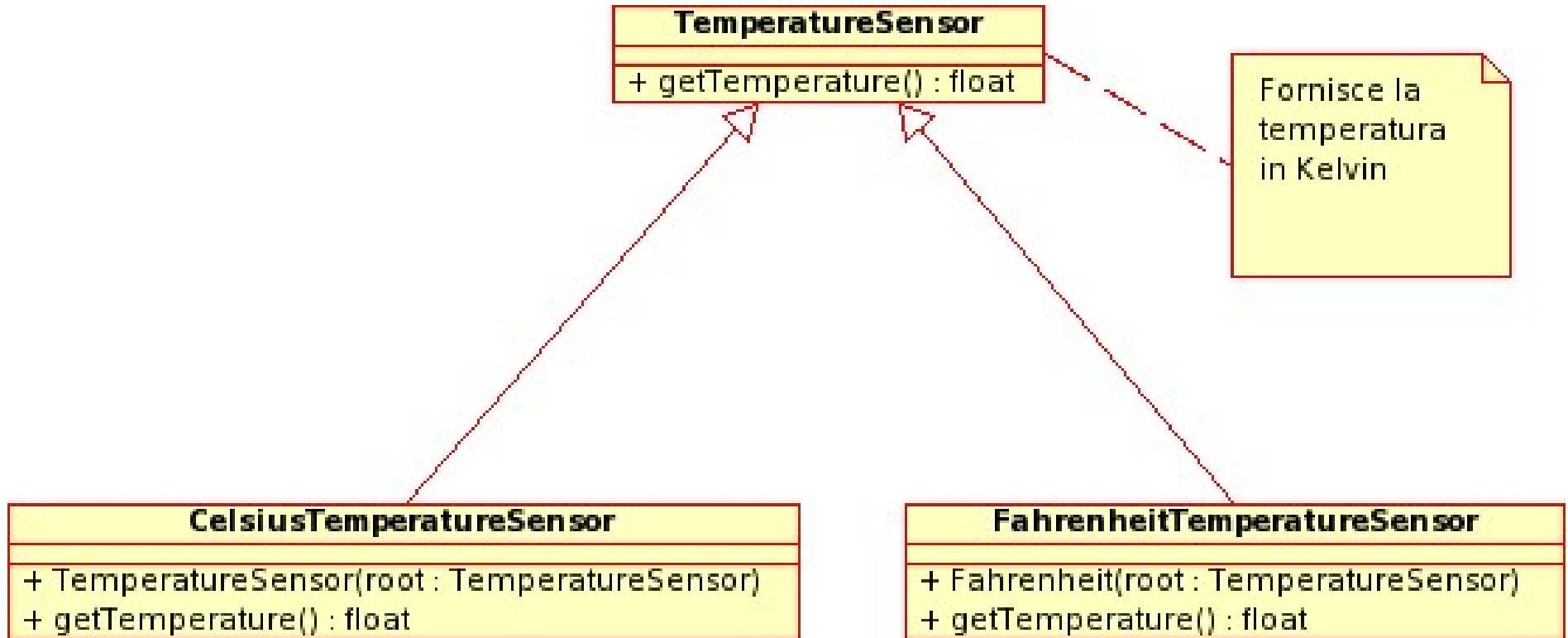
Se non è vino, la Factory restituisce sempre e solo bottiglie di plastica! :-)

Decorator



- **Obiettivo:** estendere/modificare a runtime le funzionalità esportate da un oggetto senza modificarne l'interfaccia pubblica
- Noto anche come “wrapper”, si utilizza quando il subclassing non e' possibile o occorre una modifica a runtime del funzionamento di un componente (si pensi ad oggetti già istanziati da un framework)
- Il Decorator si presenta con la medesima interfaccia dell'oggetto originale ma ha implementazione differente

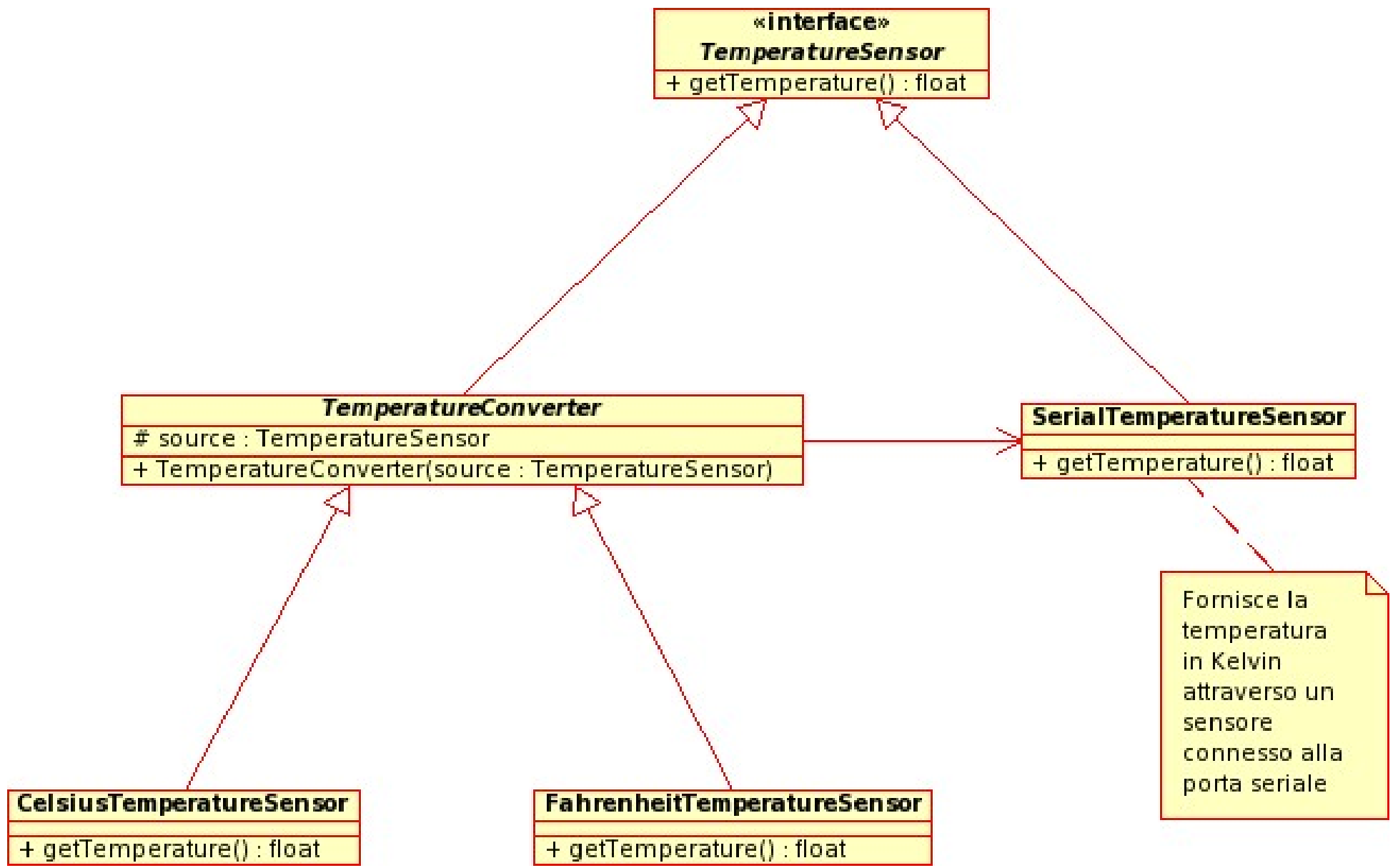
Decorator



Decorator



CRS4 - <http://www.crs4.it>



Decorator in Java



```
public interface TemperatureSensor {
    public float getTemperature();
}

public class SerialTemperatureSensor {
    public float getTemperature() {
        float temperature;
        // legge la temperatura dal sensore seriale...

        return temperature;
    }
}
```

Decorator in Java



```
public abstract class TemperatureConverter implements
    TemperatureSensor {
    protected TemperatureSensor source;

    public TemperatureConverter(TemperatureSensor sensor) {
        this.source = sensor;
    }
}
```

L'oggetto originale viene utilizzato come “sorgente di dati”, che saranno poi elaborati dalle implementazioni concrete del Decorator

Decorator in Java



```
public class CelsiusTemperatureSensor extends
    TemperatureConverter {

    public CelsiusTemperatureSensor(TemperatureSensor sensor)
    {
        super(sensor);
    }

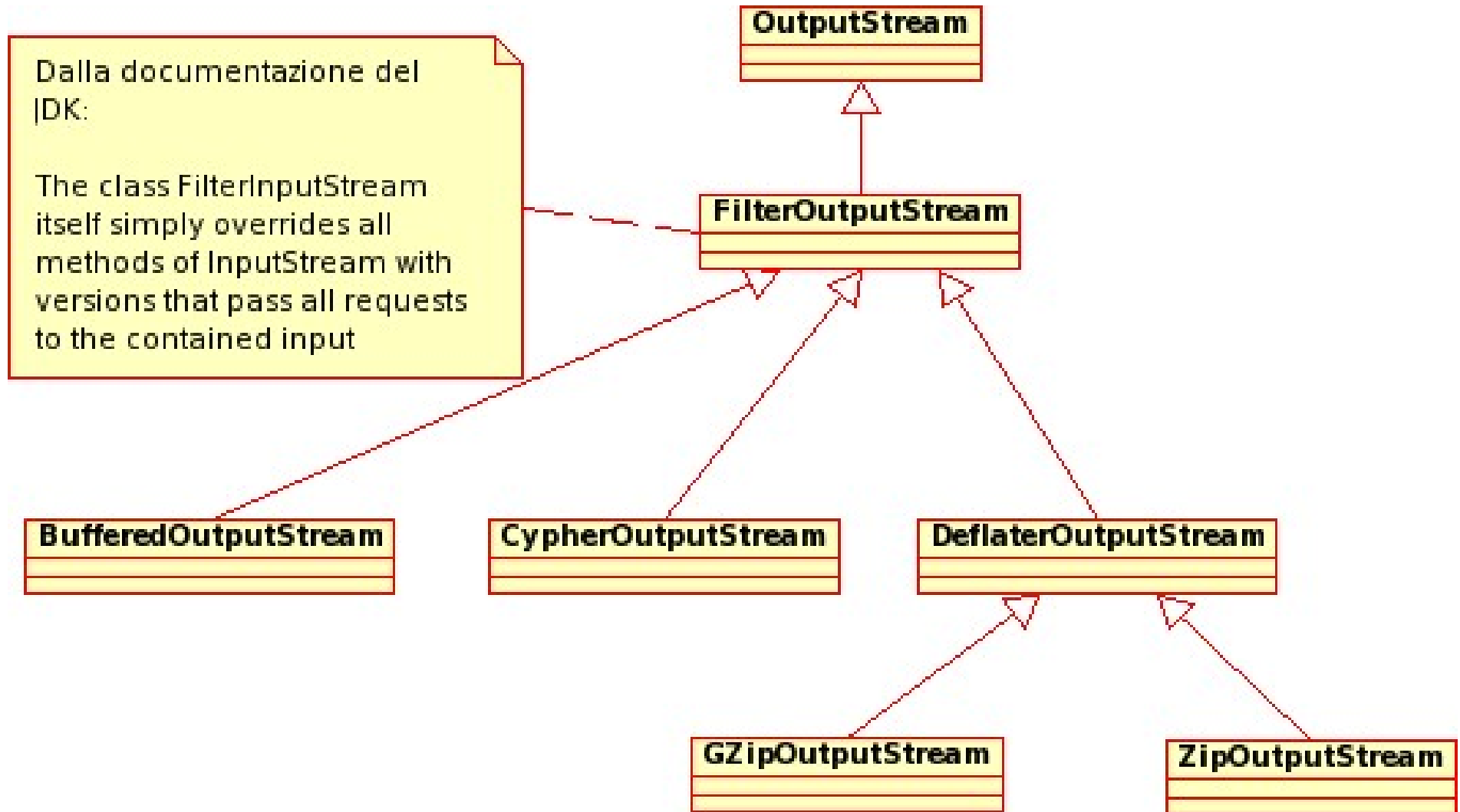
    public float getTemperature() {
        return (source.getTemperature() - 273.15);
    }
}
```

Decorator in J2SE



- Il DP Decorator e' presente in alcune classi fondamentali per la gestione delle funzionalita' di I/O:
 - InputStream e OutputStream sono le classi basi per la gestione dei flussi generici di ingresso ed uscita
 - Opportune sottoclassi di FilterInput/OutputStream forniscono, in modo trasparente, funzionalita' di buffering, compressione e cifratura

Decorator in J2SE



Adapter



- **Obiettivo:** rendere interoperabili due oggetti aventi interfacce incompatibili
- E' un Design Pattern mirato al riuso del codice e fornisce una soluzione “collante” tra due oggetti non progettati per lavorare assieme
- Si pensi, ad esempio, ad un software di comunicazione per il quale è richiesto il supporto di un nuovo protocollo: sarà compito di un adattatore i due componenti

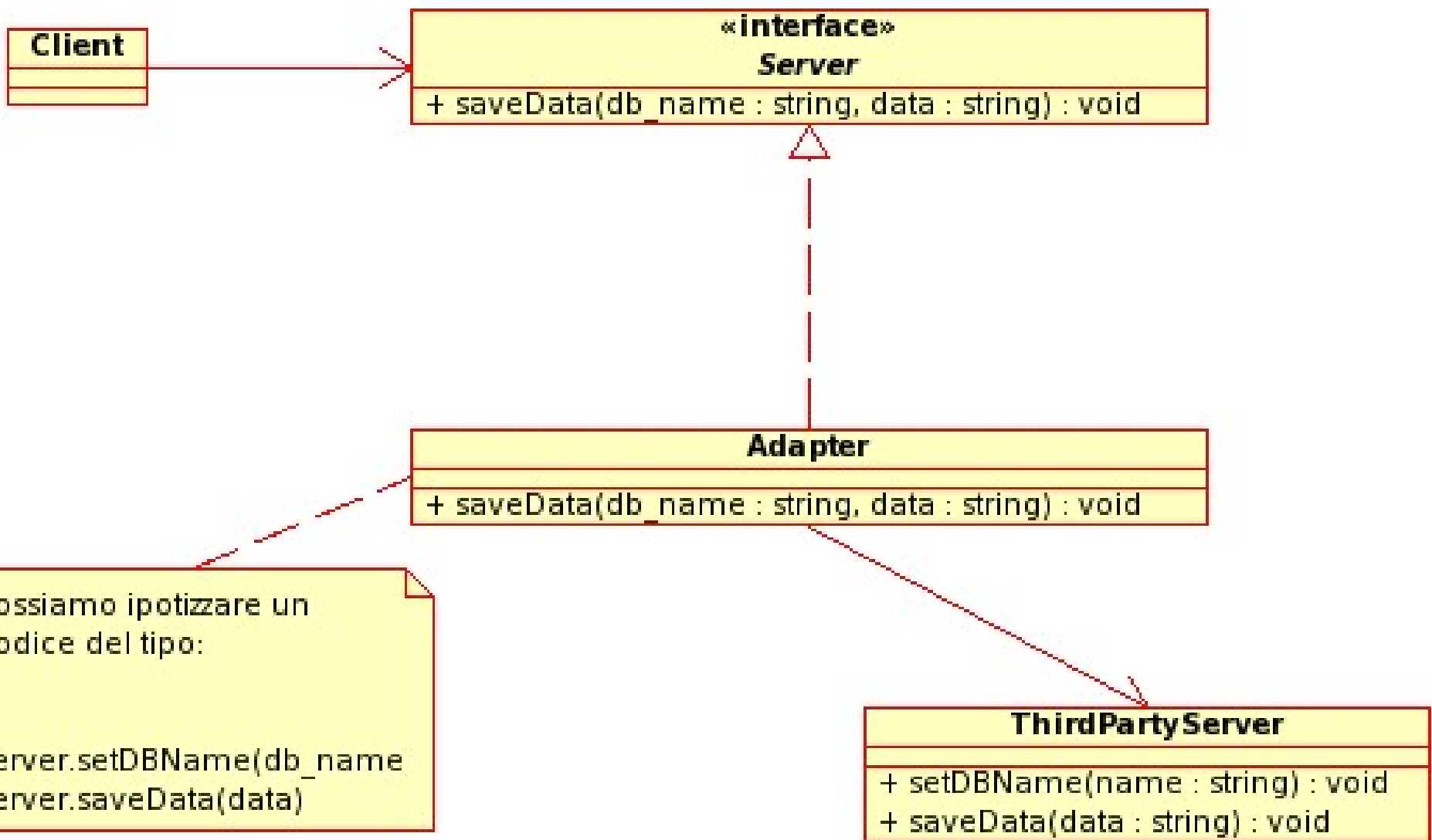
Adapter



- L'oggetto Adapter espone una interfaccia pubblica compatibile con quella attesa dall'oggetto *client* e invoca, eseguendo eventuali procedure accessorie, gli opportuni metodi sull'oggetto *server*
- Un Adapter che esporta un'unica interfaccia pubblica che colleziona funzionalità di diverse classi e' detto Facade



Adapter



Possiamo ipotizzare un codice del tipo:

```
server.setDBName(db_name)
server.saveData(data)
```

Adapter in J2SE



- Un esempio di Adapter in J2SE e' presente nella gestione I/O. Dispongono di alcuni adattatori:
 - DataInput/OutputStream: gestione tipi primitivi e stringhe, generando un opportuno stream di byte[]
 - Writer e Reader si occupano della gestione dei caratteri nei formati Unicode e UTF-8

Adapter in J2SE



- L'esempio più forte di adozione del DP Adapter è il framework della Java Native Interface (JNI):
 - Gli strumenti forniti con il JDK permettono di creare degli Adapter tra codice nativo (librerie a caricamento dinamico scritte con qualsiasi linguaggio) e il codice Java in esecuzione sulla JVM
 - Un blocco di codice intermedio si occupa di tradurre le metodi e oggetti Java nei corrispondenti nativi

Adapter: esercizio



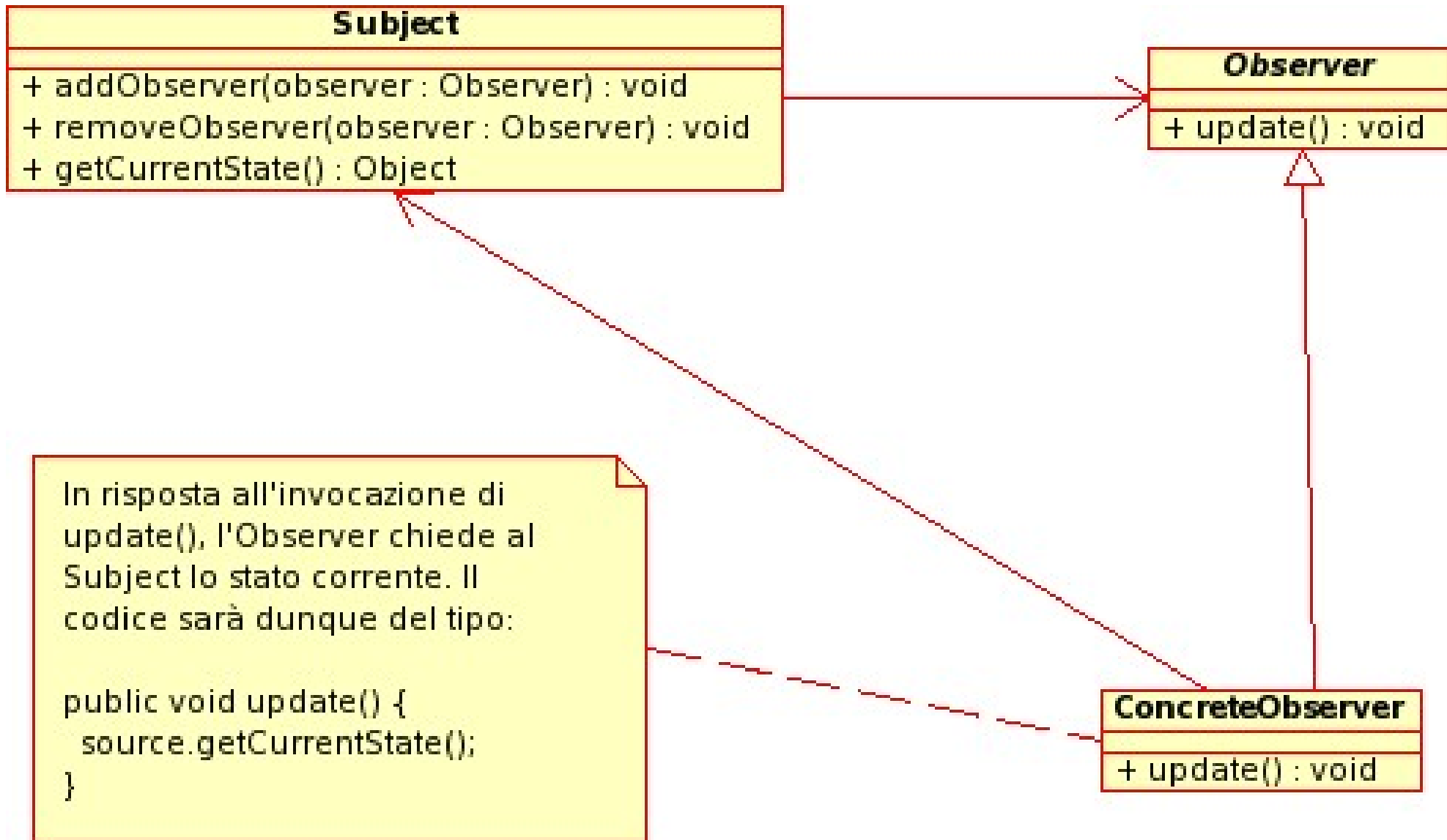
- Scrivere un Adapter che permetta di adattare un `java.lang.Integer` per... gestire i numeri romani!

Observer



- **Obiettivo:** notificare le modifiche allo stato di un oggetto
- Diversi i casi in cui questo DP è utilizzato efficacemente:
 - Gestione eventi (mouse, tastiera) delle interfacce grafiche
 - Propagazione di variazione su strutture dati
 - Implementazione di sistemi asincroni (non bloccanti)

Observer

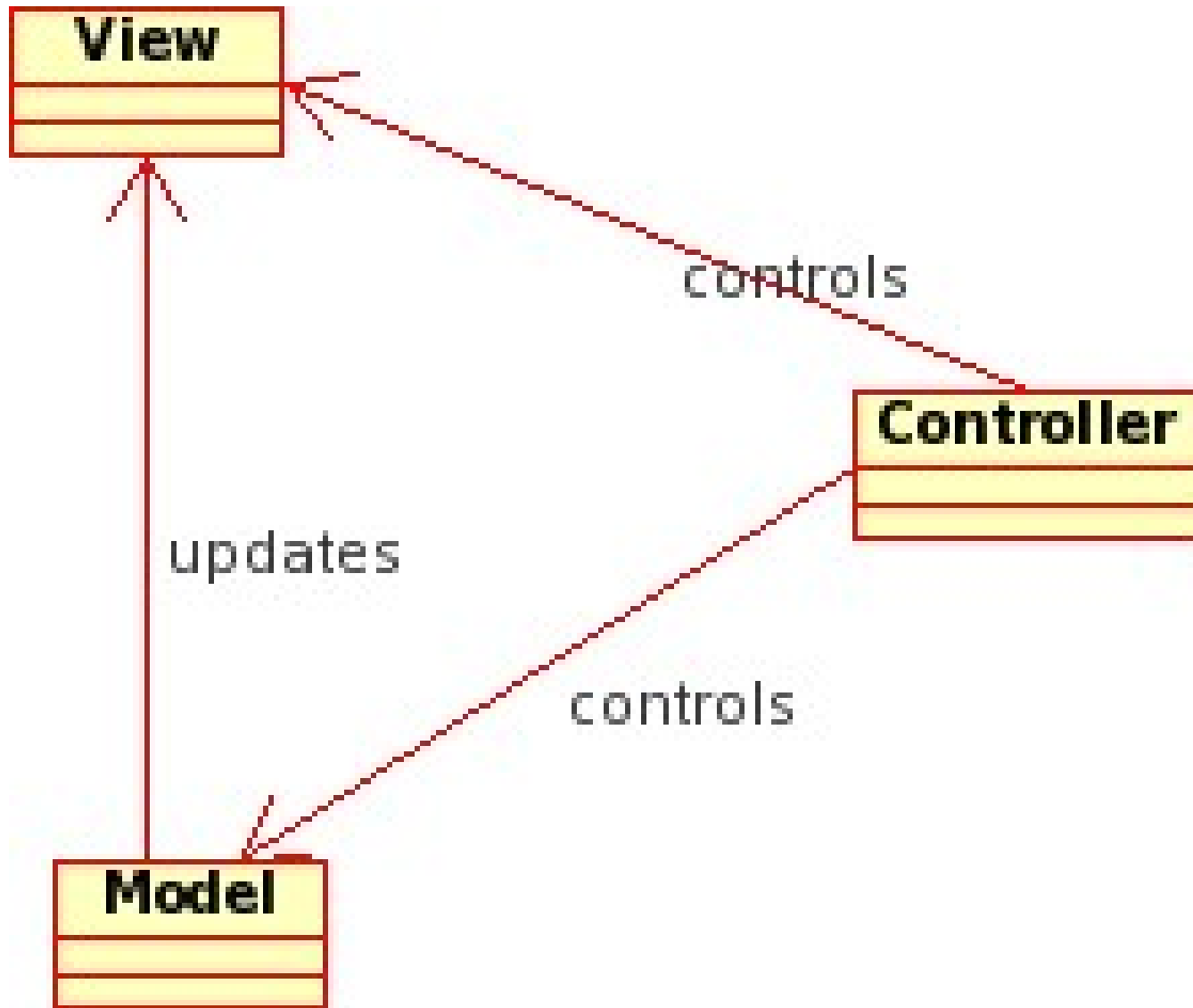


Model-View-Control



- Observer e' la base del Model-View-Control, un DP fondamentale per l'implementazione di complesse applicazioni
- MVC definisce:
 - Model: e' la descrizione dei dati gestiti dall'applicazioni
 - View: costituisce l'insieme delle rappresentazioni del modello (GUI, web service...)
 - Control: definisce la logica con cui le azioni sulle viste si realizzano in modifiche sul modello

Model-View-Control

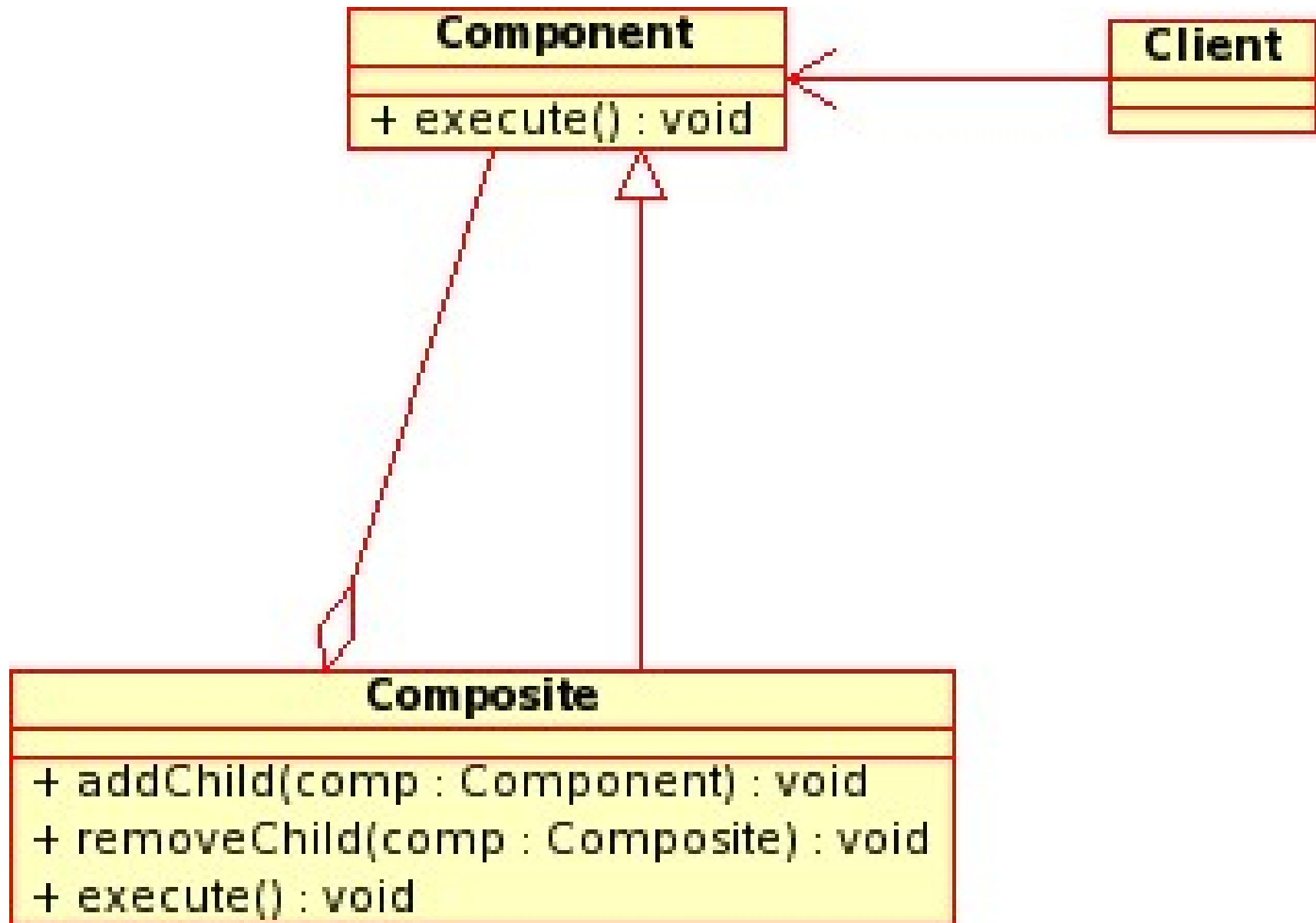


Composite



- **Obiettivo:** gestire efficacemente strutture composte (ad esempio, alberi)
- Permette di maneggiare in maniera uniforme contenitori e componenti elementari.
 - Componenti e Contenitori condividono la stessa interfaccia pubblica
 - I Contenitori hanno la possibilità di aggiungere nuovi Componenti e Contenitori

Composite



Composite in Java



```
public class Component {...}
```

```
public class Composite extends Component {  
    private Collection children;
```

```
    public void addChild(Component c) {...}
```

```
}
```

Composite e MVC in J2SE



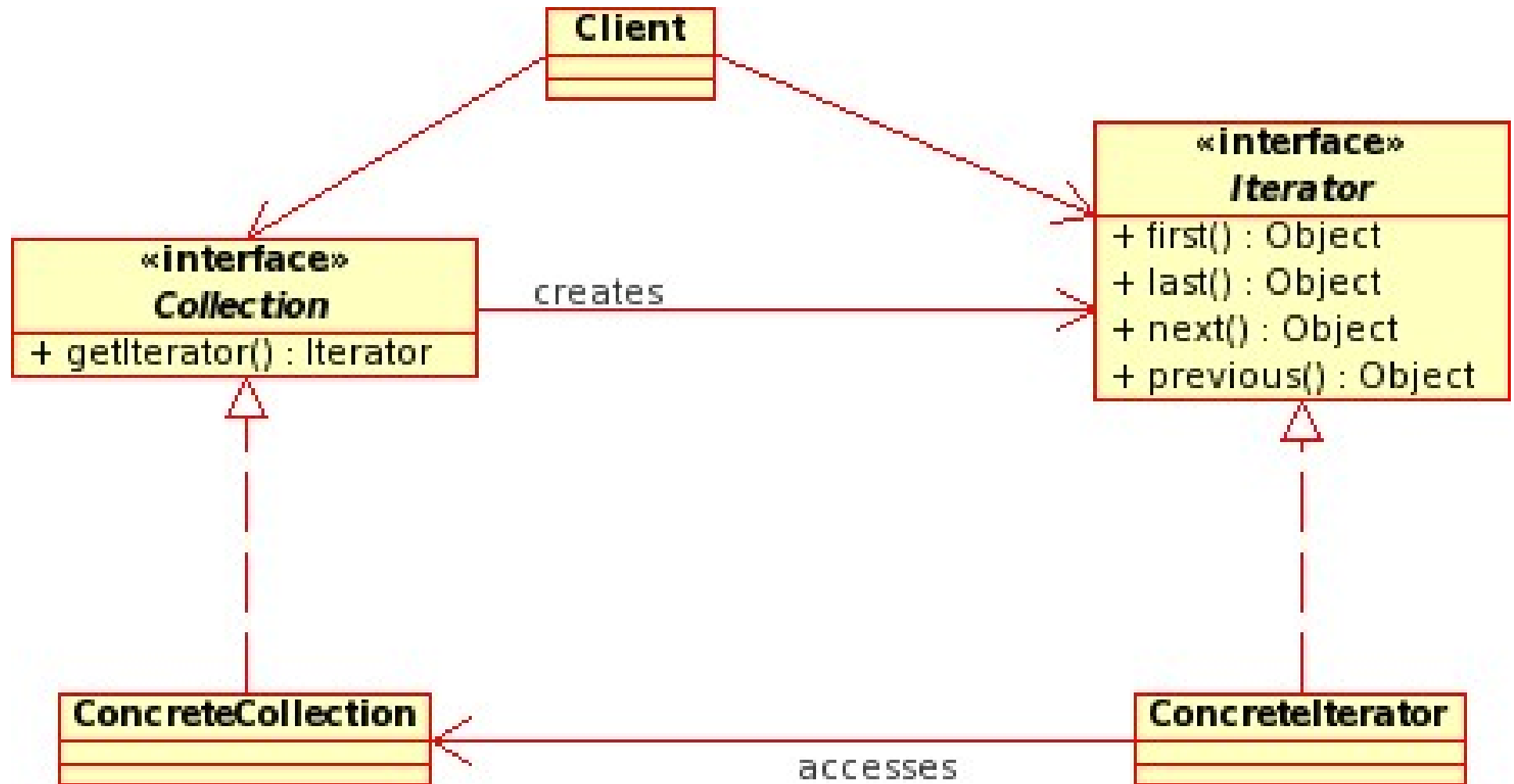
- Il DP Composite è fondamentale nel design delle interfacce grafiche
- I componenti delle librerie AWT e Swing sono modellati come Component e i pannelli sono modellati come Container (Composite):
 - Component: Button, Label, TextField...
 - Container: Panel, Frame, Window, Dialog...
- MVC modella la libreria Swing, dove ogni vista ha un corrispondente modello (si veda, ad esempio, JTable e TableModel)

Iterator



- **Obiettivo:** disporre di uno strumento generico per accedere sequenzialmente e ad una collezione di dati
- Strutture dati diverse (liste, vettori, vettori associativi, insiemi...) hanno strumenti di accesso completamente diversi e un cambio di struttura dati può richiedere la modifica del codice client: un Iterator semplifica notevolmente l'accesso ai dati, fornendo una interfaccia “standard” e generalizzata

Iterator



Iterator in J2SE



- La Collection API, introdotta con l'avvento della piattaforma Java2, fornisce una ricca insieme di classi per la gestione di collezioni di dati:
 - Vettori
 - Liste
 - Tabelle di hash
 - Insiemi
- L'accesso uniforme ai dati è ottenuto attraverso una istanza di Iterator

Template



- Supponiamo che sappiate che un algoritmo è composto da un certo numero di step eseguiti sempre in un certo ordine
- Tuttavia questi step possono cambiare nella loro implementazione a seconda del tipo di oggetto
- Esempio.
Sappiamo che un animale che vuol mangiare una banana in cima ad un albero deve eseguire sempre le stesse operazioni:

Template



```
public abstract class Animale{
    public void mangia(Frutto f){
        localizza(f);
        accedi(f);
        sbuccia(f);
        ...
    }
}
```

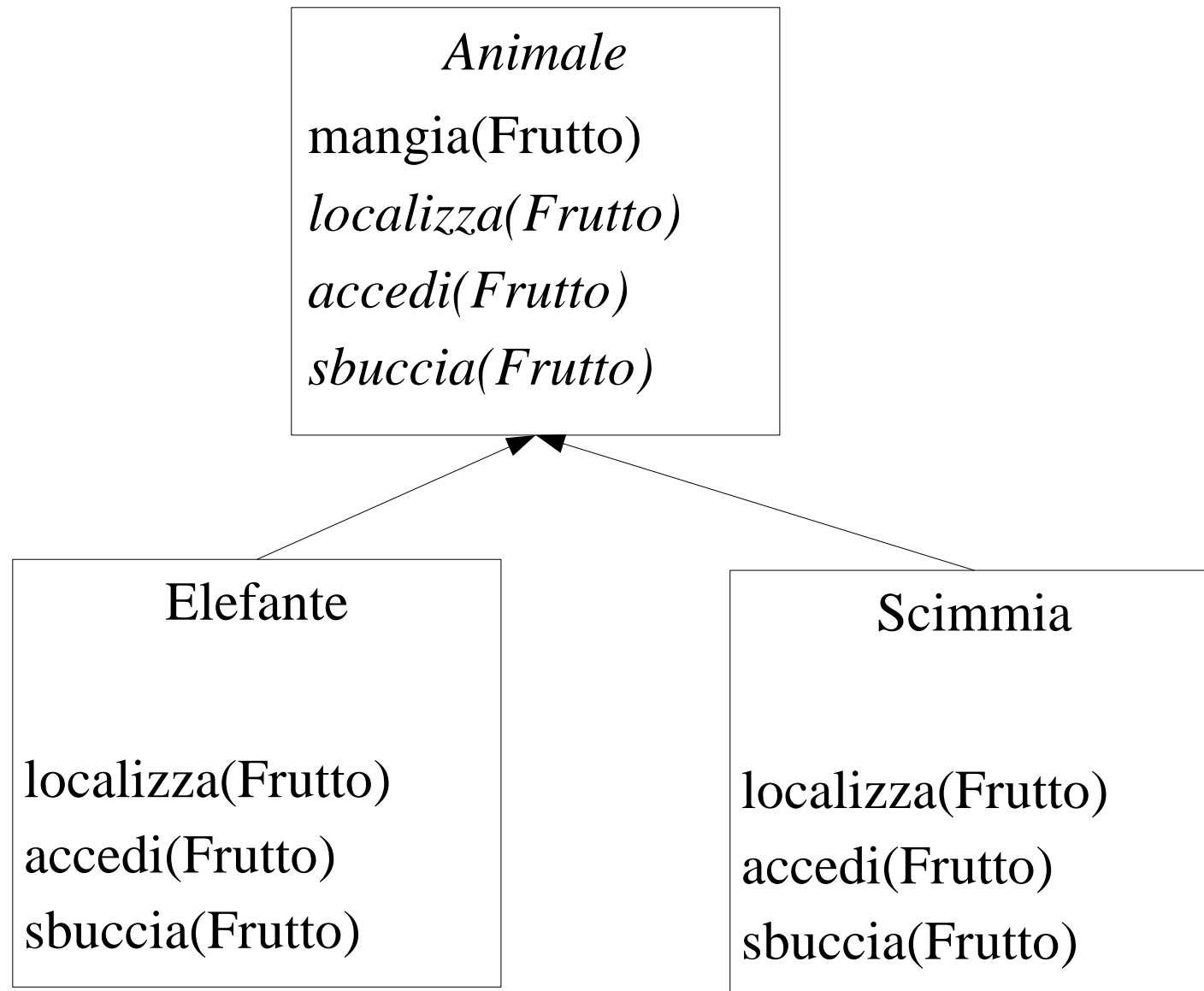
localizza, accedi, sbuccia sono tutti metodi astratti della classe Animale. Saranno le sottoclassi a implementarli.

Template



- Ad esempio, una scimmia localizza il frutto con la vista, accede ad esso arrampicandosi e lo sbuccia usando le “mani”
- Un elefante lo localizza con la vista, accede con la proboscide, lo sbuccia con la bocca sputando la buccia, e poi ingurgita cio' che resta per mangiarlo

Template

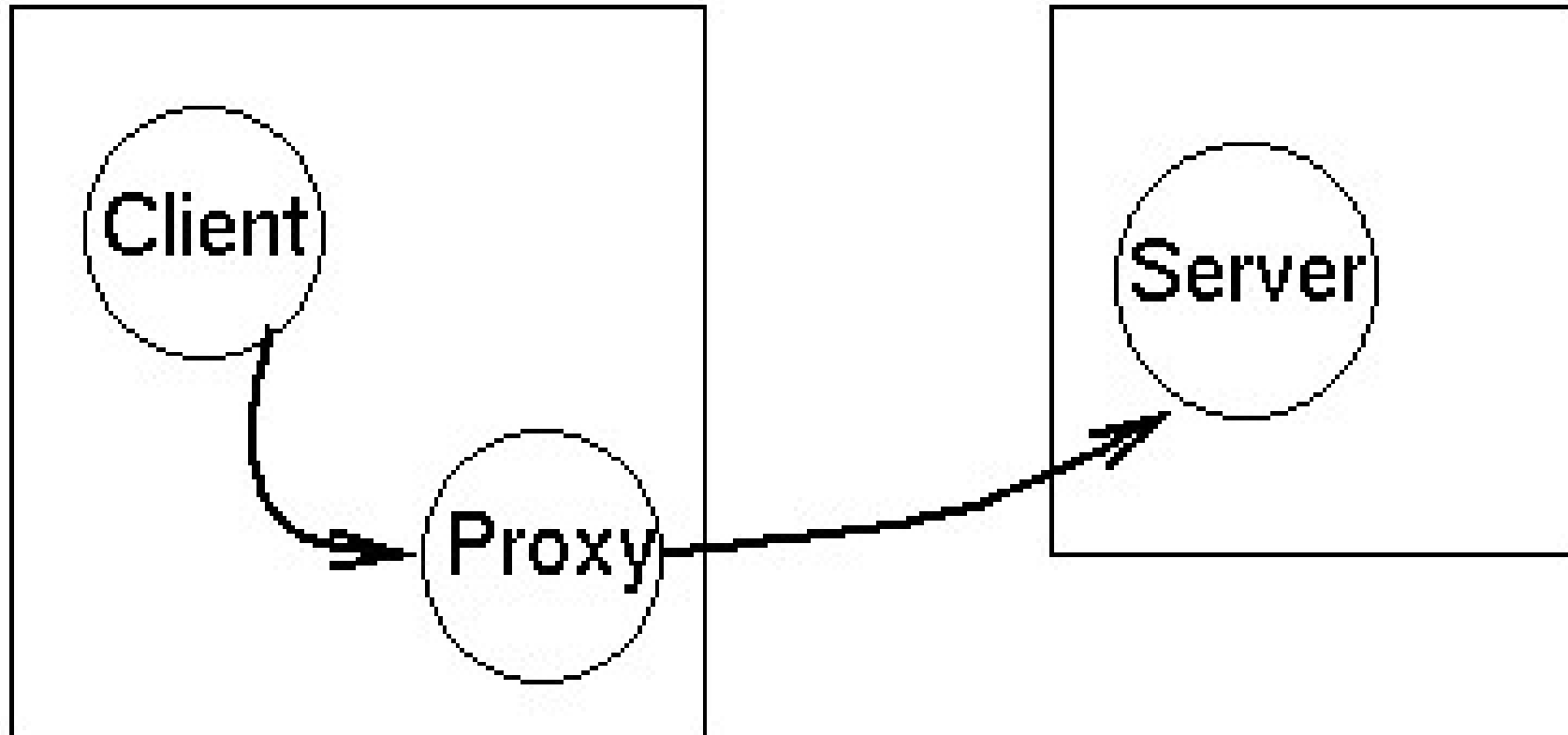


Proxy



- Problema: si ha la necessità di accedere alle funzionalità di un oggetto “inaccessibile” come se questo fosse un oggetto ordinario
- Per esempio l'oggetto è...
 - Esiste ma in un altro processo
 - E' ancora serializzato
 - E' compresso
 - Non e' ancora stato costruito ...

Proxy



Remote Proxy



- Fornisce un oggetto locale che rappresenta un altro oggetto remoto (in generale in un diverso address space) con il quale e' collegato

Virtual Proxy



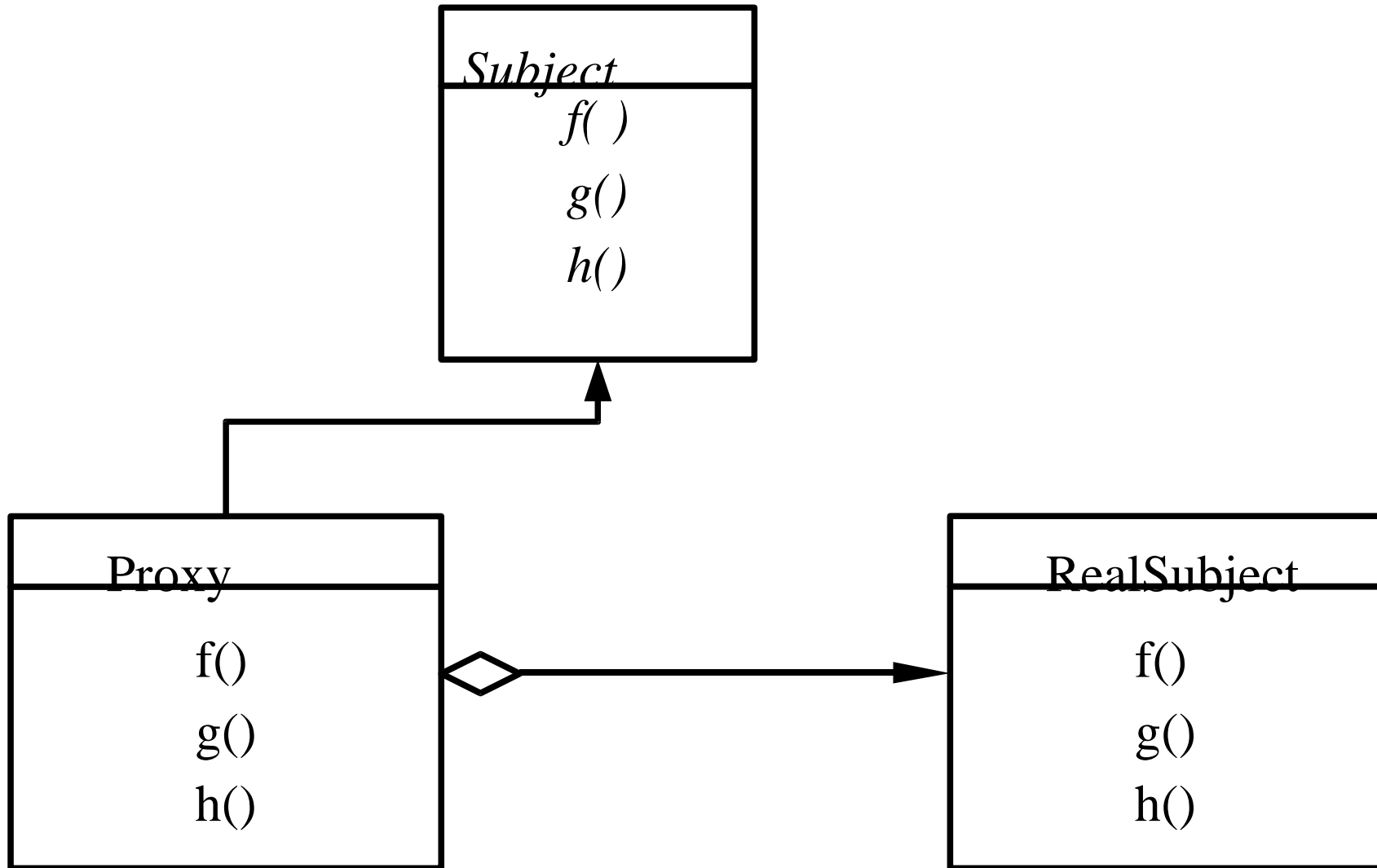
- Crea oggetti on demand solo se veramente necessario. Altrimenti cerca di sostituirsi in tutto e per tutto al suo subject.

Protection Proxy

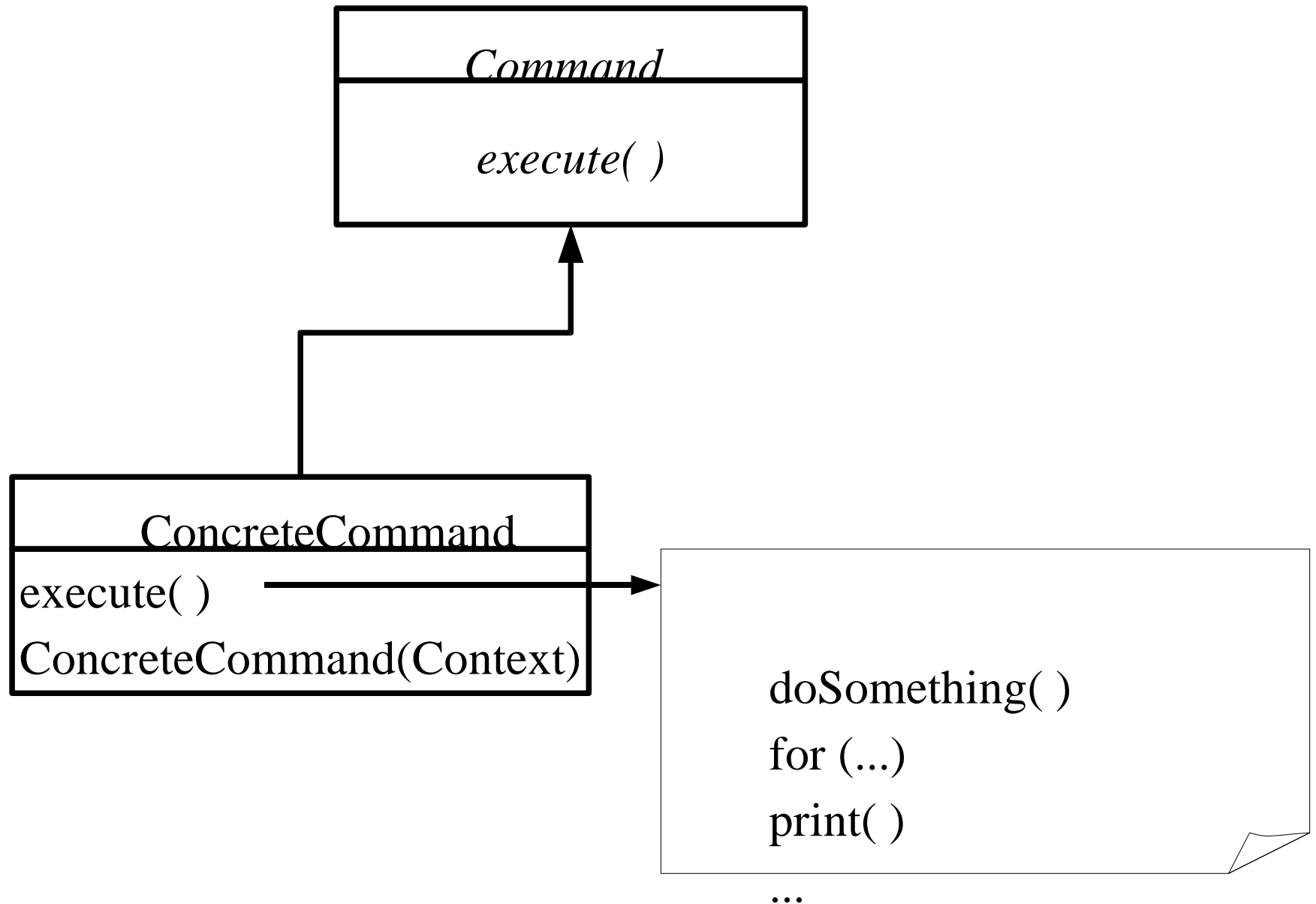


- Aggiunge dei controlli all'esecuzione di metodi sul subject.
- Ad esempio se il subject è il vostro conto in banca il proxy potrebbe chiedervi il PIN prima di eseguire l'operazione PRELIEVO

Proxy



Command



Command



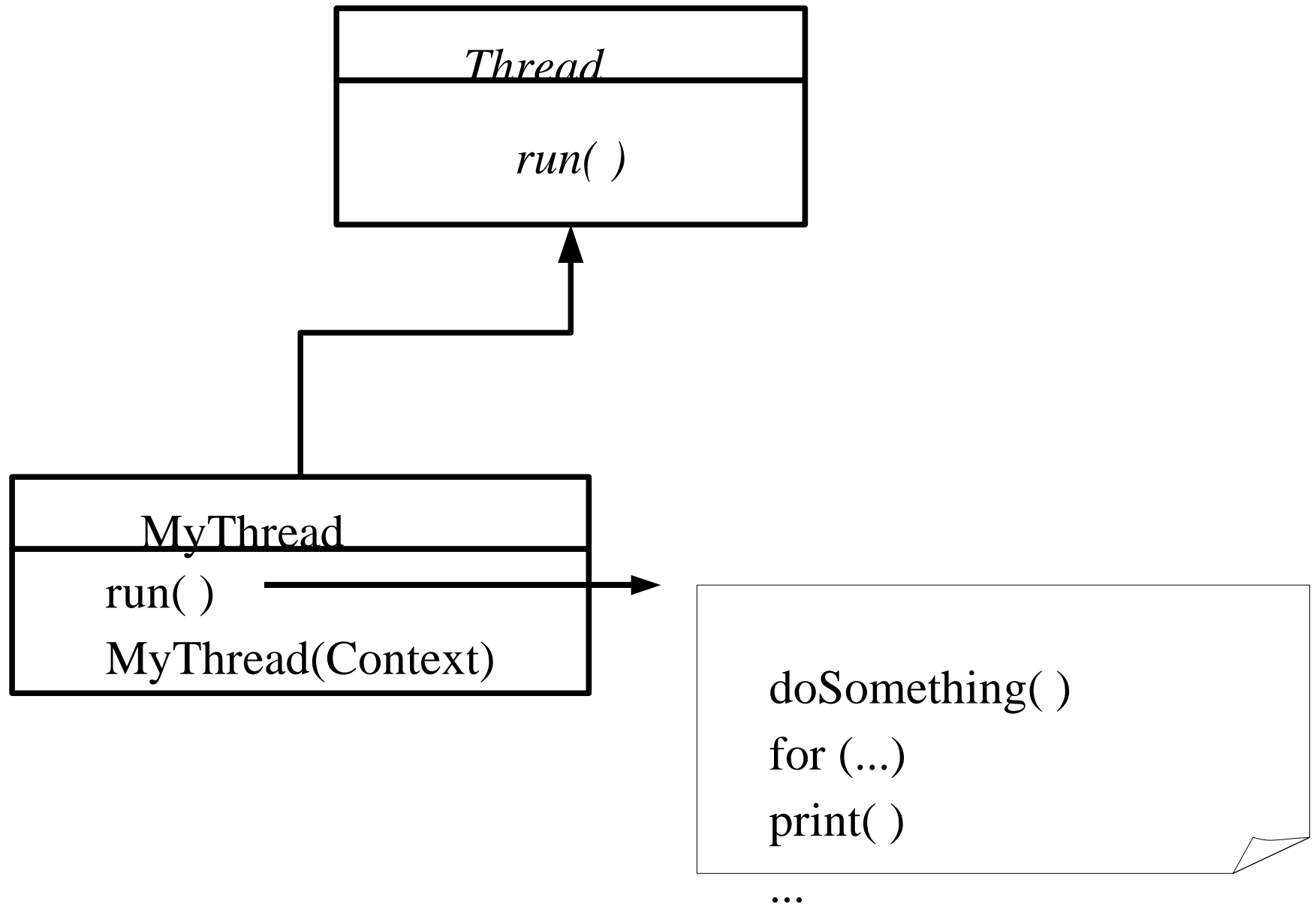
- *Problema*: Schedulare delle azioni per l'esecuzione temporizzate
- *Problema*: Una GUI in cui lo stesso comando puo' essere inviato in diversi modi (bottone, menu, ...). Tutti i modi sono inibiti se il comando non è disponibile
- *Problema*: mettere in una coda una serie di azioni
- *Problema*: UNDO

Command



- Tutti questi problemi trovano una soluzione comune nel Command
- Il Command incapsula un'azione in un oggetto. Questo consente di costruire code di Command, di associare un Command a delle GUI, di schedulare l'esecuzione temporizzata

Command e Thread



Command: esercizio



- Costruire una classe Task che ogni 10sec viene messa in esecuzione.

*Suggerimento: Utilizzare le classi
`java.util.Timer` e `java.util.TimerTask`*

State

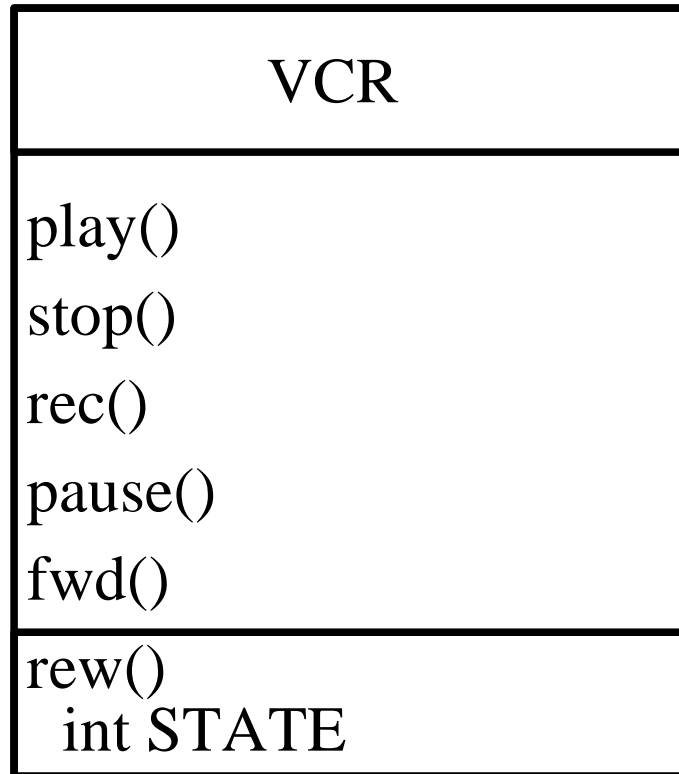


- Problema: simulare un video-registratore
- I comandi possibili sono play(), rec(), pause(), fwd(), rew(), stop().
- Gli stati possibili sono:
 - STILL, PLAYING, FASTPLAYFW, FASTPLAYRW, PAUSED, RECORDING, REWINDING, FORWARDING,

State



- Inizio con una classe VCR.



State



- Il problema è che in ogni metodo avrò dei costrutti molto spaghetti-code:

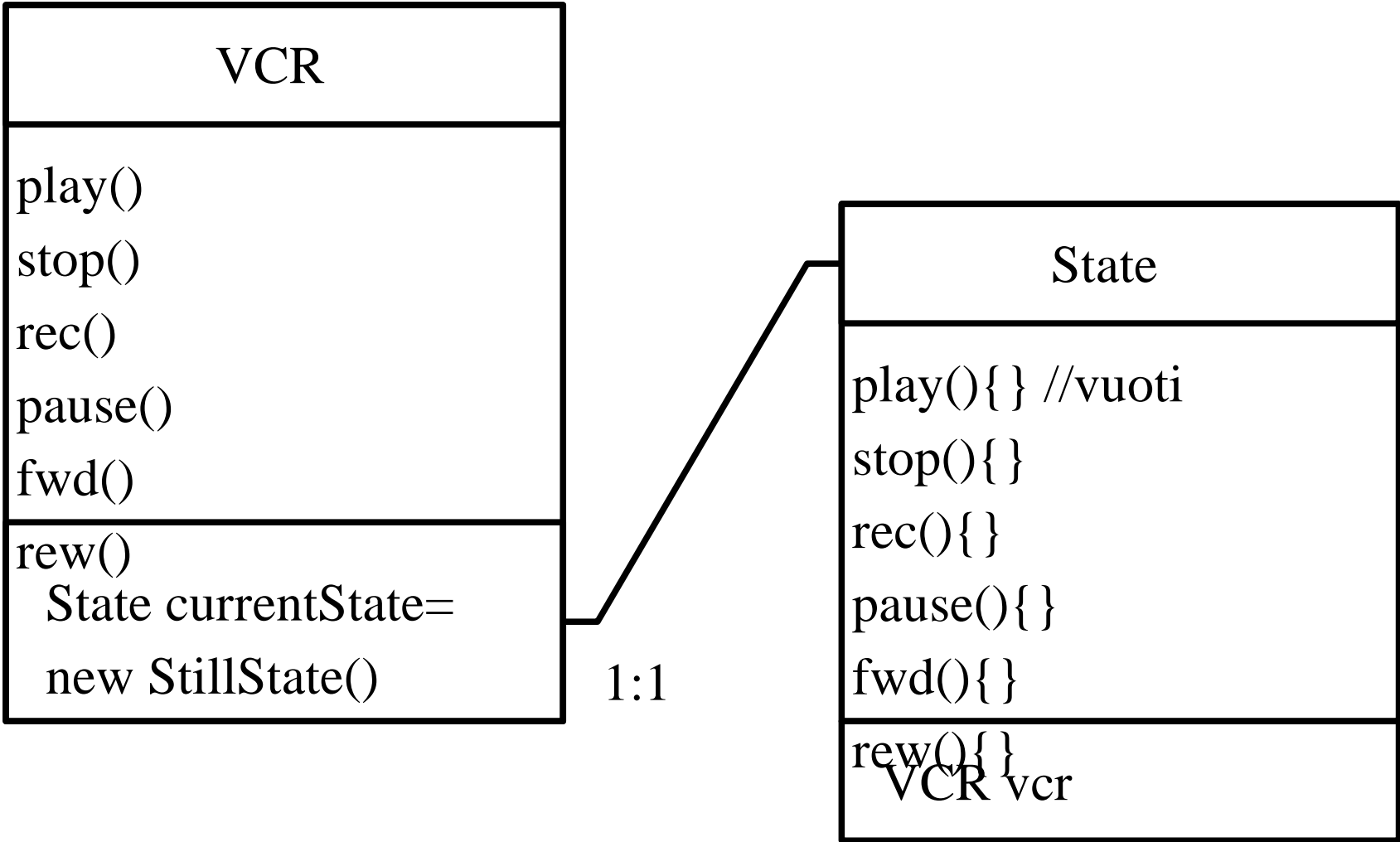
```
public void play() {  
    if (STATE == PLAYING)  
        //do nothing  
    else if (STATE == STILL)  
        startTapeEngine();  
    ...  
    else if (STATE == RECORDING)  
        ...  
}
```

State

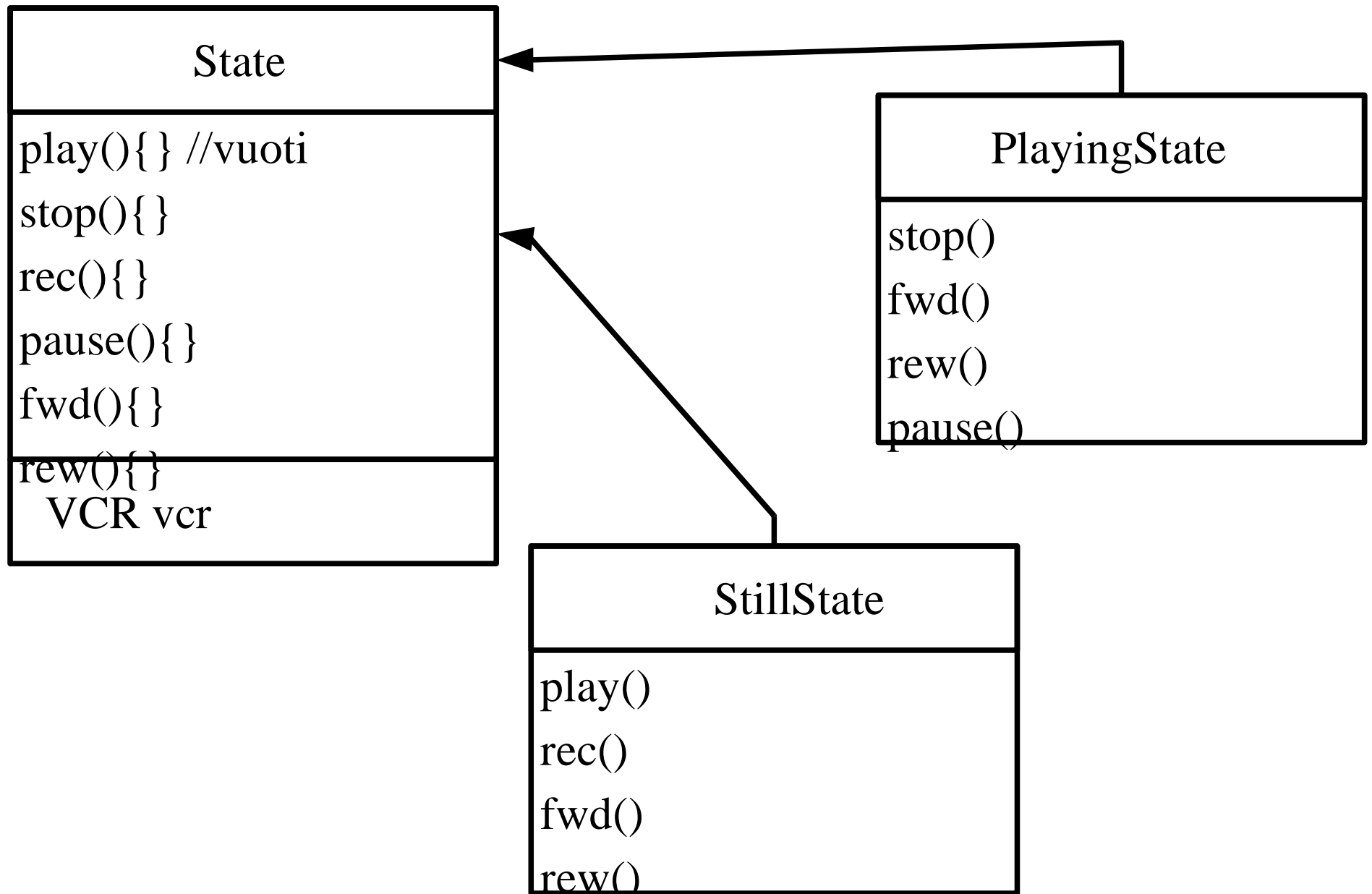


- Cerchiamo un approccio più Object-Oriented

State



State



Reflection



- La riflessività è la capacità del codice di eseguire una “computazione” su una “computazione”
- In altre parole: dato un programma, questo è riflessivo se è in grado di estrarre informazioni sulla sua struttura
- La riflessività può essere anche “attiva” se il programma può anche cambiare la sua struttura oltre che “capirla”

Reflection: esempio



```
Frame f=new Frame("Pippo");
```

```
f.setVisible(true);
```

 ... è equivalente a:

```
Class myclass = Class.forName("java.awt.Frame");
```

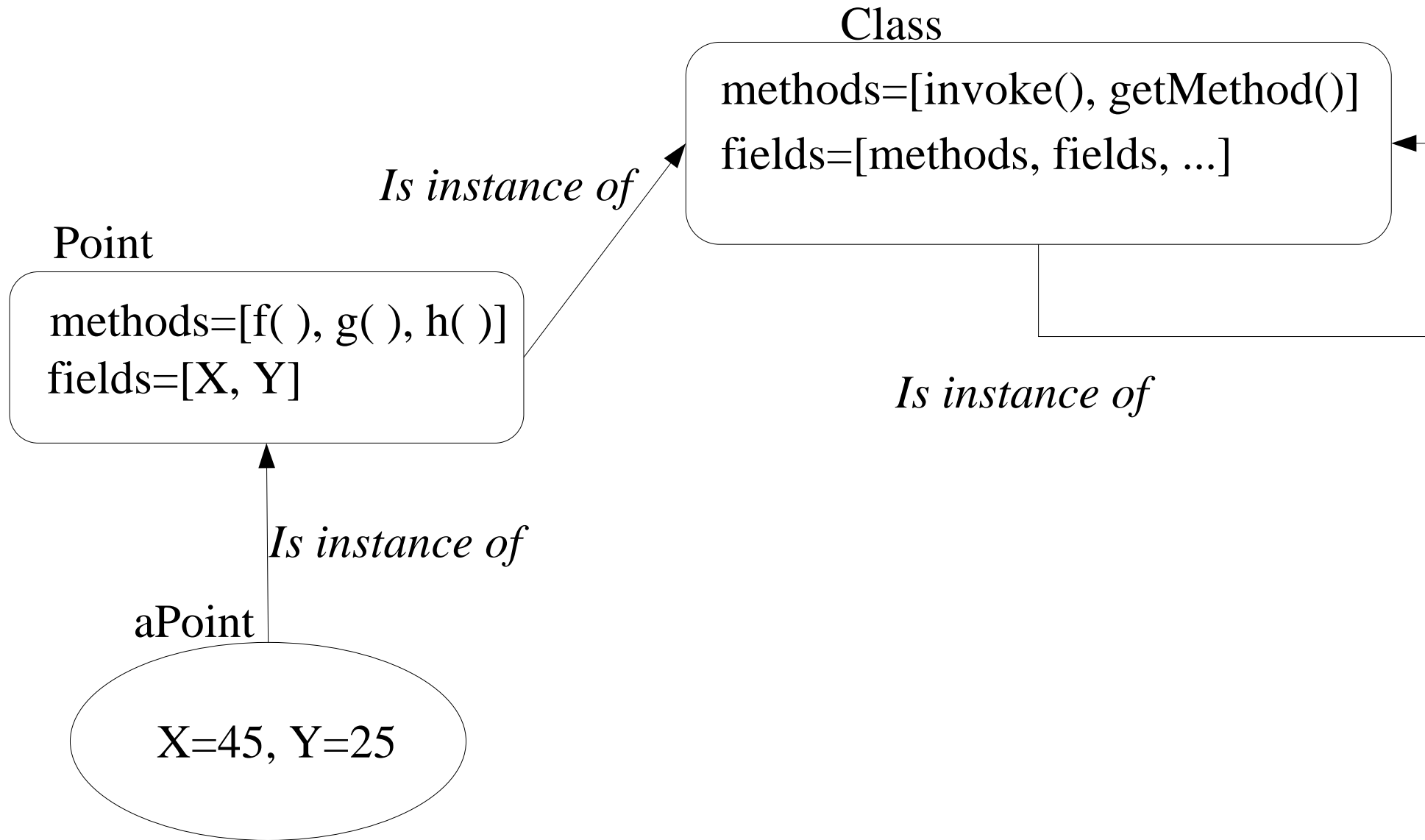
```
Constructor constructor =  
myclass.getConstructor(new  
Class[]{String.class});
```

```
Object o = constructor.newInstance(new  
Object[]{"Pippo Riflessivo"});
```

```
Method m2 = myclass.getMethod("setVisible", new  
Class[]{Boolean.TYPE});
```

```
m2.invoke(o, new Object[]{new Boolean(true)});
```

Reflection: classi e metaclassi



Reflection API



- Vedere documentazione SDK

Reflection API: esercizio



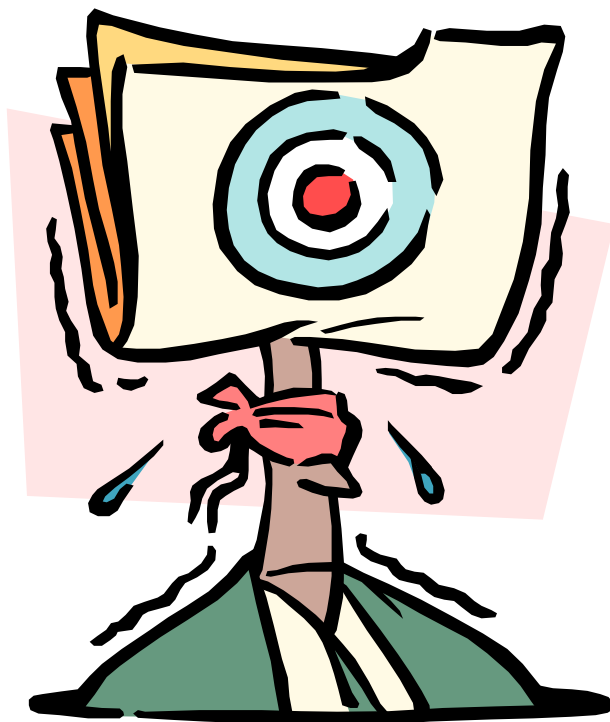
- Scrivere un programma riflessivo in grado di caricare una classe digitata dall'utente, stabilire se è in grado o meno di stampare un testo attraverso l'analisi dei suoi metodi.

Suggerimento: leggere i suoi metodi e vedere se ce n'è qualcuno che inizia per "print" e che ha tra i suoi parametri una stringa di testo. Da fare su Eclipse.

Domande?



CRS4 - <http://www.crs4.it>

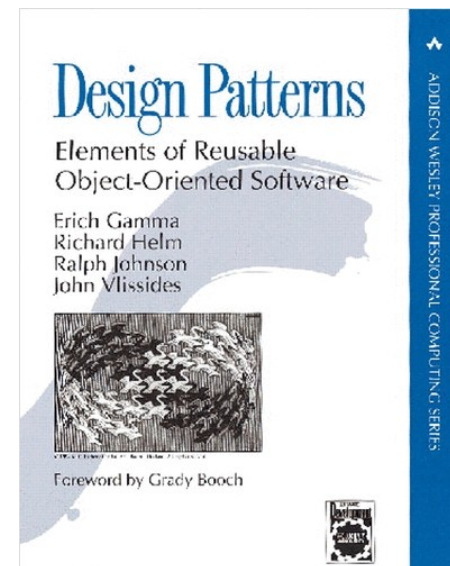


Davide Carboni - Stefano Sanna
 {dcarboni|gerda@crs4.it}

Bibliografia



- **Design Patterns**
Elements of Reusable Object-Oriented Software
E. Gamma, R. Helm, R. Johnson, J. Vlissides
Addison-Wesley, 1994
ISBN: 0201633612



Conclusioni



- I DP sono la formalizzazione di buone soluzioni a problemi generali che possono presentarsi nella progettazione/sviluppo di un software
- I DP forniscono **una possibile** soluzione ad una classe di problemi
- L'uso esagerato/sistematico dei DP puo' portare ad un peggioramento della qualita' del codice (*non facciamo la classe String come Composite di caratteri e stringhe, a loro volta generati da una factory...*)

Grazie... :-)



Copyright (c) 2004-2005 CRS4

Scritto da Davide Carboni e Stefano Sanna

è garantito il permesso di copiare, distribuire e/o modificare questo documento seguendo i termini della Licenza per Documentazione Libera GNU, Versione 1.1 o ogni versione successiva pubblicata dalla Free Software Foundation. Una copia della licenza in lingua italiana è disponibile presso:

<http://www.softwarelibero.it/gnudoc/fdl.it.html>

Davide Carboni - Stefano Sanna
{[dcarboni](mailto:dcarboni@crs4.it)|gerda@crs4.it}